Sopor: An extensible watchOS application for sleep session recording

Candidate: hwbakker

Master's Thesis Spring 2020

Blank page.

Abstract

Most medical examinations require a user or patient to be present at a hospital to be examined. However, with technology being increasingly more available and capable we can move this examination closer to the patient's home, thus reducing both the effort required by the patient and the strain and stress on the healthcare system. In the CESAR project, our goal is to increase the percentage of diagnosed obstructive sleep apnea cases, reduce the time it takes to get a diagnosis, and allow user-friendly and cost-efficient tools to be used for diagnosis at home.

Mobile wrist-worn devices are becoming increasingly more powerful and they are equipped with a growing number of on-device sensors. This paper implements an extensible watchOS application, allowing users to collect data from the sensors on the watch. To achieve this we designed and implemented an application named `Sopor` which collects and processes data collected by the watch. The application collects data from the sensors on the watch and passes them on to a sink which allows events to be aggregated, stored locally or offloaded to an online location. In addition to `Sopor`, an application named `Virga` is developed, which allows data stored online to be retrieved and processed on a computer. `Virga` connects to an iCloud database and downloads data stored by `Sopor`.

Several experiments are performed showing `Sopor`'s battery usage, resource efficiency, and extensibility. The experiments show that the application is stable and suitable for long-running sleeping sessions. When `Sopor` runs on the watch, results show that the application can run for eight hours depleting less than 50% of the battery. Experiments show that the CPU and memory usage on average is low and predictable. The final experiment performed shows that the application is extensible without requiring global knowledge of the application. The thesis concludes with some open problems and ideas for future work.

Blank page.

Acknowledgments

I would like to express my great appreciation to my supervisor Dr. Thomas Plagemann. His door is always open and he is always ready for a good discussion. Your thoughts, insights, and feedback have been invaluable. Thanks for allowing me to work with and contribute to the CESAR project.

I would like to thank my partner, Michelle, for supporting and pushing me through the development and writing of this thesis. Thanks for helping me with planning and developing a strategy on how to approach the subject.

I also want to thank my friends, family, and my co-workers for many interesting discussions around different aspects of the implementation of Sopor and Virga.

This project would not have been possible without your help.
Thank you.

Blank page.

Contents

List of Figures

List of Tables

List of Listings

Blank page.

Part I

Introduction

1 Introduction

Sleep apnea is a disorder that has become a significant health issue for many individuals all over the world. It is a disorder that causes breathing stops or shallow breathing during sleep [1]. The most common type of sleep apnea is Obstructive Sleep Apnea (OSA), and is caused by obstructions in the patients' airway [2]. Individuals are often not aware of their disorder, and it is estimated that 80% of the cases go undiagnosed [3]. In 2007, the World Health Organization (WHO), estimated that more than 100 million people are affected by this disorder [4]. A more recent study performed by ResMed in 2018, indicates that sleep apnea in general impacts nearly one billion people [5].

There are usually two ways to detect whether or not an individual suffers from OSA. The classical approach to diagnose OSA is to perform a polysomnography (PSG). This means to sleep overnight at a dedicated sleep center, where the medical doctors attach a vast array of sensors to the individual's body. Another is to get equipment by your physician in order to perform tests at home [2]. Both of these are high-effort options, meaning that they do require individuals to communicate and often take a trip to their medical practitioner, in addition, these options are expensive. There are few, if any, low-efforts options for individuals to use at home to detect occurrences of OSA.

In recent years, there has been a tremendous increase in processing power on mobile phones and wearable devices. By utilizing wearable devices and applications for these devices it is possible to collect huge amounts of biometric data. The CESAR project at UIO[1] aims to provide alternatives to performing diagnosis using PSG with these kinds of devices. This approach may reduce the threshold to perform clinical diagnosis of OSA and reduce the time it takes to get a diagnosis for sufferers of OSA.

During the research phase of the project, three smartwatches are investigated, 1) Garmin Fenix 5, 2) Fitbit Ionic and 3) Apple Watch. The research find that the different watches excels in different areas, and it is decided to move forward with the Apple Watch. In addition to being the watch that excels in most areas, Apple has a focus on health and while not released, it is rumored that the company has a strong focus on sleep tracking [6]. This opens up possibilities of more sensors becoming available directly on the watch, which should be accessible to developers through APIs.

The main motivation of this thesis is to develop a simple-to-use extensible watch

[1]Information about the CESAR project can be found at https://www.mn.uio.no/ifi/englis h/research/projects/cesar/

application that collects biometrical data, thus paving the road for developers to create more sophisticated sensor-wrappers and *sinks*[2]. For the CESAR project, this thesis aims to create a foundation that future researchers can build upon. In addition to developing the application, another goal of the thesis is to expand our knowledge of what the small devices we wear on our wrists are capable of doing.

1.1 Problem statement

Smart wrist-worn devices are becoming more popular every year. Small devices collecting and sending data to a smartphone companion app or even small computers doing all the work themselves are here to stay. Making use of these devices and showing that these devices can be used in a research-setting is valuable for future researchers. These small computers on our wrists are used for everything between notifying us of new incoming messages to collecting valuable health information during the day. During the night, most of these devices have to be charged. While sleeping, there are interesting data-points that can be collected and inform us of a user's health situation. These data-points are not collected if the device is charging and not worn during the night, thus we are losing a lot of valuable information. Therefore, we look into designing and implementing a watchOS application, `Sopor`, that collects and processes sensor data from the watch. From this we define three main goals that the thesis need to cover:

Goal 1 Creating an application that can collect data without using too much battery, thus allowing the user to top-up their battery once waking up and using the watch as they normally would.

Goal 2 Integrate multiple sensors from the watch to the application and allowing it to easily be extended with new sensors.

Goal 3 Verifying that the data collected by the watch can be fetched from an online location to process the data.

These three goals are what we want to accomplish in this thesis. Satisfying them will give us an indication into how useful these wrist-worn devices are and how they can be used to further develop knowledge about a user's health situation.

1.2 Contributions

This thesis aims to demonstrate that it is possible to create an application that collects, processes, and stores data from sensors on the watch. It also demonstrates that a user can utilize the watch as he or she normally would with regards to battery life. The collected data is shown to be accessible from multiple online locations, paving the road for further analysis of sensor events.

[2]See Section 4.2.4 for a description of sinks.

The contributions from `Sopor` is a modular and extensible Apple Watch application capable of collecting sensor data. It allows collected events to be sent to multiple sinks to be processed and stored. In addition, `Sopor` performs its operations with a focus on resource efficiency especially with regards to battery usage. With this foundation, researchers may improve and develop new features for the application allowing us to learn more about the potential of wrist-worn devices in a healthcare setting. `Virga` contributes by allowing researchers to download data from iCloud to their local machine. This is useful as it empowers researchers to analyze, process, and gather insights into the collected data-points. For CESAR, the created applications act as a foundation for future researchers and developers. They support the project by creating a low-threshold and non-intrusive way to extract events which can be used as a part of more complex event processing and analysis.

1.3 Thesis structure

In this thesis, the foundation of an extensible data collection application for Apple Watch is created. The application collects and processes data from sensors on the watch and sends them to online storage locations. The application can be extended with new input and output sources. The thesis is divided into five parts and seven chapters.

The motivation, contributions, and thesis structure are outlined in Part 1. Part 2 contains one chapter discussing physiological computing for both watch- and non-watch alternatives. Further, a comparison between the three considered watches is performed comparing the hardware with each other focusing on sensor accuracy, efficiency, and market share.

Part 3 gives an introduction to the Apple Watch operating system, watchOS. First, an overview of the general application architecture is given, then the application lifecycle, communication, and energy consumption are discussed. The chapter concludes with an overview of the available sensors and how to distribute applications on the Apple platforms.

The fourth part consists of two chapters. The first chapter goes deep into the design of the application, starting with a requirement analysis. The analysis is done in order to highlight what the application needs to be able to perform. Next, the application components are separated out into different concerns, where all components solve their own issues. The design iterations are shown next, starting with the initial design and ending with the final design implemented in Chapter 5. Finally, Chapter 4 ends with a discussion of the data formats that can be used in the application. Chapter 5 starts with an overview of the different application components and their responsibility. Next, the different concerns identified in Chapter 4 are implemented. Chapter 5 concludes with a few miscellaneous items including how to use multiple sinks, discussing long-running sessions, and detailing `Virga` the data processing application that can process data stored online using Apple's CloudKit framework.

The final part, Part 5, consists of two chapters. Chapter 6 evaluates the implemented application. The chapter is divided into two sub-sections where the first one involves performing three practical experiments showing that the application performs according to the hypothesis outlined in the problem statement of the thesis. The second part of Chapter 6 examines the requirements outlined in Chapter 4 and discusses to what extent they have been met. The final chapter, Chapter 7, wraps up and concludes with some open problems and outlines items for future work.

Part II

Physiological Computing

2 Physiological computing

Physiological computing concerns technological systems and devices that record quantitative data generated by human use, and that can display the data on the user interface of the device or connected devices. Activities like walking, thinking, and breathing are activities that can be recorded and data-points can be generated which in turn can be analyzed. Physiological computing can be seen as an extension of physical computing. While physical computing concerns primitive events such as movement, physiological computing concerns the more delicate aspects such as heart rate or oxygen saturation [7]. [8] argues that physiological computing has enormous potential to revolutionize human-computer interaction (HCI). Computers have seen an exponential increase in computing power during the last few decades, often referred to as Moore's law [9]. While the exponential curve has reduced somewhat in later years [10], there are still performance and capability performances in computing devices every year. In terms of physiological computing, power and battery improvements have allowed the development of smartwatches and other physiological computers.

2.1 Physiological computing alternatives

The following section presents an assortment of different non-watch and watch alternatives with respect to physiological computing. The non-watch alternatives are provided in order to give increased insights to the reader about different physiological options. In the end, the selection as to which watch to move forward with in this thesis is presented.

2.1.1 Non-watch alternatives

Multiple non-watch alternatives are interesting to discuss, the following are low-cost physiological computing devices that have been considered and investigated by [7] and [11]. All the devices are built on open platforms and allow the sensors to be used in different projects as discussed in [11].

BITalino is an open-source, low-cost toolkit used to create prototype applications using biosensors collecting body signals. This small programmable computer allows connecting multiple sensors to measure body signals. It features Bluetooth for communication and can be attached to a rechargeable battery [7]. The BITalino is a hardware unit where you need to make use of its software development kit (SDK), and do some programming before you can get data from the device. Another option is hardware from Shimmer, which produces wearable wireless sensors. These sensors are used to monitor different health aspects, their technology allows for simple capture and transmission to other devices such as a phone or computer [7]. In addition, it also features an SDK that can be used to integrate with Shimmer devices and other hardware devices such as phones [12].

These non-watch alternatives are devices that can be used to create prototypes and test new sensors, however, they are not low-threshold options for the average Joe, as they require a rather huge investment in terms of time and money as compared to something like a smartwatch.

2.1.2 Watch Alternatives

For this thesis, three watches are considered; 1) Garmin's Fenix 5, 2) Fitbit's Ionic, and 3) Apple's Apple Watch. These were chosen due to their feature set, availability, and development SDKs.

2.1.2.1 Garmin Fenix 5

The Garmin Fenix 5 is the watch with the highest starting price of the three watches, with a starting price of $500. Released in June 2018, it packs a lot of sensors and is a device focused on outdoor activities [13]. It has a round display and is advertised with having more than 20 hours of battery life with GPS activated. The watch is rather big with a volume of 0.034 cubic meters, and with a weight of 85 grams [13].

Figure 1: Garmin Fenix 5 [14]

With the Garmin Fenix 5 you can, as a developer, get access to among other sensors the heart rate- and accelerometer sensor. Garmin sells a lot of different devices with different features, and following their documentation is it easy to understand which watches have access to the different sensor APIs. When creating applications for the Garmin watch, you essentially create two separate applications, one for the watch and one for the phone.

According to an estimation by the International Data Corporation, IDC, Garmin shipped 1.5 million wearable devices in Q2 of 2018 [15]. This number consists of all wearable devices that Garmin sells; thus, the actual sales figure of the

Garmin Fenix 5 is lower. Although these estimated numbers have to be taken with a grain of salt, they give a good indication of the distribution of sales by the different companies. In IDC's Q4 2018 report Garmin was not mentioned among the top brands [16].

2.1.2.2 Fitbit Ionic

The Fitbit Ionic was released on October 1st, 2017. The starting price of the watch is currently $270, the cheapest of the three. Some of its advertised features are multi-day battery life, heart rate sensor and built-in GPS, as well as incorporating wallet-free payments [17].

Figure 2: Fitbit Ionic [18]

Creating applications for Fitbit is done using the Fitbit Studio application. The development language is JavaScript combined with CSS and HTML. It can resemble Node.js or React [19]. Through the development API, you have access to the heart rate sensor which samples in near real-time, as well as the accelerometer and gyroscope. The sampling rate cannot be found on Fitbit's SDK website, however, on user forums, numbers ranging from 30 samples per minute to 10 samples per second are discussed [20].

According to IDC's estimates, Fitbit shipped 5.5 million units and their market share was estimated to be 9.4% in Q4 of 2018, down from 11.9% during Q4 2017 [16]. Again, these numbers are likely to be lower due to the fact that Fitbit has many products within the wearable category, and they do not reveal sales numbers for individual products [21].

2.1.2.3 Apple Watch

The Apple Watch was first released in 2015, since then there have been several new versions. The Apple Watch Series 4 and 5, features an ECG sensor that can indicate whether or not your heart rhythm displays signs of atrial fibrillation. Another addition of the Series 4 and the Series 5 is fall detection, which gives the watch the ability to contact emergency services and relatives if it detects that the owner has fallen and is unable to move. The smaller of the two Apple

Watches (40mm) weighs in at 68 grams and has a volume of 0.015 cubic meters [22], about half that of the Garmin Fenix 5.

Figure 3: Apple Watch [23]

The Apple Watch features amongst other things an accelerometer sensor, a heart rate sensor, and a microphone [22]. Access to these is provided through the SDK. With the Watch Connectivity framework that Apple provides you are able to implement two-way communication between an Apple Watch and an iPhone. The HealthKit framework makes it easy to gather sensor data on the watch and send the data to the phone for processing. Since the Apple Watch and the iPhone are programmed using the same programming languages, Swift and Objective-C, the implementation should be simpler as compared to the Garmin Fenix 5 and Fitbit Ionic. Regardless of the Apple Watch series, the development API is the same, disregarding some of the improvements and sensor upgrades in the latest models. The Apple Watch is capable of monitoring heart rate variability, however, the API is limited [24].

The analysis performed by IDC estimates Apple Watch shipments at 10.4 million Apple Watches during the Q4 quarter of 2018, maintaining its position at the top of the smartwatch market with 27.4% market share [16]. Healthcare providers are said to tinker with the idea of bundling Apple Watches with insurance packages, this could greatly increase shipments for the Apple Watch [25].

2.2 Comparison

Although little research has been put into measuring the accuracy of different smartwatches, the Apple Watch is seen to receive the lowest error rate [26] among a handful of smartwatches. One problem for this study is that none of the two other watches was included, however, a handful of other comparable watches was tested. Another test performed by [27], showed that the overall average heart rate variance of the Apple Watch vs. a regular chest strap was 0.67, compared to the Garmin Fenix 5 who has a variance of 1 and the Fitbit Ionic came out last with a variance of 5.67.

Table 1 shows the different features of every watch. This is the foundation used to decide which watch we move forward with.

Table 1: Comparing Garmin Fenix 5, Fitbit Ionic, and the Apple Watch

	Garmin Fenix 5	Fitbit Ionic	Apple Watch
Heart rate sensor	Yes	Yes	Yes
Tomsguide heart rate test, overall variance [27]	1	5.67	0.67
Accelerometer	Yes	Yes	Yes
Gyroscope	Yes	Yes	Yes
Oximeter	Yes[3]	Yes[4]	Yes[5]
Storage	16GB	2.5GB[6]	16GB
Weight	85 grams	50 grams	68 grams
Battery life	14 days[7]	5 days[8]	18 hours
Development Language	Java	JavaScript	Swift
Phone development language	Java/Swift	Java/Swift	Swift[9]
Price	$499.99	$269.95	$399
Shipment Volume (Q42018)[10]	2.5M	5.5M	10.4M
Year-over-year growth (Q42017->Q42018)[11]	N/A	3.0%	21.5%
Market Share (wearable category) (Q42018)[12]	6.5%	9.4%	27.4%

By all accounts, the watches are similar when it comes to sensors. The accuracy of the sensors is important, [27] and [26] has found that the Apple Watch is more accurate than the two other watches. When it comes to battery life, the Garmin Fenix 5 features the longest battery life, with the Apple Watch having the shortest battery life. When it comes to the development of the watch application, Apple Watch development uses the same development language and is strongly connected to the iPhone. This allows synergies if an application for the phone is needed as well. The two other watches use a combination of languages, and the Fitbit *has* to be written in two different languages. With the Garmin watch, one can develop on the Android platform and thus only having to

[3]Sampling rate unknown.
[4]No public API is available at this point in time.
[5]Apple Watch is capable of measuring blood oxygen, but the sensor is not activated.
[6]No official numbers have been released from Fitbit.
[7]Smartwatch mode with activity tracking and 24/7 wrist-based heart rate monitoring.
[8]Battery life lasts up to five days but when GPS is active the battery life is up to 10 hours.
[9]Application will only be available for iOS devices.
[10]Analysis performed by IDC. Garmin numbers are from Q417.
[11]Analysis performed by IDC.
[12]Analysis performed by IDC. Garmin numbers are from Q417.

stick to one language, which is the same case as the Apple Watch. The Garmin watch performs poorly both when it comes to price and prevalence, compared to the other watches. Although the Apple Watch is more expensive than the Fitbit Ionic, it performs better with respect to sales and has a larger market share.

Based on the findings the Apple Watch is chosen as the watch to use for this Master Thesis. The reason for this choice is the accuracy and prevalence of the Apple Watch as well as the tight integration between the iPhone and the Apple Watch which will make for a simpler application stack.

2.3 User interaction on the Apple Watch

Applications on the Apple Watch behave similarly to phone applications, however, on a much smaller screen. On the hardware side, the Apple Watch has a touch screen used for tapping the correct UI elements. As seen in Figure 4, the Apple Watch has the *Digital Crown* on the top right side. It works as a home button when pressed and acts as a scroll wheel whenever scrolling lists on the watch. In addition, the button can be used to activate Siri, by *press-and-hold*-ing. The *side button* is used to show the *Dock* when performing a single press. This button can also activate Apple Pay, by double-clicking, and turn off the watch by using a *long press* and swiping on the screen to confirm.

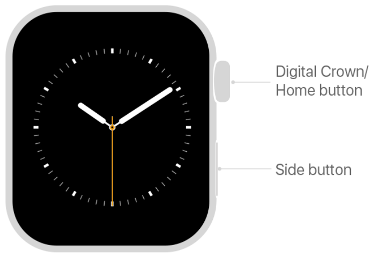


Figure 4: Illustration of the Apple Watch [28].

On the software side, there are buttons, labels, tables, and other UI elements such as on the phone. In addition to the application itself, an app can provide complications and notifications. A complication is a small UI-element that presents some item of information on the watch face. As exemplified in Figure 5, there is a complication showing the current weather at the bottom left of the watch face.

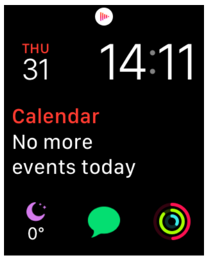


Figure 5: Screenshot of a watch face.

Part III

watchOS

3 watchOS

This section mainly discusses the watchOS platform. We start by discussing the fundamental architecture of watchOS which is inherited from iOS. Then, we move to themes like watchOS and iOS coupling, energy consumption, accessing user information, the Swift programming language, Apple Watch sensors, and lastly, some words about distributing applications on both the iOS and watchOS platforms.

3.1 Application architecture

Every iOS and watchOS application has to interact with some or all of the different layers of the application architecture hierarchy. On a high level, there are four layers; Core OS, Core Services, Media, Cocoa Touch [29], these are shown in Figure 6. These layers communicate with each other to prohibit direct communication with the hardware. The lower layers concern basic operating system services, while the higher layers provide access to the user interface, and other high-level APIs. All these layers work together to create a unified experience for the user as well as the developer. Every layer has a large suite of frameworks that provides different APIs to control most aspects of the watch and phone. The following sections present these layers.

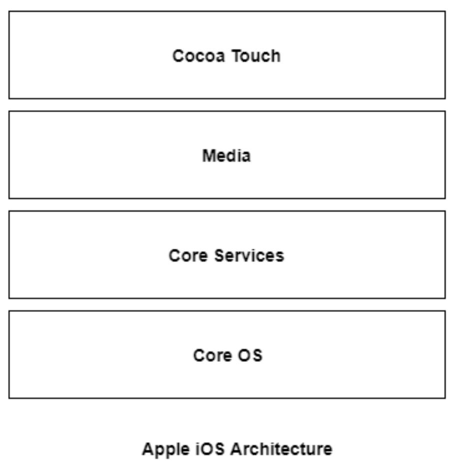


Figure 6: iOS and watchOS application architecture [30]

3.1.1 Core OS

The Core OS layer operates the operating system and a variety of low-level services such as the file system, sockets, and security-related services such as the Keychain and the Certificate framework. This layer is inherited from macOS and acts as a foundation providing access to the underlying hardware [31].

3.1.2 Core Services

This is an object-oriented layer that sits on top of and provides an abstraction to the Core OS layer. Core Services allows you to manage and access key operating system functionalities such as location, network, and threading services. It also acts as a foundation for the layers above it [31].

3.1.3 Media

The media layer provides access to different media APIs, everything from audio to images and video is provided and manipulated through this layer. In this layer, you will find graphical frameworks such as Core Graphics and Core Animation which allow the creation of graphical interfaces, images, and animations. On the audio-side frameworks like the Core Audio and the Media Player framework allows to create, record, and manipulate audio [31].

3.1.4 Cocoa Touch

Cocoa Touch is the user interface layer and framework used to create the graphical user interfaces in the applications. The layer provides an abstraction layer and gives access to a set of control elements for the graphical interface. The layer follows the Model-View-Controller (MVC), software architecture pattern. *Cocoa Touch* is the touch variant of *Cocoa* which is used on macOS [32].

3.2 Application lifecycle

Every application created on both iOS and watchOS follows the same application lifecycle pattern states. There are five distinct states an application can have; (1) Not running, (2) Inactive, (3) Active, (4) Background, and (5) Suspended.

Figure 7 shows how the five possible application states are related. The application can have the state *not running* which is the simplest state where no application processes are running. The *inactive* state is where an application is not receiving events, however, it is allowed to execute code. When the application transitions to the *active* state, execution of program code is being performed and the application is receiving events. In the *background* state the application is allowed to execute code and perform requests, however, the application is not in the foreground, meaning the user is not able to see the application or the application is not in the user's viewport on the phone. In the *suspended* state, the application is in memory, but not executing. This means that it can be terminated by the system without notice [33].

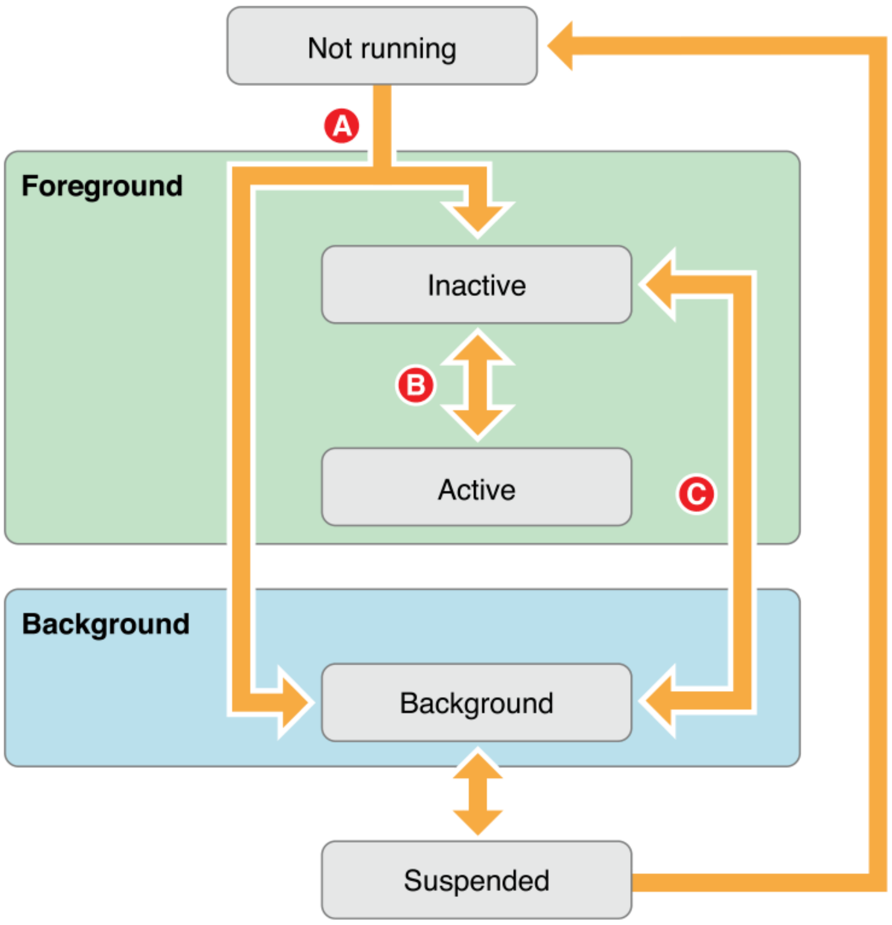


Figure 7: iOS and watchOS application states. [34]

Both iOS and watchOS use delegation heavily through delegation objects. These objects perform actions on behalf of another. This allows an object to just relay a message to another object which returns the required result. An example from iOS would be to delegate interactions within a class that extends the `UITableView` to a class that extends the `UITableViewDelegate`, by doing this, you may create two different table views which use the same interaction mechanisms and/or connects to the same underlying data. Whenever the application state is changed, the system calls the respective delegate methods. This allows a developer to save state or performs certain actions on state changes [35].

3.3 iOS and watchOS coupling

In all watchOS versions up to 6.0, all watch applications created *had* to be connected to an iOS application. This changed in watchOS version 6.0. From this version, developers are allowed to create stand-alone watch applications [36]. The previous versions made use of the `WatchConnectivity` framework which handled two-way communication between the iPhone and Apple Watch. In the first few watchOS versions, most of the application rendering happened on the phone and was sent to the watch to be displayed, this resulted in poor application performance. Applications created with the WatchKit framework can work on its own completely without any input from the phone. This allows for better performance and reduces the somewhat complex application logic found in the `WatchConnectivity` framework. When creating a completely independent watch application, there's no way to directly communicate with the phone. Therefore, the best solution is to use some sort of cloud storage, such as CloudKit to share data between the watch and phone application.

Communication using the `WatchConnectivity` has been seen by developers to be unreliable and unstable. The individual watch applications are not given control over when data transfer happens; sometimes it is immediate, and other times it may take a while for the application to be allowed to send data to the phone. This issue has been discussed online on multiple forums such as on [37] and [38]. Because of this, allowing stand-alone applications is a welcomed feature.

3.4 Energy consumption

Both platforms take measures to decrease energy consumption whenever possible. Apple touts 18 hours of battery life on their latest Apple Watch [39]. This is enough to get through a day without depleting the battery 100%. By utilizing different methods they can achieve the stated battery life. Apple's chips are created specifically for maximizing power savings. One technique they use is to batch CPU operations. This way the fixed overhead that is common for each action (e.g., waking up the device) only needs to happen once while handling many operations, thus effectively batching actions before turning idle again [40]. Apple provides complete documentation on how to create power-efficient applications, presented in [41].

Networking is one of the biggest drains on both the phone and the watch battery [40]. Therefore Apple's guideline is to minimize network requests. Again, batching is useful here as well. By batching network requests, the radios on the watch or phone do not have to power up and down all the time. Activating the radio and then batching network requests minimizes the fixed overhead cost.

Specifically, on the watch side, Apple recommends offloading complex and/or lengthy operations to the phone or a back-end server in order to reduce power consumption on the watch [40]. In addition, a watch application may not have enough time to finish the operations before the application is suspended. Taking this into account, there's are limitations as to what kind of and how often operations can be done on the watch.

3.5 Requesting access

A user expects to have control of their personal information and sensor data like the microphone and camera, therefore an application that wants access to this data and control these sensors needs to ask for the user's permission [42]. This is done through system APIs, and the platform itself has control over which apps have been granted access and which have not.

Apple provides some guidelines as to how access should be requested, with respect to the user experience. An app should only ask for personal data/sensor data when it needs it [42]. There is no reason for an RSS-reader to have access to the microphone, therefore the application should not ask for that access. On the other hand, an application that does VoIP needs access to the microphone, in that case asking for permission should be no problem. When asking for a user's permission a `purpose string` is needed, which should describe what the purpose and need of the access is. The VoIP application may want to use "*The app records your voice during phone calls*" as its purpose string.

Applications requesting access to health data need to specifically ask for the specific items the application wants to read or write. On both iOS and watchOS HealthKit is the framework where this type of data can be accessed. HealthKit acts as a central repository for generated health and fitness data, and an application can perform read and write operations once the application has been given sufficient access to the HealthKit database. In order to access HealthKit data, the application has to ask for explicit permission from the user. In HealthKit this is done through the `HKHealthStore().requestAuthorization()`-method. In order to ask for the user's permission, the `Privacy - Health Update Usage Description` and the `Privacy - Health Share Usage Description` purpose string needs to be present in the `Info.plist` file.[13] Being able to collect all this data and storing it in a central place allows multiple applications to work together and help to give the user a better experience. For example, a sleep tracking application can look at a user's workout data and deduce that the days a user performs a workout, he or she usually sleeps better. Listing 1 shows an example of code that asks the user for permission to access the HealthKit database. The `dataTypes` variable is a set containing `HKObjectType`-objects which the application asks for access to read/write in the `requestAuthorization()` call[14].

[13]Further documentation for the Info.Plist file can be found here: https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/AboutInformationPropertyListFiles.html

[14]Further documentation for the `requestAuthorization` method can be found here: https://developer.apple.com/documentation/healthkit/hkhealthstore/1614152-requestauthorization[15].

Listing 1 Asking the user for access to the HealthKit database

```
let healthStore = HKHealthStore()
let dataTypes = Set(arrayLiteral: quantityType)
healthStore.requestAuthorization(toShare: nil, read: dataTypes)
{ (success, error) -> Void in
    if (success){
        print("Authorized to use HealthKit.")
    }else{
        print("Not authorized to use HealthKit.")
        print("Error: " + error?.localizedDescription)
    }
}
```

An application is not required to write to all HealthKit data types, in fact doing so is not recommended. Apple touts that collaboration is key when working with HealthKit data, thus a developer is encouraged only to focus on a subset of tasks that he or she knows about. This also allows the user to choose the right app for him- or herself.

3.6 The Swift programming language

Swift is a general-purpose programming language created by Apple. It first launched in Q2 of 2014 as an open-source project, and has been in active development ever since. The language is set to replace Apple's Objective-C language sometime in the future, however, as of now, they live side-by-side. The language is strongly typed, allowing for type safety during development. Swift allows for *optionals* and class extensions. These and other language features are further described in Section 3.6.1. The language's compiler is highly optimized enhancing and speeding up the performance as well as ensuring static type safety optimizing development speed [43].

3.6.1 Language features of Swift

There are several language-specific features in the Swift language. These are described in the following section. We focus on features that are not widely present in other popular programming languages.

3.6.1.1 Optionals in Swift

Swift optionals allow variables to either have a value or be null, i.e., a variable is optional. This implies that optional types may experience null-pointer exceptions, whereas non-optional types may not. This language feature allows code to be performed depending on whether or not a variable is present using `guard`, as shown in Listing 2. The `guard` keyword allows an application to stop executing if a value is not present and it will proceed if the value is present. In Listing 2, the `hr` variable is seen as unwrapped and non-optional after the guard, which allows the developer to be certain that no null-pointer exception will be experienced on that variable after the `guard` statement.

Listing 2 Checking for optional values in Swift

```
let heartRate = Optional<Int>
// some other code here
guard let hr = heartRate else
{
    // handle the error case here.
    // heartRate is null
    // must exit scope of current method
}
// continue, knowing that hr is not nil
```

3.6.1.2 Extending classes

Extending Swift classes is a nifty feature that allows a developer to extend classes not written by themselves and without having direct access to the class' source code. This is known as retroactive modeling [44]. With extensions it is possible to define new instance methods, type methods, and computed properties. However, an extension may not override existing functionality. If functionality has to be overridden, then sub-classing the specific class is the way to go.

Listing 3 Extension syntax in Swift

```
extension SomeClass {
    // add methods or properties here.
    // they will be available for all
    // instances of the SomeClass class.
}
```

The extension syntax is shown in Listing 3, once inside the brackets of `SomeClass` you can write functions and properties just as you would in any other class. In Listing 4 we extend the `Double` class with a method called `rounded`. When creating a `Double` object in a Swift project, this method is just as much a method of the `Double` class as any other method of that class.

Listing 4 Extending the `Double` class

```
extension Double {
    /// Rounds the double to decimal places value
    func rounded(toPlaces places:Int) -> Double {
        let divisor = pow(10.0, Double(places))
        return (self * divisor).rounded() / divisor
    }
}
```

3.6.1.3 Automatic Reference Counting

Memory management in Swift can be thought of as "it just works", however, in the background the Automatic Reference Counting, ARC, system runs. It is somewhat similar to Java's garbage collector feature. ARC means that the developer does not need to think about managing the application's memory. The ARC frees up memory used by classes whose instances are no longer reachable in the code. The ARC only works on reference types such as instances of classes. ARC does not work on structs and enumerations, as those are value types. If the ARC malfunctions and deallocates an object that the application still needs and at some point references, it will crash. To ensure that this does not happen, the ARC system tracks the number of properties, constants, and variables that reference the given instance. Once no element references the instance it can safely be deallocated.

3.6.1.4 Type inference

Type inference is a feature of Swift. Although present in many modern languages it is a noteworthy feature and worth mentioning. In old-fashion languages, a developer explicitly has to give the data type for the variable in question, with type inference this is not necessary as the compiler itself can deduce which data type the variable is. Type inference is exemplified in Listing 5.

Listing 5 Type inference in Swift

```
let n:Int = 10 // explicitly typed
let m = 5 // compiler deduces that this is an Int
let q = 3 // same as m
let p = m + q // compiler knows p has to be Int
```

3.7 Application design with SwiftUI

Launched in 2019, SwiftUI is a brand new way to develop user interfaces across all of Apple's platforms. SwiftUI allows a developer to build views for watchOS, iOS, iPadOS, and macOS in one single file. It uses a declarative syntax, meaning the developer tells the system what the user interface should do, and the system will make sure that it happens. It is different from imperative syntax where statements are used to change the state of the user interface. The imperative syntax takes commands and performs them whereas a declarative syntax takes some logic describing the user interface and behind the scenes makes sure that the view is created [45]. Previous design frameworks such as UIKit, AppKit, and WatchKit used the imperative syntax. Notice that all the platforms use different frameworks, meaning if you knew something about UIKit used on iOS that did not directly apply to AppKit which is used on the Mac, this is not the case with SwiftUI. With SwiftUI you can learn one design framework and be proficient and create user interfaces for all Apple platforms. While there are optimizations possible for the different platforms, it is a game-changer for developers of apps on the Apple platforms and allows quicker design and development cycles across platforms. The code in Listing 6 describes a list-view with rows containing an image, some text and potentially a yellow star indicating the item is a favorite. Figure 8 shows the resulting user-interface of the code snippet. Writing the same view using UIKit would require a lot more work and method delegates.

Listing 6 List view example. Source code from [46]

```
import SwiftUI

struct ListView: View {
    var body: some View {
        List(landmarks) { landmark in
            HStack {
                Image(landmark.thumbnail)
                Text(landmark.name)
                Spacer()

                if landmark.isFavorite {
                    Image(systemName: "star.fill")
                        .foregroundColor(.yellow)
                }
            }
        }
    }
}
```

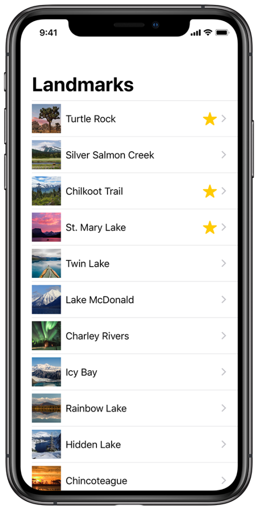


Figure 8: List view using SwiftUI

3.8 Sensors on iOS and watchOS

For the thesis, it is useful to have some knowledge about the sensors that are available on both the iPhone and the Apple Watch. The available sensors on each device are shown in Table 2. Following the sensor-table, is a description of the sensors specifically available on the Apple Watch.

Table 2: Sensors on the iPhone and the Apple Watch

	iPhone	Apple Watch
Accelerometer	Yes	Yes[16]
Gyroscope	Yes	Yes
Microphone	Yes	Yes
Camera	Yes	No
Optical heart sensor[17]	No	Yes
Electrocardiogram sensor	No	Yes

3.8.1 Accelerometer

An accelerometer is a sensor that measures the acceleration of an object, or its change in velocity [47]. On the iPhone and Apple Watch, the accelerometer sensor measures the acceleration the user imposes on the device. It can track velocity in three dimensions, often referred to as *x*, *y*, and *z*. On the iPhone, it can also be used for games, such as steering a car or plane. On the Apple Watch, this sensor is mainly used for health and fitness-related activities. [48] showed that it is possible to track "Jumping Jacks" using the Apple Watch and the watch's accelerometer.

3.8.2 Gyroscope

A gyroscope is used to measure a device's orientation and angular velocity. Gyroscopes are used in airplanes, cars, and small electronic consumer devices. The sensor itself consists of a disc that can spin rapidly since it is able to alter its direction freely. A gyroscope is used to indicate which way is "down" [49]. Both the iPhone and the Apple Watch has this sensor built-in. For the Apple Watch, the gyroscope helps in knowing when the user turns their wrists to look at the device, making sure the device wakes up. The gyroscope sensor, together with the accelerometer sensor in the watch, is able to detect if a user falls. If the watch detects a fall and that the user is not moving for two minutes, it will notify the user's emergency contacts [50].

[16]Up to 32 g-forces

[17]Allows heart rate to be measured.

3.8.3 Microphone

A microphone is a device that converts sound waves into electrical signals [51]. This device is used in many different situations, everything from capturing sound during a phone call to capturing sound from a live concert. watchOS 6 introduced a feature that measures ambient sound around the user, sending a notification to the user if the decibel level is at a level that can damage the hearing of the user [52]. Both the iPhone and the Apple Watch have a microphone sensor available, however, the iPhone's microphone is more accurate according to [53].

3.8.4 Optical heart sensor

An optical heart sensor is capable to measure a user's heart-rate. It works by shining a light through the skin and measure how the lighting scatters off the user's blood vessels [54]. Some devices using this technology are also able to measure blood oxygen saturation, *SpO2*. On the Apple Watch, the optical heart sensor is capable to measure heart-rates within the range of 30-210 beats per minute. The watch can also notify users if it detects irregular heart rhythms, so a user can contact a medical doctor to investigate the irregularity further. A teardown of the sensor has revealed that the watch is also capable of detecting oxygen blood levels, however, this feature is not activated as of watchOS 6. It may be activated in the future as discussed in [55].

3.8.5 Electrocardiogram sensor

An electrocardiogram (ECG), sensor monitors and measures the user's heart rhythm and tries to detect any problems in the rhythm. It can be used to understand the physical health of the user under monitoring. Normally, this sensor is only found in expensive medical equipment, however, since the Apple Watch Series 4, the watch has been equipped with this sensor. The sensor is cleared by the Food and Drug Administration, FDA, meaning it is classified as a "Class 2 Medical Device" [50]. With the ECG sensor in the Apple Watch, the user can take an electrocardiogram in just 30 seconds. After 30 seconds the watch will notify the user with a rhythm classification, essentially telling the user either "you are fine" or "you need to see a medical doctor".

3.9 Distributing applications

Applications cannot be freely distributed on an iPhone or Apple Watch. All applications that a developer wants to share with other users has to go through a review process performed by Apple. This also applies if the developer wants to distribute the application to testers. Applications in development can be distributed to users through Testflight, and applications ready for production have to be distributed through the Apple App Store.

3.9.1 Testflight

During the development process of an application, there may come a point where the developer needs to give access to testers who can test the application and report back to the developer if there are any major issues or give feedback. Distributing an application to testers is done through Testflight, a service provided by Apple that allows users to download new builds of an application. Testflight allows up to 10,000 external testers to test the application. Inviting testers to try the application is as simple as sharing a link with them.

3.9.2 App Store

When the application is ready for production, the application can be distributed through Apple's App Store. This is the only way to distribute production applications on iOS, iPadOS, tvOS, and watchOS. For an application to be allowed on the App Store, it has to go through a vetting process performed by Apple. In the early days of the App Store, the review time was around seven days. Today, Apple states that 50% of applications are reviewed within 24 hours and 90% of applications are reviewed in 48 hours [56].

Part IV

Design and Implementation

4 Design

The design section is introduced with a set of requirements that the application should satisfy. Section 4.2 discusses different application parts which are described using the separation of concerns design paradigm. Section 4.3 introduces and discusses different potential designs of the application. Section 4.4 presents different data format options that can be used in the application.

4.1 Requirements

In the following part, the requirements for the application are outlined and described. These requirements should be met by the application to provide a good user experience as well as creating the foundation for future development.

4.1.1 Extensibility

The application should enable future growth and development. Extending the use-case of the application by adding new sensors for data collections or sinks for data-processing should be simple for a developer. Low coupling and high cohesion is a requirement when making the application since this reduces the code complexity. As discussed in [57] an application is extensible if new features can be added without having global knowledge of the system. The application and its documentation should also be made freely available which would allow for open source contributions.

4.1.2 Resource efficiency

When creating an application for a wrist-worn device, resource usage, and resource efficiency is of high importance. One issue is the limited battery on these devices as well as the limited CPU and memory performance. Although the performance keeps improving with every iteration of the Apple Watch, it is important to keep resource efficiency in mind when developing applications.

Battery

It is important to not deplete the battery 100% during a tracking session. Often a user would like to use their watch during the day as well as the night, therefore the application should reduce energy consumption as much as possible. The requirement is that the user should be able to charge the watch for about an hour at the end of the day, sleep with their watch on and then in the morning, recharge the watch before putting it back on and have a battery that lasts the whole day.

It takes about 1,5 hours to charge an Apple Watch up to 80% from a fully depleted battery [58]. While not 100% correct as batteries charges slower when it fills up [59], this gives a recharging rate of 0,89% per minute. The scenario we need to account for is that a user should have enough battery when going to work, allowing the watch to last the whole day. Having around 80% battery when the user is ready to go to work would be suitable. A person usually needs anywhere from 30 to 90 minutes to get ready in the morning [60]. Let's take the median, 60 minutes, in order to get to 80% the battery should be at 26,6% $(80 - 60 * 0,89)$ when waking up, this allows the Apple Watch to deplete 73.4% during the night, if the user goes to bed with 100% battery. However, it is unreasonable to assume that the user would be able to charge the watch to 100% every night. Therefore, a requirement should be to not let the battery deplete more than 50%. This would allow the user to go to bed with around 80% and wake up with somewhere around 30% battery left on the device.

Networking

When it comes to networking there are two main requirements the application should handle.

1. The application should not use excessive bandwidth. If the application is supposed to send data to some online location, it is a requirement to reduce the amount of data transmitted over the network.
2. The application should allow for delay-tolerant transport, meaning if no network connection is available the data should be tried again at a later date and not be discarded.

CPU and memory usage

The application should not use excessive memory, and is should minimize CPU usage. In addition, it is important for the application to not have any memory leaks as it will run for multiple hours at a time, doing pretty much the same hundreds of times, therefore even a small memory leak can result in crashes [61].

4.1.3 Data collection

The application needs to be able to collect data from sensors on the watch, which will allow the processing of data. Without being able to collect data no value can be extracted from the application. In addition, it is essential for the application to provide useful data and insights to all stakeholders of the application.

4.1.4 Privacy

The application will handle sensitive health and biometric data for users. Therefore, it is a requirement that the data collected is stored and processed in a safe manner. The data should be handled in accordance with the General Data Protection Regulation (GDPR) [62], thus allowing individuals control over their personal data among other things.

4.1.5 Storage

The Apple Watch has limited storage. It comes with 16GB of storage, but much of this space is taken up by the operating system, apps, and artifacts (music, images, etc.). Therefore, the application should limit the local storage needs on the watch. Temporary storage is feasible, but will not work in the long run due to the limited capacity, therefore offloading should be part of the solution.

4.1.6 Stakeholders

A stakeholder is an individual or a business that has a specified interest that can affect or be affected by the application [63]. During the design phase there has been identified three stakeholders of the application.

1) End-users

With the application, we are targeting a variety of different users with respect to age and technical understanding. Therefore, the application needs to be easy to use and should be usable for everyone. The end-users will be the ones who will request new functionality for the watch application.

2) Researchers and Medical Doctors

These are not direct users of the application, but rather users of the collected data. They process data collected by the application for the user, they are likely the ones requesting new sensor wrappers and sinks.

3) Developers

The developer who builds upon the application should be able to simply understand the data flow of the application. A developer should be able to write new *sensor wrappers* or create new *sinks* as needed without having deep knowledge of the inner workings of the application. Therefore, it is a requirement that the different parts of the application have strict and understandable contractual obligations towards each other.

Recap

We have discussed the requirements the application should meet or take into consideration, to recap they are:

1. Allow for extensibility
2. Use resources efficiently
3. Collect data from sensors
4. Handle user data with care
5. Reduce local storage needs
6. Be usable for different stakeholders

4.2 Separation of concerns

Separation of concerns is a design paradigm that is used to separate parts of a program into distinct parts, allowing for simple and more understandable projects [64]. By using this principle, complexity is reduced and the application becomes more modular. Modular programming is used throughout the application to reduce coupling and allowing for testable units of code. Each module is given an interface it abides to, this makes it easy to extend and build upon. This section discusses and proposes different separations and connects each concern with applicable requirements outlined in Section 4.1.

4.2.1 Session Recording

Whenever the user is wearing the watch to do recordings, there are a lot of things happening in the application. First and foremost, every sensor is sampling events and posting them to some array. Figure 9 shows the lifecycle of a session recording. The session recording concern is a vital part in order to satisfy the data collection requirement as presented in Section 4.1.3.

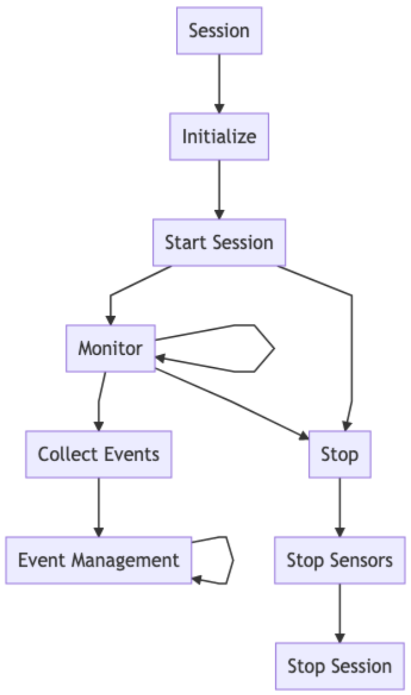


Figure 9: Session Recording lifecycle

Initialize: Sets up all connected sensors with the correct configuration.

Start Session: Starts the session, creating the first meta-event and tell all connected sensors to start collecting data.

Monitor: A timer that fires on some configurable frequency to make sure that the events can be handled. This is an ongoing task.

Collect Events: Every sensor initialized collects its events and sends them to the `Session.events` array.

Event Management: A function on a timer that handles the events. In the configuration, the programmer can add sinks that can parse, process, and handle the events.

Stop: When the user presses the `end-recording` button, the application proceeds to tear down the session.

Stop Sensors: The sensors are called and asked to stop collecting events.

Stop Session: Once the sensors has been stopped the session itself can stop. The application returns to a stale state where the user can start a new session.

4.2.2 Sensors

A sensor's lifecycle depends on the `Session`, whose responsibility is to tell the sensor to start and to stop as well as where to send the events to. The data collection requirement, presented in Section 4.1.3, is satisfied by the sensors. The sensors are the objects that collect data-points and emits them to the correct location. Figure 10 shows the lifecycle of a single sensor. There can be multiple sensors running in parallel.

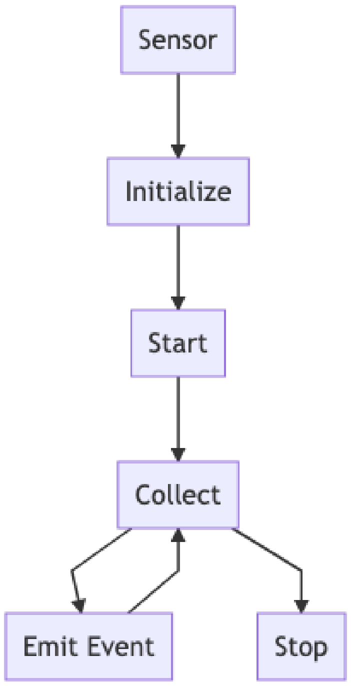


Figure 10: Sensor lifecycle

Initialize: Different sensors have different configurations, therefore all sensors take some given number of arguments during its initialization. During initialization the required variables are instantiated.

Start: When starting a sensor the sensor's collector function starts running, collecting events at a given interval, or when pushed from the hardware.

Collect: The collect function is responsible for collecting the event from the hardware.

Emit Event: The sensor needs to emit the event as a JSON data object to the `Session.events` array.

Stop: The sensor stops collecting events.

4.2.3 Event Management and Storage

While it is possible to store the data directly on the watch, this is not feasible in the long run since the watch has limited storage, as discussed in Section 4.1.5. Therefore, the data collected by the application should be offloaded to some storage location. One possibility is to offload data to the user's iCloud storage, or some other third-party storage option. Using Apple's CloudKit framework makes the development lifecycle simpler, and reduces third-party API dependencies.

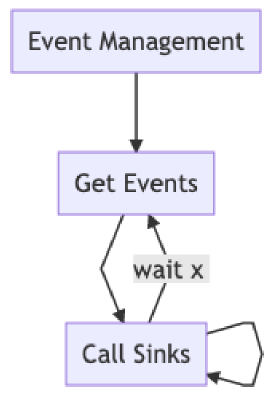


Figure 11: Event management

Get Events: Get the events as strings from the `Session.events` array.

Call Sinks: Sinks can be configured to be called every time new events are retrieved from the event array. When called the application waits for `x` seconds before it calls the `Get Events` function again.

4.2.4 Sink

When using the sink-pattern, all events are dropped into the given sink when retrieved. This allows every sink to handle the events on their own terms. A sink can, for example, send the data to `stdout` and another can take the same data and send to the cloud, or a sink can parse and aggregate data before sending to another sink which can do analysis on the data. This pattern allows for flexibility and makes the code more extensible as a developer can create a sink and just configure the `SessionController` to call the respective sink when retrieving events.

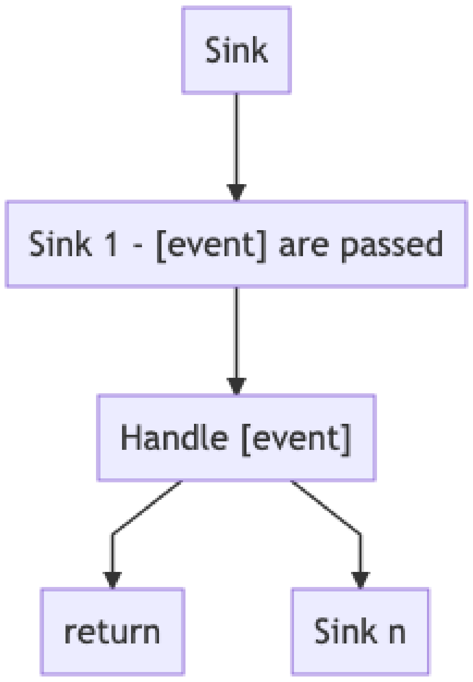


Figure 12: Sink management

Sink 1: Events are passed to the sink. The `event` array is processed and handled in isolation in this sink.

Handle [event]: Does some operation on the `event` array. Examples include uploading to the cloud, remove uninteresting events, or perform an aggregation operation on the events.

Return: Returns to the caller, no further action is made by the sink.

Sink n: A sink can call another sink. This allows a sink to do `x` on the input, `z`, transforming them to `z'` and passing `z'` into `sink n`.

When creating Sinks, it is important to understand where the data is sent. Developers who extend this part of the application need to pay close respect to the privacy requirement as outlined in Section 4.1.4. The developers have access to biometrical data from the users which are classified as sensitive, therefore handling the events should be done carefully. This means that creating sinks whose task is to send data over the Internet should be created in a secure fashion.

4.2.5 User interface

It is a requirement that the user interface should be easy and simple to understand in addition to be useable for different types of users. In Figure 13, the state machine of the user interface is presented. It is designed in a way such that no user should have any difficulties maneuvering the user interface.

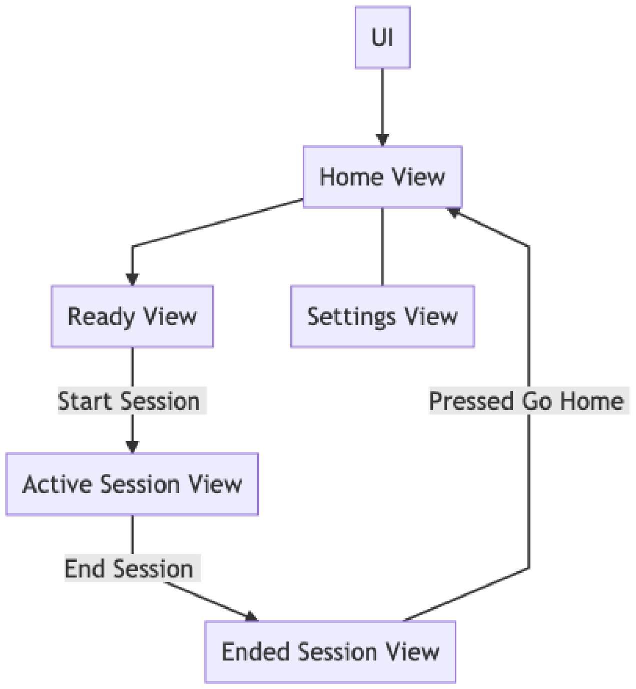


Figure 13: User Interface management

Home View: This is the initial view of the application.

Settings View: From the *Home View* the user has the option to go to the *Settings View*. Here the user is able to set different configurations and control the behavior of the application. The view is modal and can be dismissed in order to return to the *Home View*.

Ready View: This view tells the user that the watch is ready. It waits for the user to press the *Start Session* button.

Active Session View: This is the view that will be active most of the time. It displays data relevant to the current session.

Ended Session View: If the user decides to end the session by pressing the *End Session* button, he or she will end up in this view. Some statistics regarding the session are shown. The user can press the *Go Home* button to get back to the initial view.

4.2.6 Summary of concerns

Table 3 presents a summary of the concerns discussed in this chapter. The concerns are implemented in Section 5.2.

Table 3: Table of Concerns

Concern	Description
Session Recording	Controlling and administrating the sensors and other data objects needed to record a sleep session.
Sensors	Collecting, retrieving, and pushing data-points.
Event Management and Storage	Handling the events and making sure they are stored.
Sink	Processing and sending data to correct places.
User Interface	Creating a simple and intuitive user interface that can be used by users with different skill levels

4.3 High level design

The design of the application went through a few iterations, the initial design was a proof of concept, the second design improved upon this and the final architectural design was a complete rethinking of how the application should handle the events. In this part, all design iterations are presented discussing reasons for the design and why it may not be perfect and what was learned from the implementation of the designs.

4.3.1 The initial design

The initial design, as presented in Figure 14, is a bare-bones watch application whose focus is to save the session to local storage on the watch. During the first iteration, the application had to be bundled together with an iOS application, allowing for communication between the watch and the phone. Therefore, the watch application was supposed to collect sensor data, store it on the watch and later, the phone would transfer the data from the watch back to the phone. While it seemed like a good idea at the time, communication between the watch and phone introduced a lot of complexity as well as unreliability. The developer has no real control over when a file is transferred back to the phone as discussed in [37] and [38]. This introduces uncertainty as to whether or not all data-points have been transferred over to the phone, meaning real-time processing is not possible.

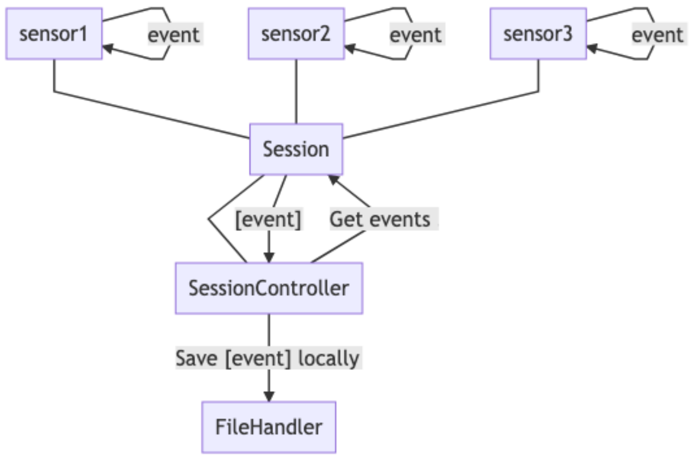


Figure 14: Initial design

In this design, every sensor-wrapper object is supposed to hold all events it collects in memory until the session is stopped, at which point the `SessionController` calls the `collectEvents` method on every sensor and writes the contents to disk on the watch. The initial design was implemented and was shown to work, but due to the connectivity issues and unreliableness of the communication between the phone and watch it was decided that the application should be implemented as a standalone application. In addition, Apple has on multiple occasions during the past few years paved the road for stand-alone watch applications, and with watchOS 6.0 they now allow users to download applications directly from the watch removing the need for the phone to install third-party applications [65]. Due to these two factors, we regard implementing the watch as an independent application the most correct and forward-thinking way to go.

4.3.2 The improved design

The improved design, shown in Figure 15, uses many ideas of the initial design, however, the implementation is done as a standalone application. Creating a standalone application introduces complexity with regard to how users would be able to access their data. The solution in this design is to use the CloudKit framework to upload data to iCloud, which allows the application to upload data to the cloud. This design also allows the phone to get access to the user's data if the same iCloud account is used.

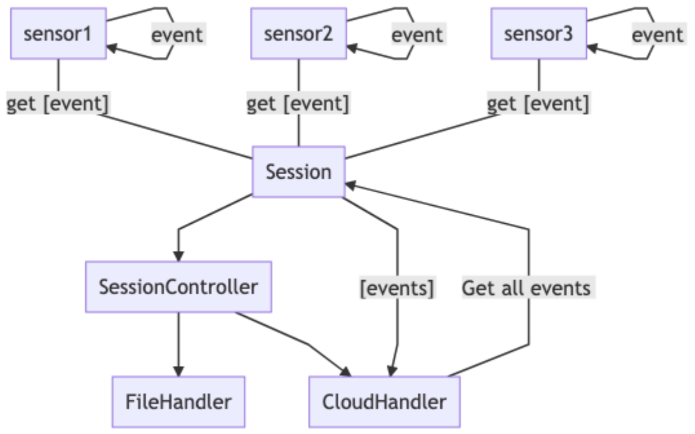


Figure 15: Improved design

As with the initial design, all sensor wrappers are required to hold their events until the session is stopped, then the events are collected by the `SessionController`. The `SessionController` then handles the events by sending them to file storage or to the cloud depending on the configuration.

This design works better than the initial design. However, when running the application for more than two hours it crashes due to some bug. After investigating, it was found that the error is caused by a memory issue which resulted in the application receiving a low-memory warning and eventually being killed by the operating system. The memory warning comes from the event collection, and the fact that the collection is done in-memory over long time periods. This makes the application basically unusable for long sleep sessions since it does not have the ability to save the collected data to disk or send to the cloud before being killed.

In retrospect, the improved design is simple enough, but the degree to which it is extensible can be discussed. As the design is described in Figure 15 it is difficult to see that it is extensible, and when implementing this design there were many hurdles when trying to add more sensors and handlers. Due to the memory issue and extensibility question, it was decided that the application design should be rethought.

4.3.3 The final design

In the final design, learnings were taken from the previous iterations. First and foremost, the sensor wrappers in this design emit events to the `Session.events` array, thus the controller has one single place to retrieve events. This makes the design between the `Session` and the `SessionController` simpler since the

SessionController has no knowledge of the sensor wrappers, e.g., how many sensors there are or their internal methods, as it communicates with the Session class.

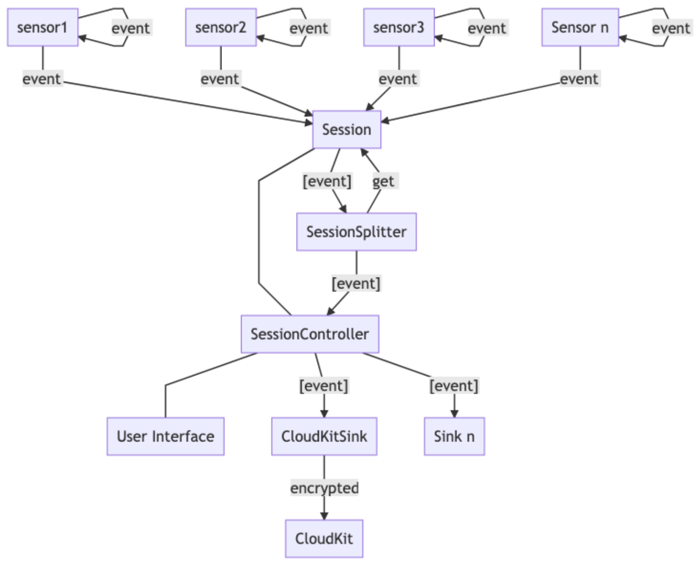


Figure 16: Final design

In order to handle the low-memory issue, the new design has a SessionSplitter class whose task is to split the Session.event-array into batches at a configurable interval. This class is triggered by a timer, and when it fires it fetches all events from the events property of the Session object and returns them to the SessionController for it to handle and process.

When designing this iteration we introduce the concept of sinks. In our design, a sink is a class designed to handle a batch of events. The SessionController should allow for multiple sinks to run in parallel. While not shown in Figure 16, a sink is also able to pass events to another sink, i.e. chaining sinks.

This design allows the application to easily be extended. If a new sensor comes along, a sensor wrapper needs to be created and it only has to adhere to a few contractual obligations towards the Session class. The Sink idea allows a developer to build features on top of the collected data such as aggregation functions, reducing functions, and storing the collected data elsewhere than local storage. With these choices, the extensibility requirements outlined in Section 4.1 should be met.

Core design elements

In the design phases, some core design elements were found, these are presented in Table 4

Table 4: Core objects

Name	Description
Event	Representing an event generated by a sensor.
Sensor	This class is a wrapper around the underlying hardware sensor on the watch. Its purpose is to generate events at a specified interval.
Session	The Session class is responsible for the current sleep session.
Session-Controller	This controller class manages the `Session` object, handling the start and stop of the session.
Sink	This object handles the batches of events sent from the `SessionController`.

4.4 The Data Format

The following section discusses some of the different data formats that are applicable for use in the application. During the creation of the application three possible data formats were considered; 1) Extensible markup language (XML), 2) Comma-separated values (CSV) and 3) Javascript Object Notation (JSON). When investigating the requirements, the focus was put on the format being extensible and storage efficient. When discussing the different options, we exemplify with an accelerometer event as shown in Listing 7.

Listing 7 Accelerometer event in Swift

```
struct AccelerometerEvent:EventProtocol{
    var sessionIdentifier: String
    var sensorName: String
    var timestamp: TimeInterval
    private var event:EventData

    private struct EventData:Codable{
        var x:Double
        var y:Double
        var z:Double
    }
}
```

XML

XML was designed in 1996, and in 1998 it officially became a W3C standard

[66]. It was developed in order to provide simplicity, generality, and easy data exchange across the Internet as well as being in a human-readable format. Each XML document consists of one or more elements surrounded by tags (< >). The format provides hierarchical data structures, thus it is can easily support complex data. Listing 8 exemplifies the accelerometer event in XML. Adding another type of sensor is as easy as changing some of the tags, and can easily be accounted for in an application.

Listing 8 Accelerometer event in XML

```xml
<sensor_event>
    <sessionIdentifier>someString</sessionIdentifier>
    <sensorName>Accelerometer</sensorName>
    <timestamp>1578476733554</timestamp>
    <event>
        <x>3.14159</x>
        <y>1.74421</y>
        <z>-2.88329</z>
    </event>
</sensor_event>
```

The versatile format has one big caveat, which is size. The data format can be up to three times as large as CSV, and twice as large as the JSON format. The data in Listing 8 is 210 bytes when using a minifier (removing whitespace and line-shifts).

CSV

While the CSV file format has been available since the 1970s, the file format was officially standardized in 2005 [67]. A CSV file uses a comma (or another specified delimiter) to separate values. Every line is regarded as one record, with one type of record this works fine, for example, a list with data from one single sensor.

Listing 9 Accelerometer event in CSV

```
sessionIdentifier,sensorName,timestamp,x,y,z
someString,Accelerometer,1578476733554,3.14159,1.74421,-2.88329
```

Listing 9 shows how an accelerometer event may look in a CSV file. The problem with this format is that it does not support data hierarchies, which is important for the application we are creating. While it is possible to create a custom parser allowing for more complex data structures, it will cause a lot of pain to maintain the parser when improving and changing the application. Therefore, this file format is not versatile enough for use in this thesis.

JSON

The JSON format was originally specified by Douglas Crockford in the early 2000s. The format is language independent, however, it takes cues from popular programming languages. The format is seen as a replacement for XML. The format shaves the end-tags (</tagname>) known from XML, saving space. In addition, the format supports complex data structures and is very versatile. An example of an accelerometer event in JSON is shown in Listing 10.

Listing 10 Accelerometer event in JSON

```
{
  "sessionIdentifier":"someString",
  "timestamp":1578476733554,
  "sensorName":"Accelerometer",
  "event":{
    "x":3.14159,
    "y":1.74421,
    "z":-2.88329
  }
}
```

The minified version of the event in JSON is 136 bytes, a reduction of 35% compared to the XML format. Other events may have even bigger reductions in size. Due to the versatility and space savings, the JSON format is used as the storage format for the data collected from the sensors.

Table 5 shows the connection between a sensor's sampling rate, number of sensors, and the amount of data when an emitted event is 136 bytes. On the lowest end, a sleep session will collect just under 4 megabytes of data, whereas on the highest side the data collected will be just under 4 gigabytes.

Table 5: Storage estimation

Sampling rate	Number of sensors connected	Data pr. Minute (mb)	Data pr. Hour (mb)	Data pr. sleep session (8h) (mb)
1	1	0,00816	0,4896	3,9168
1	3	0,02448	1,4688	11,7504
1	5	0,0408	2,448	19,584
10	1	0,0816	4,896	39,168
10	3	0,2448	14,688	117,504
10	5	0,408	24,48	195,84
25	1	0,204	12,24	97,92
25	3	0,612	36,72	293,76

Sampling rate	Number of sensors connected	Data pr. Minute (mb)	Data pr. Hour (mb)	Data pr. sleep session (8h) (mb)
25	5	1,02	61,2	489,6
50	1	0,408	24,48	195,84
50	3	1,224	73,44	587,52
50	5	2,04	122,4	979,2
100	1	0,816	48,96	391,68
100	3	2,448	146,88	1175,04
100	5	4,08	244,8	1958,4
200	1	1,632	97,92	783,36
200	3	4,896	293,76	2350,08
200	5	8,16	489,6	3916,8

4.4.1 Event Types

In general, the application creates two types of events; meta events and sensor events. The following part will explain these two event types.

Meta Event

This event is emitted at the start and the end of every session. The start-meta event, shown in Listing 11 and Listing 12, contains information about every sensor applied to the given session, as well as other metadata important for analyzing the dataset. The end-meta event, shown in Listing 13, contains aggregated information about the session, such as total events captured, running time of the session, and the total decrease in battery level during the session. Both the start and end meta event contains a session identifier in order to relate the meta event to the sensor events.

Listing 11 Start meta event 1/2

```
{
    "userID": "UUID",
    "userInfo": {
      "age": 25,
      "gender": "male",
      "weight": 85.5,
      "height": 193
    }
  ...
```

Listing 12 Start meta event 2/2

```
{
  ...
    "sessionIdentifier": "UUID",
    "timestamp": 1578483807.661,
    "sensor": "MetaSensor",
    "event": {
      "type": "Start",
      "device": "Apple Watch",
      "version": "Version string of device",
      "date": "somedate",
      "Sensors": ["list of applied sensors and their configs"]
    }
}
```

Description of Start-Meta Event

```
*UserID*: An UUID identifying the current user.

*UserInfo*: JSON object containing data about the user.

*SessionIdentifier*: A UUID as string identifying the given session.

*Timestamp*: Epoch timestamp for the event.

*Sensor*: Name of the given sensor. MetaSensor here.

*Type*: Type of event in for the given sensor.

*Device*: Which device the session is running on.

*Sensors*: A list of sensors as a JSON object. All the sensors
will have a meta-event themselves with information related to
how the sensor collects data.
```

Listing 13 End meta event

```
{
    "userID": "UUID",
    "sessionIdentifier": "UUID",
    "timestamp": 1578483807.661,
    "sensor": "MetaSensor",
    "event": {
        "type": "End",
        "eventCount": 100,
        "runningTime": 67807.142
    }
}
```

Description of Stop-Meta Event

```
*Event Count*: Number of events collected during the session.

*Total Running Time*: The total running time of the session.
```

Sensor events

Every sensor that creates an event follows the same event format. The fields `sessionIdentifier`, `sensorName`, `timestamp`, and `event` must be present in every data-point emitted from the sensor. The `event` field is its own JSON object. It is custom for each sensor, therefore specific sensor data is added in this object. The events created by the heart-rate, battery and accelerometer sensor are presented and exemplified in Listing 14, Listing 15, and Listing 16.

Listing 14 Example of a heart rate event

```
{
  "sessionIdentifier":"426AB2BF-CA06-4CD2-8882-EB471E2CBDFF",
  "timestamp":1578483807.6612411,
  "sensorName":"Heart Rate",
  "event":{
    "heartRate":62
  }
}
```

Listing 15 Example of a battery event

```
{
  "sessionIdentifier":"426AB2BF-CA06-4CD2-8882-EB471E2CBDFF",
  "timestamp":1578483809.7238933,
  "sensorName":"Battery",
  "event":{
    "batteryPercent":81,
    "batteryState":1
  }
}
```

Listing 16 Example of a accelerometer event

```
{
  "sessionIdentifier":"426AB2BF-CA06-4CD2-8882-EB471E2CBDFF",
  "timestamp":1578483810.3035579,
  "sensorName":"Accelerometer",
  "event":{
      "x":0.01434326171875,
      "y":-0.0277252197265625,
      "z":-0.9988861083984375
  }
}
```

5 Implementation

This chapter describes the implementation of an app called Sopor.[18] Based on the final design from Chapter 4, the different application components are presented as well as how the separate concerns identified are solved.

5.1 Application components

In Sopor, the different application components are given specified tasks to be in control of and perform. Some components have contractual obligations towards other components, meaning that the components' interfaces should not change. This is useful for further development since the components can be improved internally without introducing change to the outside world. In addition, it increases the stability of the software since changes to the components' APIs are not introduced without careful consideration. In this section, the implementation of the components is presented.

5.1.1 Sensor

At the heart of Sopor, we have the sensor wrappers for the sensors in the Apple Watch. All sensors are required to follow the interface described in Listing 17. Furthermore, all sensors can extend the interface to account for special cases for the given sensor.

Listing 17 Sensor interface

```
protocol SensorInterface {
    var sensorName:SensorEnumeration { get }
    func startSensor(session:Session) -> Bool
    func stopSensor() -> Bool
    func collectEvent()
    func storeEvent(data:Data)
}
```

Description of the sensor interface

sensorName: All sensors have a given sensor name that uniquely identifies them. The *sensorName* has to be defined as a `SensorEnumeration`[19].

startSensor(): Whenever the session starts, the `Session` class calls the `StartSensor` method. This function handles the initiation of the sensor. Every sensor needs a different implementation of this method. The method takes a `Session` object, which is referenced multiple times during the session. It returns a boolean value detailing if the sensor was successfully started or not.

[18]Latin for "deep sleep".

[19]The `SensorEnumeration.swift` file contains a KV-store of all sensor wrappers.

stopSensor(): This method is called by the `Session` and indicates that the sensor should stop collecting events. This function is called when the user has ended the session, or if for some reason the application is required to shut down, such as in the event in which the watch is completely drained for its battery.

collectEvent(): The sensors implemented in Sopor use different collection methods for gathering events. The heart rate sensor wrapper subscribes to the heart rate sensor in the watch and is notified for every new event, whereas the accelerometer sensor can be given a sampling rate to follow. Regardless of the data collection method, the `collectEvent` method is called to handle the new data-points.

storeEvent(): All sensors that inherit from the base `Sensor` class[20] get this function for free. This function takes an encoded JSON event and stores it in the current session's `events` array.

A sensor wrapper is not required to follow any given initializer, since different sensors may be set up with different arguments. The `SensorInterface` is an interface defining which functions and properties a sensor wrapper should have, and since every sensor often has some special implementation detail the interface leaves some work for the developer. The interface makes every sensor conform to the contractual obligation between the `Sensor` class and the `Session` class. Table 6 shows the implemented sensors in Sopor.

Table 6: The supported sensors in Sopor

Sensor	Responsibility	Configurable[21]
Accelerometer	Collects accelerometer data	Yes
Battery	Collects battery information	Yes
Heart Rate	Collects heart rate of user	No
Meta	Collects metadata about a session	No
Gyroscope	Collects gyroscope data	Yes

5.1.2 Events

Events represent data-points collected from the sensors, these are in essence simple struct objects. All `Event` structs follow the `EventProtocol`. The metadata the struct needs to implement is shown in Listing 18. These properties are necessary in order to figure out from which sensor the event comes from, at what time the event is created, and which session it belongs to. In addition, events have an internal structure for the specific event data that the specific event-struct needs to represent, this is shown in Listing 19. The internal struct describes the actual information the event collects when retrieving the data.

[20]The Sensor class is defined in `Sensor.swift`.
[21]A sensor is considered configurable if the sampling rate can be changed.

Listing 18 EventProtocol

```
protocol EventProtocol:Codable {
    var timestamp:UInt64 { get set }
    var sensorName:String { get set }
    var sessionIdentifier:String { get set }
}
```

Listing 19 Accelerometer event struct

```
struct AccelerometerEvent:EventProtocol{
    var sessionIdentifier: String
    var sensorName: String
    var timestamp: UInt64
    private var event:EventData

    private struct EventData:Codable{
        var x:Double
        var y:Double
        var z:Double
    }
}
```

`Event` objects have no methods, this is by design as the events are only data-points and should not perform any actions. Table 7 shows the `Events` that are available in Sopor.

Table 7: Events correlation to sensor-classes in Sopor

Event name	Connected To Sensor-Class
GyroscopeEvent	GyroscopeSensor
HeartRateEvent	HeartRateSensor
BatteryEvent	BatterySensorWatch
AccelerometerEvent	AccelerometerSensor
MetaEvent	MetaSensor
MetaStopEvent	MetaSensor

In addition to events related directly to `Sensor` classes, it is possible to describe events that happen outside of sensors, such as in the `AggregationSink`, described in Section 5.2.3.3, where multiple events are aggregated and reduced to one single event. One such event is defined in `AggregatedMetric.swift`, shown in Listing 20. This struct is essentially identical to the other event structs shown in Table 7, having a custom internal representation of the data, in addition to conforming to the `EventProtocol` representation.

Listing 20 AggregatedMetric struct

```
struct AggregatedMetric:EventProtocol{
    var sensorName: String
    var timestamp: UInt64
    var sessionIdentifier:String
    private var event:EventData

    private struct EventData:Codable{
        var metricValue:Double
        var type:String
    }
}
```

5.1.3 Session

The `Session` class handles a single session. The class has a list of sensors, stored in the `sensorList` variable and the `Session` object is responsible for interfacing with the sensors, i.e. calling their start and stop methods. The `Session` class also holds the list of events emitted by the sensors in the `events:[Data]` array. During the first design and proof-of-concept, the session class was responsible for pulling data from the sensors at a given interval, this was rethought in the final design and it was decided that the sensors were responsible for pushing the events to the active `Session`. The Session also have a few helper functions, such as `getLatestBatteryWatchEvent()` which returns the latest battery information and `getLatestHREvent()` which returns the latest heart rate for the user. These methods are called by the `SessionController` class and serve the function of giving feedback to the user interface thus being informational to the end-user.

5.1.4 Session Controller

The `SessionController` handles the interaction between the user interface, the `Session` class and other helper modules. Whenever a user wants to start a session the `SessionController` calls the `Session`'s `startSensors()` method, the same goes when the user wants to end a session, the session's `endSession()` method is called. The user interface classes may call the `SessionController` in order to get the latest data-points. This class is also responsible for starting and triggering the timer that will call the `SessionSplitter`'s `splitSession` function, the trigger frequency is given by the `BATCHFREQUENCY` as discussed in Section 5.1.5. The return value of the `splitSession` function is an array of `Data` objects which the `SessionController` sends to the configured `Sinks`.

5.1.5 Session Config

The `SessionConfig` class allows the developer to set up the sensors with different configurations. The ease of configurability has been a valuable feature during the development of `Sopor`. The `getSensorList()` function initializes a list of sensors and passes them back to the caller. The `getSinkMethod()`, shown in Listing 21, is a function that the `SessionController` can use when handling the batch of events, this way the developer can easily change the behavior of the sinks. It is also in the `SessionConfig` class where the frequency of the `SessionSplitter.splitSession()` function can be set through the `BATCHFREQUENCY` property. As a default, this is set to 120 seconds.

Listing 21 Get the sink configuration

```
static func getSinkMethod() -> ([Data], String) -> [Data] {
    return runSinks // Return the configured Sink-workflow
}

static private func runSinks(events:[Data], UUID:String) -> [Data]
{
    let filtered = FilterSink.runSink(
                events: events,
                sensorsToRemove: ["Battery"]
            )
    let aggregated = AggregationSink.runSink(
                events: filtered,
                sessionIdentifier: UUID
            )
    return aggregated
}
```

In the future, the `SessionConfig` class should also handle different user-set configurations. A user may want to only collect data from some sensors, or control the sampling rate for a given set of sensors where applicable. This class can be used to set and get these user-specified configurations.

5.1.6 Session Splitter

This class was implemented in response to a memory issue that caused the application to crash every two hours when using the initial and improved design of the application, as discussed in Section 4.3.2. Whenever Sopor collected data for too long, the application would essentially run out of available memory and crash. This class is the solution to that problem. The function `splitSession()` handles the logic of splitting and copying all events to a new array and removing the elements from the active `Session`'s `events` array.

Listing 22 shows the `splitSession()` method. First, it fetches the number of elements in the `Session`'s `events` array and stores this in the `eventCount` variable. Then it moves the sub-range `0->eventCount` to a new array. When it removes the subrange in the `events` array, the function essentially clears out the array. Finally, the elements removed from the `events` array are returned to the caller. This method is partially thread-safe. Other threads may append to the `events` array while this method runs, prepending and other operations are not thread-safe in the current implementation. One way to implement full thread-safety requires to introduce a lock to the `splitSession` method, this allows the thread to execute knowing no-one else can access the `events` array at the same time. One issue with this could be that that the `removeSubrange` would take so much time to complete such that the sensors would be waiting for a long time before being allowed access to the `events` array. The current implementation is sufficient enough for its use.

Listing 22 Splitting the events

```
func splitSession(session:Session) -> [Data]{
    let eventCount = session.events.count - 1
        // Index starts at 0
    let splittedEvents = Array(session.events[0...eventCount])
    session.events.removeSubrange(0...eventCount)
    return splittedEvents
}
```

5.1.7 Sinks

A `Sink` in Sopor is essentially an event consumer. It is responsible for applying some function on the events it consumes. All `Sink` classes need only to implement one function which is presented in Listing 23. In Sopor, a sink takes an array of `Data` objects, it processes them and then returns the same or possibly a mutated array to the caller. Since a `Sink` returns a `Data` array to the caller, multiple `Sinks` can be used on the same batch of events. This pattern allows, for example, a sink to aggregate some data, then pass it to a sink which filters out events based on some function, and lastly the final array can be passed to a sink whose task is to store the data, either locally or online in the cloud.

Listing 23 Sink protocol

```
protocol Sink {
    static func runSink(events:[Data]) -> [Data]
}
```

5.2 Implementation of concerns

In Section 4.2, different components are conceptualized and decomposed into separate concerns. This section discusses and presents the specific implementation details in Sopor of the concerns described in Section 4.2.

5.2.1 Sensors

In Sopor, all sensor wrappers adhere to a strict basic interface as presented in Listing 17. This interface defines methods that each sensor must follow. The different wrappers are presented in Table 6. In this section, each wrapper is presented and specific implementation details are discussed.

5.2.1.1 Sensor inheritance

All sensors in Sopor inherit from the `Sensor` class.[22] Inheritance allows some methods to only be implemented in this class while being available for all subclasses. In the current implementation, this applies only to the `storeEvent()` method. The inheritance model is presented in Figure 17.

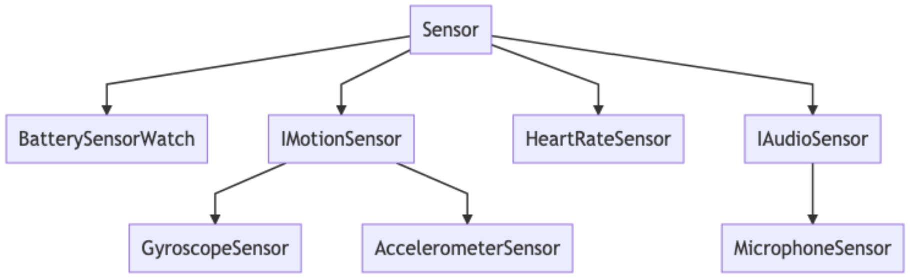


Figure 17: Sensor inheritance in Sopor

5.2.1.2 Accelerometer

When collecting Accelerometer data, the `CMMotionManager`[23] object needs to be instantiated. The `CMMotionManager` object makes sure that the data that is made available for collection has been processed to remove environmental bias, such as gravity effects. This object provides methods for starting, stopping, and managing motion services on the device. By using the `CMMotionManager`'s `isAccelerometerAvailable` method one can make sure that the accelerometer sensor is available on the given watch. Next, one has to set the `accelerometerUpdateInterval` property in order to set the sampling rate for the sensor. Finally, the sensor updates have to be started using the `startAccelerometerUpdates` method. Listing 24 shows how the accelerometer sensor is started.

[22]Defined in the Sensor.swift file.
[23]Part of the Core Motion framework.

Listing 24 Starting the accelerometer

```
func startAccelerometers() {
    // Make sure the accelerometer hardware is available.
    if self.motion.isAccelerometerAvailable {
        self.motion.accelerometerUpdateInterval = 1.0/60.0
        self.motion.startAccelerometerUpdates()
        // other setup can happen here.
    }else{
        log.information("Accelerometer is not available")
    }
}
```

When the session runs, a timer triggers at the same frequency as the sampling rate given in the `startAccelerometer()` method. The trigger activates the `collectEvent()` method that collects and stores the collected event, as shown in Listing 25.

Listing 25 Collecting accelerometer events

```
func collectEvent() {
    let event = createEvent()
    if let unwrappedEvent = event{
        let encodedEvent = self.encodeEvent(event: unwrappedEvent)
        storeEvent(data:encodedEvent)
    }else{
        fatalError("AccelerometerEvent is nil")
    }
}

func createEvent() -> AccelerometerEvent {
    if let data = self.motion.accelerometerData {
        let timestamp = NSDate()
        let x = data.acceleration.x
        let y = data.acceleration.y
        let z = data.acceleration.z

        return AccelerometerEvent(x: x,
            y: y,
            z: z,
            timestamp: timestamp,
            sessionIdentifier: self.sessionId?.description ?? "NA")
        }
    return nil // accelerometerData not available
}
```

5.2.1.3 Gyroscope

In general, this sensor behaves much like the accelerometer sensor described in Section 5.2.1.2. As with the accelerometer, the gyroscope can be accessed using the `CMMotionManager` object. By calling the CMMotionManager's `isGyroAvailable()` method, one can make sure the device has a gyroscope available. On earlier models of the Apple Watch, this sensor is not available. However, newer Apple Watches allow access to the collected gyroscope data. One can set the sampling rate using the `gyroUpdateInterval()` method, and start subscribing to updates by calling the `startGyroUpdates()` method.

5.2.1.4 Heart rate

The heart rate sensor on the Apple Watch cannot be accessed directly, as getting the raw heart rate readings is not allowed [68]. The application can, however, access the near real-time heart rate by starting an `HKWorkoutSession`, which streams data to given methods once the data becomes available. Therefore, the delay in getting the heart rate is reduced when running in this mode. Another method of retrieving the heart rate data is to query the `HealthKit` database. This has one big drawback which is that the data is not readily available, and often it is delayed by minutes. Due to this, the first option is implemented, which essentially makes the application a workout application in Apple's eyes.

Before the heart rate sensor can be started, the workout session needs to be configured and the query for the heart rate has to be created. Creating the heart rate query is shown in Listing 26. Essentially the query will query for the `HKQuantityTypeIdentifier.heartRate` sample. The heart rate sensor wrapper is defined in the `HeartRateSensor` class.[24]

[24]Defined in the HeartRateSensor.swift file.

Listing 26 Creating heart rate query

```
func createHRStreamingQuery(_ workoutStartDate: Date) -> HKQuery? {
    guard let quantityType = HKObjectType.quantityType(
        forIdentifier: HKQuantityTypeIdentifier.heartRate)
    else { return nil }

    let datePredicate = HKQuery
        .predicateForSamples(withStart: workoutStartDate,
                             end: nil, options: .strictEndDate)
    let predicate = NSCompoundPredicate(
            andPredicateWithSubpredicates:[datePredicate])

    let heartRateQuery = HKAnchoredObjectQuery(type: quantityType,
        predicate: predicate, anchor: nil,
        limit: Int(HKObjectQueryNoLimit)) {(query,
            sampleObjects, deletedObjects, newAnchor,
            error) -> Void in
                self.collectEvent(sampleObjects)
        }

    heartRateQuery.updateHandler = {(query, samples, deleteObjects,
        newAnchor, error) -> Void in self.collectEvent(samples)
    }
    return heartRateQuery
}
```

All workout sessions need to have an `HKHealthStore` object available, which in the HeartRateSensor class is bound to the `healthStore` variable. The `HKHealthStore` object is the access point for all HealthKit managed data. This object is also used when instantiating the HealthKit query created by the function shown in Listing 26.

5.2.1.5 Battery

The `BatterySensorWatch` class[25] measures the watch's battery level. This is useful in order to understand the resource usage of Sopor. The collect function for this sensor runs on a configurable timer set by the `samplingRate` variable.

To start collecting battery information from the watch, the operating system needs to be notified that the application wants to monitor the battery state by setting the `isBatteryMonitoringEnabled` property to `true`. In addition, a timer has to be instantiated in order to collect the battery data at a given interval. This is shown in Listing 27.

[25] Defined in the BatterySensorWatch.swift file.

Listing 27 Starting battery monitoring

```
override func startSensor(session:Session) -> Bool {
    currentSession = session

    // Enable battery monitoring
    WKInterfaceDevice.current().isBatteryMonitoringEnabled = true

    // Configure a timer to fetch the battery data.
    self.timer = Timer(
        fire: Date(),
        interval: (self.samplingRate),
        repeats: true, block: { (timer) in
            // Get the battery data on an interval.
            self.collectEvent()
        })

    // Add the timer to the current run loop.
    RunLoop.current.add(self.timer!, forMode: .default)

    return true
}
```

Every time the timer in the battery sensor triggers, the `collectEvent` is called. This event calls two helper functions, `createEvent` and `encodeEvent`, in order to create the `BatteryEvent` which is stored in the `Session`'s `events` array. When creating the `BatteryEvent` the `WKInterfaceDevice` class is invoked and multiple helper methods are called in order to get the data-points needed for the `BatteryEvent`. The `collectEvent` and the `createEvent` method is presented in Listing 28 and Listing 29 respectively.

Listing 28 BatterySensor: Collect event

```
func collectEvent(){
    let event = createEvent()

    // Convert the BatteryEvent object to a data object
    let encodedEvent = encodeEvent(event:event)
    storeEvent(data:jsonEncodedEvent)
}
```

Listing 29 BatterySensor: Create event

```
func createEvent() -> BatteryEvent{
    let batteryLevel = WKInterfaceDevice.current()
                                        .batteryLevel
    let batteryState = WKInterfaceDevice.current()
                                        .batteryState
                                        .rawValue
    let device = WKInterfaceDevice.current()
                                  .model

    let event = BatteryEvent(device: device,
                             batteryLevel: batteryLevel,
                             batteryState: batteryState,
                             sID: sessionID.description)
    return event
}
```

5.2.1.6 Microphone

The Apple Watch is equipped with a microphone which is accessible for developers using different APIs. One API that can be used is the `AVAudioRecorder` which allows recordings to be made. While not implemented in the final delivery, it is easy to get started with voice recording on the Apple Watch, as shown in Listing 30, however, getting a fully-fledged recording setup is more complicated. One major hurdle is whether or not to record the whole session, or just record parts of the session based on some trigger. Recording for more than a few hours can drain the battery heavily while recording only parts of a session would potentially leave out important data-points. Another way to tackle the issue would be to record using the phone which presumably is next to the user when sleeping. Since the phone is often connected to power during the night the battery issue is reduced to a trivial point. In addition, recording with the phone allows the sound to be recorded all the time regardless of the user's sleeping position. Sleeping positions are rather dependent on the individual in question as concluded in [69] and some users even sleep with their hands under the pillow, thus any sound will be muffled or not noticeable at all. Therefore, an application on the phone would be better at picking up sounds created by the user.

Listing 30 Microphone sensor pseudocode. Code from [70].

```
let dirPath = getDirectory()
let pathArray = [dirPath, "audio.m4a"]
guard let filePath = URL(string: pathArray.joined(separator: "/"))
    else { return }

let settings = [AVFormatIDKey: Int(kAudioFormatMPEG4AAC),
                AVSampleRateKey:12000,
                AVNumberOfChannelsKey:1,
                AVEncoderAudioQualityKey:
                    AVAudioQuality.high.rawValue]

//start recording
do {
    audioRecorder = try AVAudioRecorder(url: filePath,
                                        settings: settings)
    audioRecorder.delegate = self
    audioRecorder.record()
} catch {
    print("Recording Failed")
}
```

5.2.1.7 Meta sensor

At the start and the end of each session, a meta event is created by the `MetaSensor`.[26] As shown in Section 4.4.1, the meta events collect metadata about the session. When starting it captures a snapshot of the available data about the user and the connected sensors. At the end of a session, the `end-meta` event collects aggregated data such as the total event count and the running time of the session. There are no sampling rate, queries, or timers required for this sensor, it creates events when its `startSensor()` and `stopSensor()` methods are called.

5.2.2 Session Recording

When a user presses the `Start Session` button shown in Figure 18, the button's associated action calls the `SessionController`'s `startSession()` method shown in Listing 31. This method initializes a new `Session` object and calls the `currentSession`'s `startSensors()` method.

[26]Defined in the MetaSensor.swift file.

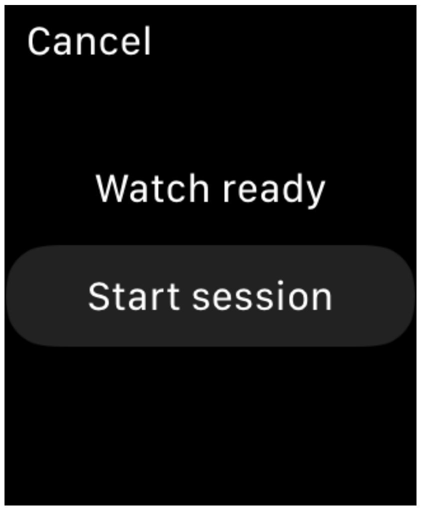


Figure 18: Sopor start session screenshot

Listing 31 Starting a Session in the SessionController class

```
func startSession(wakeUpTime:Date) -> Session{
    let SESSION_UUID = UUID()
    let sensorList = SessionConfig.getSensorList(
                            SESSION_UUID: SESSION_UUID
                    )

    currentSession = Session(wakeUpTime: wakeUpTime,
                            sensorList: sensorList,
                            sessionIdentifier: SESSION_UUID
                    )
    currentSession.startSession()
    setupBatchTimer(interval: SessionConfig.BATCHFREQUENCY)
    return currentSession
}
```

5.2.2.1 Data acquisition

There are essentially three phases of the data acquisition; 1) starting, 2) collecting, and 3) stopping. During the start phase, all sensors are started. In the collection phase, sensor wrappers make sure to collect the emitted events. During the stop phase, all sensors are turned off in order to preserve battery life and avoid memory leaks. The phases are described in detail in the following section.

Starting the data acquisition

Figure 19 shows the flow of events when starting the `Session`. The

`SessionController` calls the `Session` in order to initialize all sensors. When the `Session` is initialized and its `startSensors()` method is invoked the sensors are started one by one. The `Sensors` starts to collect events from the hardware.

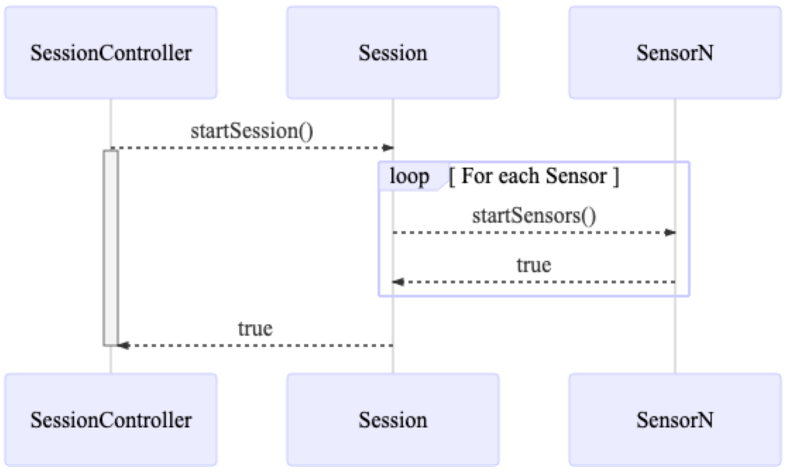


Figure 19: Sopor starting the data acquisition

In the current implementation, all `startSensor()` methods are called synchronously. One could improve the current implementation by setting up the method calls in such a way that they are run in parallel as shown in the pseudocode presented in Listing 32. Since this does not give a noteworthy speed increase in the current implementation due to the fact that all sensors are local to the Apple Watch it has not been implemented. Once starting to work with sensors that need to connect through Bluetooth which takes longer to set up, performing the setup in parallel would give a beneficial speed increase.

Listing 32 Starting Sensors in parallel

```
func startSensors(currentsession:Session) -> Bool {
    let asyncTask = async {
        sensors.parallelMap {$0.startSensor(session:currentsession)}
    }
    asyncTask.start()
    asyncTask.wait()
    return asyncTask.isSuccessfullyCompleted
}
```

Collecting

When the user is in an active session, the sensors are constantly emitting events to the `Session`'s `events` array. Depending on the sampling rate of the connected sensors this array may grow with everything from 5-200 events every second. The implementation of how the data is collected by the different sensors is discussed in Section 5.2.1.

Since all sensors inherit from the base `Sensor` class, every sensor also inherits the `storeEvent()` method. The `storeEvent()` method takes a `Data` object and calls the `appendToEventArray()` method of the `Session` object. This is the main way of collecting events for all the sensors, and every sensor has a contractual obligation to emit events to the `storeEvent()` method. Figure 20 shows how the created `Data` object is passed to the current `Session` object.

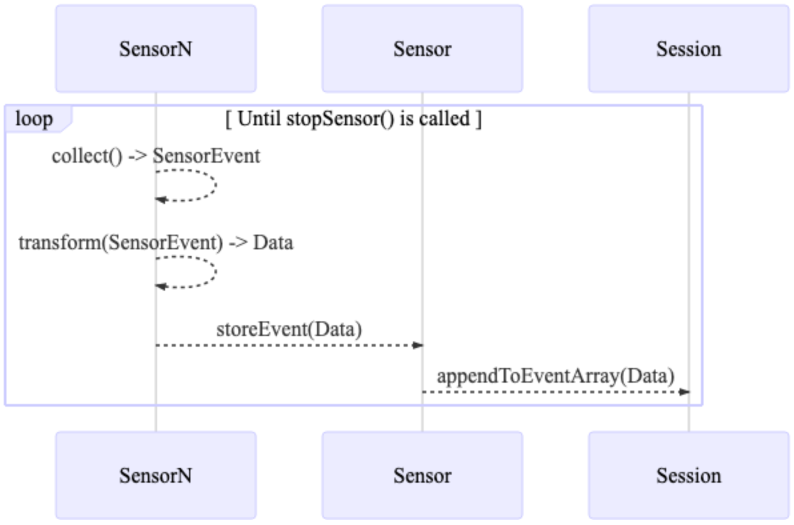


Figure 20: How data is collected and sent to the Session

Stopping the data acquisition

Once the user wakes up and wants to end the recording session, the user needs to hit the *End Session* button in the user interface. The touch triggers the `Session` to terminate and thus the `Sensors` are told to stop collecting events. Hitting the *End Session* button, as shown in Figure 21, essentially trigger the same operations as shown in Figure 19 replacing `start` methods with `stop` methods.

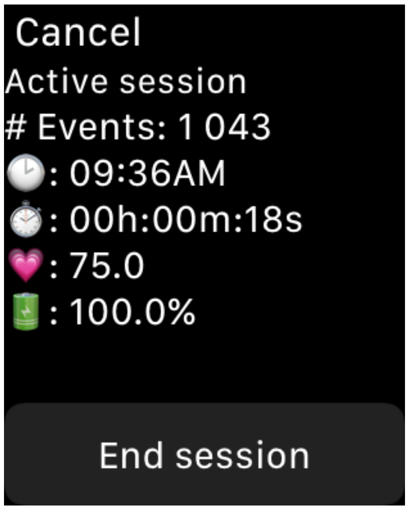

Cancel
Active session
Events: 1 043
🕑: 09:36AM
⏱: 00h:00m:18s
💗: 75.0
🔋: 100.0%

Figure 21: Sopor active session screenshot

5.2.2.2 Splitting and batching

As discussed in Section 4.3.2 and Section 5.1.6, there was a memory issue that occurred when running the application for a long time before storing the collected events in a different manner than just in-memory. Therefore, the design was rethought and the `SessionSplitter` class was created. This design allows batching of events, i.e. collecting for x seconds then handling those events and then discarding them from the array they were stored. This reduces the in-memory requirements and usage since memory is freed after each batch-period. This, however, introduces higher CPU usage as the processing of the events is not “free”.

Figure 22 shows the complete batching flow. The sensors continuously emit events to the `Sensor` object. Once the timer triggers, which is based on the `BATCHFREQUENCY` value set in the configuration file, the `SessionController` calls the `handleBatch()` method which triggers the `splitSession` method in the `SessionSplitter` class. The `splitSession` method, shown in Listing 22, returns a subset of the events in the `Session`’s `events` array to the `SessionController` which calls the `Sinks` with the returned array of data objects.

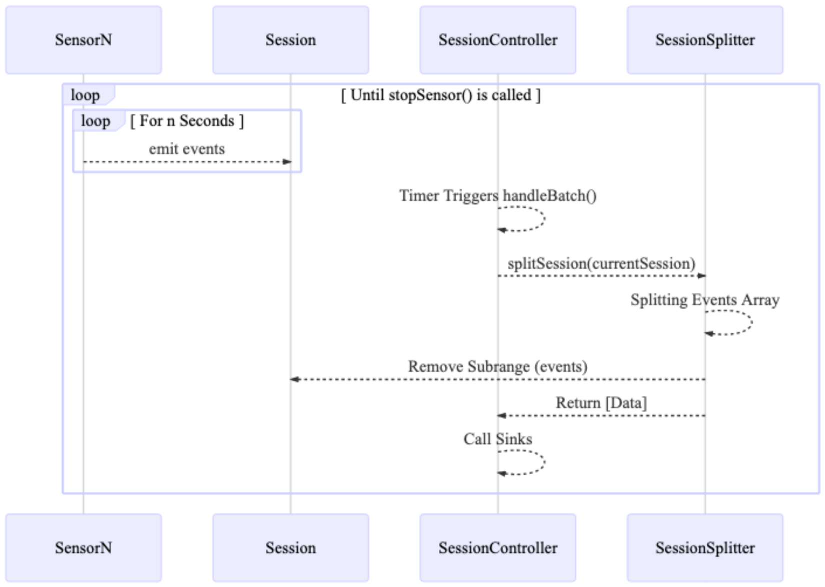


Figure 22: Sopor splitting flow

The current implementation triggers the `handleBatch()` method based on the elapsed time between the last invocation of the timer. Another solution possible solution could be to trigger the `handleBatch()` method based on how many objects are stored in the `events` array. The timer-based trigger was implemented since it was simpler to implement and easy to reason about in the code. For developers, the implementation is also easy to understand and the timer-approach is used multiple places in the codebase.

5.2.3 Sink

As discussed in Section 5.1.7, the `Sinks` are essentially event consumers returning the same or a mutated array of data objects, i.e., an event array where a function has been applied to the events. All Sinks implements the simple protocol shown in Listing 33, the protocol ensures that all `Sinks` take the same input- and output object type, namely an array of `Data` objects.

Listing 33 Sink protocol

```
protocol Sink {
    static func runSink(events:[Data],
                        options:[(String, Any)]
                        ) -> [Data]
}
```

For the initial work of Sopor, six different Sinks have been created. These are created in order to show different uses of sinks. Future development may use implemented sinks as a base when developing new ones. The sinks are discussed in the following subsections.

5.2.3.1 Console Sink

This is the simplest Sink that has been created in Sopor and is mainly used for development and debugging purposes. This is the purest Sink as well, it takes the input array and decodes it, prints it to the console, and then returns the initial array.

The use for this sink is mainly for the developer of a new sensor, it makes it easy for the developer to see what data is being passed to the sink and if the format is correct and according to the specifications.

5.2.3.2 Filter Sink

The Filter Sink takes an input array and an array of key-value pairs where one key needs to be named `sensorToRemove` and contains an array of `Strings` where each string represents one sensor to remove. Given the array of `Strings`, it removes all events that originate from a sensor in that array, and returns the filtered array. The pseudocode of the method is shown in Listing 34. This sink is useful in order to remove events from specific sensors, and is designed to be used in conjunction with other sinks. A trivial addition to the filter sink would be the *keep-sink*, which retains all events from a given sensor.

Listing 34 Pseudocode for filtering out events

```
func runSink(...){
    let sensors = options["sensorToRemove"] // gives [String]
    var mutatedEvents = events.filter({$0.sensorName in sensors})
    return mutatedEvents
}
```

5.2.3.3 Aggregation Sink

When running the sensors there may be times when it is unnecessary to handle every single event and instead only the average of a value in a given interval is interesting. One such example could be the accelerometer sensor which by default has a sampling rate of 30 events every second. For storing purposes, only storing the average accelerometer data for every second instead of 30 events for the same second is useful and saves storage space. In addition to reducing the storage requirements, the aggregation sink also contributes to reducing the number of bytes required to transfer when offloading data to the cloud, without loosing too much precision in the collected data-set

In the current implementation, the events that should be aggregated are removed from the list the method returns. It would also be useful to extend the functionality in such a way that the original events are not removed in the aggregation sink, however, such a feature would remove some of the benefits of reducing storage needs as discussed above.

In Sopor, there are a few aggregation functions that have been created, and some are only described here and have yet to be implemented.

Listing 35 Aggregation of heart rate events 1/2

```
static func runSink(events: [Data], sID:String) -> [Data] {
    var mutableEvents = events
    var aggregationEvents:[Int] = []

    mutableEvents.removeAll(where: {event in
        let json = // Function that gives an deserialized event
        let sensorName = json["sensorName"]
        if(sensorName == "Heart Rate"){
            let hr = json["event"]["heartRate"]
            aggregationEvents.append(hr)
        }
        return (sensorName == "Heart Rate")
    })

    mutableEvents.append(
        getAggHREvent(aggEv: aggregationEvents,
                              sessionIdentifier: sID))
    return mutableEvents
}
```

Listing 36 Aggregation of heart rate events 2/2

```
static func getAggHREvent(aggEv: [Int], sID:String) -> Data{
    if (aggregationEvents.count == 0) {return Data()};
    let sumHR = aggregationEvents.reduce(0, +)

    let aggEvent = AggregatedMetric(
            metricValue: Double(sumHR/aggregationEvents.count),
            type: "Avg(hr)",
            sessionIdentifier: sID)

    return AggregatedMetric.encodeEvent(event: aggEvent)
}
```

Average heart rate

This method takes the average heart-rate events and creates a single event containing only the average heart-rate for the given batch. The implementation of this sink is shown in Listing 35 and Listing 36. This implementation also removes all heart-rate events, while retaining all other events and the single aggregated heart rate metric. Variations of this aggregation function have yet to be implemented such as `Min` and `Max` of the batch as well as `Average` over a given timespan instead of the whole batch. The `Min` and `Max` functions are trivial to implement, however, an average over a given timespan requires more logic, and possibly a temporary storage location if the timespan crosses two or more batches.

Average accelerometer

The accelerometer on the Apple Watch can handle a high sampling rate over a long period of time, however, if the purpose is to store or transfer this data it can be storage intensive. Instead of setting down the sampling rate of the accelerometer sensor, the sampling rate is kept high but the average is calculated in this sink. With this design, the accelerometer data is as accurate as possible, while still keeping the storage needs low. One possible improvement over the current aggregation sinks is to make a single generalized aggregation sink. This sink would take a list of sensors to aggregate the data on, thus removing the need for specialized aggregation sinks.

5.2.3.4 File Sink

The `File Sink` writes the contents of the events to local storage. This is useful if the network connection is unreliable or not present, in addition, it can be used to extend the application to perform on-device analysis after a session has been completed. The File Sink introduces the concept of `buckets`, i.e., a folder with a certain number of batches, the number of batches in each bucket is determined in the initial configuration of the Sink. When using the `File Sink`, every session creates a directory on the format `Session_starttime_UUID`. Every `Bucket` is named according to the earliest and the latest event in that single bucket, therefore the directory name is often changed multiple times during a session, the naming convention is `Bucket_start-timestamp_end-timestamp`. The filename for which a batch is created has the format `earliest-timestamp_latest-timestamp` and is never changed once created. The File Structure created by the `File Sink` is exemplified in Listing 37.

Listing 37 File structure created by the File Sink

```
|-- Session_1580335200237_3a7de1c9-eb15-4ae9-8cf5-fc4fc8377074
|    |-- Bucket_1580335200237_1580335260887
|    |    |-- 1580335200237_1580335210301.json
|    |    |-- 1580335210322_1580335220873.json
|    |    |-- ...
|    |    |-- ...
|    |    |-- 1580335250206_1580335260887.json
|    |-- Bucket_1580335260891_1580335320938
|    |    |-- 1580335260891_1580335270653.json
|    |    |-- 1580335270702_1580335280762.json
|    |    |-- ...
|    |    |-- ...
|    |    |-- 1580335250322_1580335320938.json
|    |-- Bucket_...
|    |    |-- ...
|    |    |-- ...
|    |    |-- ...
```

Using this tree structure makes it easier to traverse the data since each batch file has the timestamps associated with it. This makes analyzing parts of the night easier due to the fact that the structure reduces the time it takes to find events that occurred at a certain time.

5.2.3.5 CloudKit Sink

In order to reduce the amount of data stored on the watch, a proof-of-concept `CloudKit Sink` is created. This Sink sends data to iCloud, and stores it in either a public or private database as described in [71]. When storing the data in the public database, data is accessible for all users. Storing the collected data in a public database is not something that should be put into production, but is useful for testing. When using this in a production setting, the data should be stored in the user's private database.

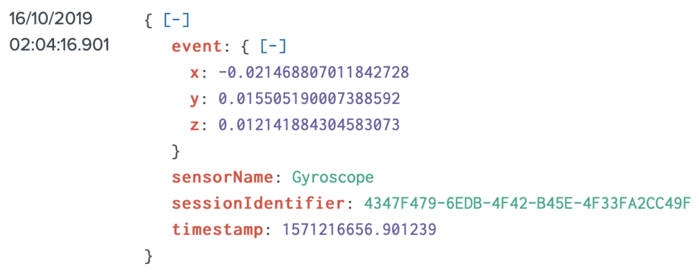
The CloudKit sink checks whether or not the user is connected to the Internet, if the user is online the data is stored in the user's private database, if not the data is stored to local storage and the application will try to upload the data once the session has finished. Pseudocode of the implementation is presented in Listing 38. In CloudKit, a schema has been created and is similar to the file structure described in Section 5.2.3.4, the complete schema is shown in Appendix Section D.3.

Listing 38 Pseudocode for the CloudKit Sink

```
func runSink(...){
    try:
        var container = CloudKit.getDefaultContainer();
        let privateDatabase = container.privateCloudDatabase;
        privateDatabase.saveRecords(events)
    catch:
        // Error occurred
        // Save to disk instead
        FileSink.runSink(events)
}
```

5.2.3.6 Splunk Sink

Splunk is an enterprise product created by Splunk Inc, it facilitates capturing, indexing and correlation of data in near real-time [72]. The product can also create graphs, reports, dashboards, and other visualizations. This `Splunk Sink` has been created in order to send data to the platform, showing that the data can be sent to multiple online locations. From the Splunk back-end, it is possible to create visualizations from sensor data that have been created. Splunk has a search query language called Search Processing Language (SPL), which is similar to Structured Query Language (SQL), and can be used to query the sensor data that Sopor has captured.



```
16/10/2019      { [-]
02:04:16.901      event: { [-]
                    x: -0.021468807011842728
                    y: 0.015505190007388592
                    z: 0.012141884304583073
                  }
                  sensorName: Gyroscope
                  sessionIdentifier: 4347F479-6EDB-4F42-B45E-4F33FA2CC49F
                  timestamp: 1571216656.901239
                }
```

Figure 23: Splunk event

Figure 23 represents one searchable `Gyroscope` event inside of Splunk. The JSON format is interpreted and neatly formatted by the Splunk system, and is searchable using SPL. The field `event` in the JSON structure contains the data collected by the Gyroscope sensor. Using the query in Listing 39 it is possible to generate graphs such as the one shown in Figure 24. The chart shows the average values every 50 milliseconds for the different axis of the gyroscope over a timespan of 15 seconds.

Listing 39 SPL for charting gyroscope data

```
sessionIdentifier="4347F479-..." host=Apple_Watch
| timechart span=50ms
   avg(event.x), avg(event.y), avg(event.z)
```

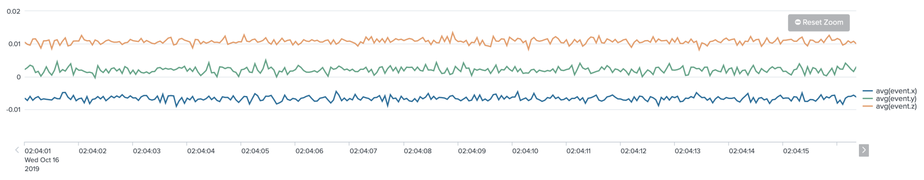


Figure 24: Gyroscope timechart in Splunk

5.2.3.7 State change Sink

When going through the batch it would be of interest to only keep an event if the datapoint has changed since last time, i.e., a change in the state. This Sink has yet to be implemented, but a possible version written in pseudocode is shown in Listing 40. The idea here is that while we sample the battery level every 10 seconds it often returns the same value, therefore it is unnecessary to store what is essentially the same event more than once. This sink filters out the same events and only keep the event if there is a change from the last known data-point. An extension of this would be if the data-point is within x% of the previous stored data-point, this would allow only spikes in data-points to be stored.

Listing 40 Pseudocode for keeping event on change

```
func runSink(...){
    let batterySensor = "Battery" // Gives [String]
    var lastBatteryStatus = // Last stored battery statys
    var mutatedEvents = events.filter(
        {
            if ($0.sensorName == batterySensor){
                if (lastBatteryStatus == $0.batteryLevel){
                    return true // Remove
                }else{
                    // New battery level
                    lastBatteryStatus = $0.batteryLevel
                    return false // Do not remove
                }
            }else{
                return false // Do not remove
            }

        })
    return mutatedEvents
}
```

5.2.4 Event Management and Storage

In Sopor, the event management is handled by the `SessionController`. It could very well have been implemented in a separate class, however, to keep it simple and not introduce unnecessary complexity it is added to the `SessionController`. The different sensor classes are responsible for putting the events in the specified array in the `Session` object for the currently running session. From here the event management of the `SessionController` takes over.

Every x seconds, defined by the `BATCHFREQUENCY` configuration variable, the SessionController's event management function calls the `SessionSplitter.splitSession()` method. This method splits the current `event`-array into two arrays, one which is passed through the event management process and a second one, the original, is just emptied. Next, the event management process calls the sink functions as defined in the `SessionConfig` class, see Section 5.1.5. Once events are passed to the sinks it is the sink's responsibility to handle the events and for example, if it is a storage sink; store them in the specified location. Making sure the events are stored correctly is out of the scope of the event management process, as it only makes sure to collect the events and pass them on to the specified sinks. The sinks themselves are responsible for handling potential errors and failures.

5.2.5 User interface

The user interface in Sopor is written in SwiftUI, Apple's declarative UI language which launched in 2019. When starting the development of Sopor SwiftUI was in early beta stages. Therefore, during the development, multiple rewrites had to be done. In the end, the interface was made rather clean in order to serve all types of users as well as reducing the amount of code in case another rewrite has to be performed. The declarative nature of SwiftUI makes it easy to take the user interface and port it to other Apple platforms since the language only expresses the logic behind the view but not the actual control flow.

Table 8: Showing the different views of Sopor

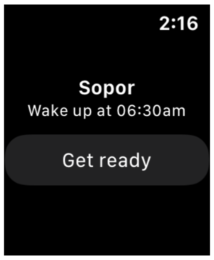
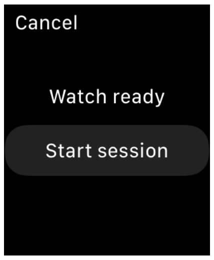
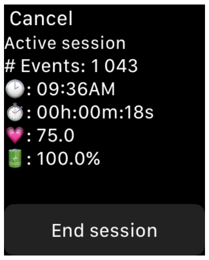
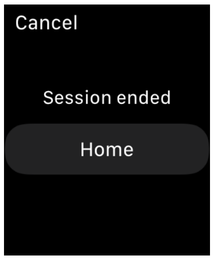
A	B	C	D
		Cancel Active session # Events: 1 043 🕑: 09:36AM ⏱: 00h:00m:18s 💗: 75.0 🔋: 100.0% 	

Table 8-A shows the `Home` view for the application. In the current implementation, there is only one button that takes the user to the `Ready` view in Table 8-B. In future versions, there should be a button to a `Settings`-view. In a `Settings` view, the user could make adjustments to the out-of-the-box configurations, such as removing or adding which sensors to activate, changing the sampling rate, and changing the `BATCHFREQUENCY`.

Before starting the session, the user is presented with the view shown in Table 8-B. This is a very simple view, and is defined in Listing 41. When presented with this view there's not much for the user to do except hitting the `Start session` button. This triggers the `startSession()` method as described in Section 5.2.2. The view in Table 8-C is shown in Listing 42, and the most complex view in Sopor. It gives information about the current `Session`, and has one button to end the session, which takes the user to the view shown in Table 8-D.

Listing 41 Declarative user interface for starting a Session

```
var body: some View {
    VStack{
        if(sessionStarted){
            // ...
        }else{
            Text("Watch ready")
            Button(action: {
                self.sessionStarted = true;
                _ = self.sessionController.startSession(
                        wakeUpTime: Date());
            }) {
                Text("Start session")
            }
        }
    }
}
```

Listing 42 View for an active session

```
var body: some View {
    VStack{
        if(sessionStarted){
            VStack(alignment: .leading){
                Text("Active session").font(.caption)
                Text("# Events: \(self.numberOfEvents)")
                Text("Time: \(self.current_time)")
                Text("Duration: \(self.duration_string)")
                Text("HR: \(self.hr_rate)")
                Text("Battery: \(self.batteryPerc)")
                Spacer()
                Button(action: {
                    // End button pressed
                    self.sessionEnded = true;
                    self.sessionController.endSession()
                }) {Text("End session")}
        }else{
            // ...
        }
    }
}
```

5.3 Miscellaneous

5.3.1 Chaining sinks

In the `SessionConfig` class, it is possible to configure the way the sinks are used when processing the batch of events. Listing 43 shows an example of how multiple `Sinks` can be chained together, the first sink prints directly to the console, the next one filters out all battery events, then all heart rate events are aggregated. Finally, the remaining events are sent to Splunk through the `SplunkSink`. This pattern makes it easy to extend Sopor with new functionality and new `Sinks` can simply be added to the chain or replace parts of it. The sink pattern also gives the developer a lot of options when it comes to parsing and handling the batch of events that are collected by Sopor.

Listing 43 Enhanced runSinks method

```
static private func runSinks(events:[Data], UUID:String)
{
    let eventsPrinted = ConsoleSink.runSink(events: events, [])
    let filtered = FilterSink.runSink(events: eventsPrinted,
                                  options: [(sensorsToRemove,
                                            ["Battery"])
                                           ])
    let aggregated = AggregationSink.runSink(events: filtered,
                                  options: [(uuid,sessionUUID),
                                           (aggregate,
                                            "HeartRate")
                                           ])
    let uploadedToSplunk = SplunkSink.runSink(events: aggregated,
                                           [])
}
```

5.3.2 Using Machine Learning on events

The scope of this thesis does not touch analysis of the data, however, a reflection upon it is important in order to understand the usage of the data-points which the application collects. Analyzing in this context is processing the data in such a way that useful information can be derived from the data. In the case of Sopor, data from the sensors may be able to assist in discovering useful information about a user's sleep pattern and improve the knowledge of the usefulness of smartwatches as a wearable device that can collect data in a useful manner.

Due to the amount of data that can be collected from the watch, it would be useful to use an approach that can analyze a lot of data automatically such as a machine learning algorithm in order to make sense of the collected data. One example would be training a machine learning model to detect which sleep stage a user is in by using the accelerometer and heart rate data collected from the watch like [73]. One approach is to use the data and use unsupervised learning and the clustering approach in order to make clusters of the data-points. Another approach is to collect data from medical equipment while at the same time collecting data from the application. By using the medical data-points to label the watch data a trained model can be made. The latter is an example of a supervised learning approach using classification.

In Sopor, machine learning could be used with a trained machine learning model using Apple's CoreML framework. This framework allows machine learning models to be created using among other methods, labeled data, which can be used by the watch to classify collected data. One concrete example would be to label sleep stages based on heart rate data collected by the application in order to create a machine learning model. This model can then be bundled with the application and used while collecting heart rate data from new sessions. The model will be able to predict which sleep stage the user is in based on the heart rate, or a range of heart rate samples.

5.3.3 Delay-tolerant CloudKit uploads

One area that has not been implemented, but that should be implemented in the future is *delay-tolerant* upload of data. In the current implementation, if the upload fails the data is lost and no re-upload is tried. This poses a problem if the user's network is down for some period of time or if there are intermittent networking issues. This section outlines how delay tolerance can be implemented in `Sopor`. One solution is already discussed in Section 5.2.3.5, the proposed solution in this section handles the errors somewhat differently.

The theorized solution focuses on changing the implementation details of how the `Sinks` behave, specifically the `CloudKit` sink. Some error handling is already present if the CloudKit API fails to upload the data, the error handling can be extended to store the data it was unable to upload locally until the network is up and running again. There are essentially two ways to handle this error, one is to store the failed upload as a file in the file system, another is to store the data in memory until network connectivity is restored. For the theorized solution, we suggest storing the data in local storage as to not receive a low-memory warning from the system and possibly being killed by the operating system, if the network is down for large periods at a time.

Storing the data to disk means giving each batch a filename. One option is to name the file with `failed_first-timestamp_last-timestamp`, borrowing the file naming from Section 5.2.3.4. These files should be stored in a separate directory, e.g. `failed_uploads/`, this way the application knows in which directory to look for files when the network connection comes back up.

The sink essentially runs as usual, if an error is received the file is stored locally. Every time it successfully uploads a data-packet, it checks for any file in the `failed_uploads/` directory and tries to re-upload that file. The steps in the proposed solution are shown in Listing 44. If there is no Internet connection during the whole period of the session, the files will still be present in the `failed_uploads` directory upon completing the session. If this is the case, the files can be stored until the next session starts, where the user hopefully has been able to get the Internet connection back up. A delete strategy would need to be implemented as well. If there is not enough storage on the device the locally stored files would need to be removed to make space for new files that the application is unable to upload. A FIFO queuing system may be used if you want to remove old un-uploaded data to make space for the new data points, or a LIFO queuing system may be used, where the newest data is deleted and disregarded due to storage capacity.

Listing 44 Pseudocode for delay tolerant file upload

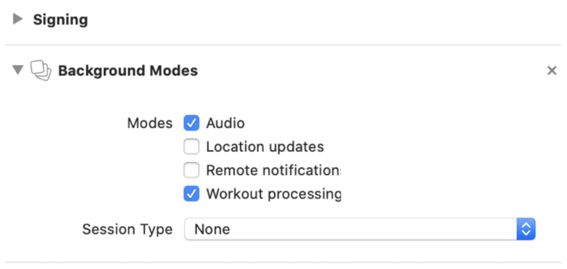
```
func successHandler(...){
    let files = getFilesFromDir("./failed_uploads/")
    for file in files{
        if(runSink(file)){
            // Is success
            // Delete file from disk
            deleteFile(file)
        }
    }
}

func runSink(...) -> Bool{
    var container = CloudKit.getDefaultContainer();
    let privateDatabase = container.privateCloudDatabase;
    let success = privateDatabase.saveRecords(events)
    if (success){
        // Network connection has been restored
        successHandler()
    }else{
        // Error occurred
        // Save to disk instead
        FileHandler.storeAsFile(events,
            "./failed_uploads/failed_first-timestamp_last-timestamp")
    }
    return success
}
```

5.3.4 Long-running sessions

One of the major issues when creating a watch application is to be able to collect data also while the user is not actively looking at the watch, i.e. when the screen turns black. Once the screen turns black the application would normally enter background mode and eventually it will be suspended. In early implementations of Sopor, this was an issue, however, it took some time before this bug was discovered. When using the Xcode, the IDE used to create iOS and watchOS applications, the application runs in the foreground and will never enter background mode even though the screen turns black. Due to this, it seemed that the application worked nicely even when the wrist was turned away from the user and the screen turned off. Since most early testing was done through the IDE, this was not an issue, however, when starting testing the application during the night this quickly became a major issue.

Once the application runs without the IDE, the application collects data when the screen is on and in the foreground, however, once the watch display turns off the data collection immediately stops. This is not the behavior Sopor should have since the collection of data should continue in the background even while the watch display is turned off.

Searching online and reading Apple's documentation it was clear that applications that start workout sessions using the `HKWorkoutSession` API were allowed to run in the background for the whole duration of the workout session. In Sopor this had to be done anyway due to the fact that we want to get near-real-time access to the heart rate data. Simple enough, however, what was not clearly described at the time of implementation was the fact that the application not only has to start a workout session, it also needs to have the correct entitlements in order to being allowed to run in the background. Thus, implementing the workout session by itself is not enough as it does not resolve the issue. The application needs to apply for the workout processing entitlement in order to be allowed to process data in background mode. Once the correct entitlement is given to the application, as shown in Figure 25, Sopor can run in the background without any modifications to the code.


Signing

Background Modes

Modes: [✓ Audio] / Location updates / Remote notification / [✓ Workout processing]

Session Type: None

Figure 25: Setting the correct entitlement in Xcode

5.3.5 Virga - Data processing application

One issue with storing the data in the cloud is that it is not readily available for processing and an application to get the data is needed. This has been implemented in an application called `Virga`.[27] This application fetches data

[27]Latin for precipitation from the cloud.

from iCloud and creates charts based on the collected data. `Virga` is written in `F#` which is a programming language designed by Microsoft. It is a functional and a strongly-typed programming language. Due to these paradigms, it is easy to reason about and understand the code.

The application is divided into five separate parts, described in Table 9. These parts are elaborated upon in this section. At the end of this section, there are some examples of data collected by `Sopor` and processed using `Virga`.

Table 9: F# project files

File	Description
Program.fs	Main program lives here.
CloudKit.fs	Gets the location (HTTP endpoints) for the data which will be processed.
CloudKitProcessor.fs	Handles fetching and processing of data.
Charts.fs	Responsible for creating the charts.
SensorDomain.fs	Contains record types for the Sensor Events. Shown in Appendix, Section E.2.

5.3.5.1 Program.fs

The main program is defined in the `Program.fs` file, it administrates calling the functions in the other modules. Due to this, there is not much logic in this module, just a few helper functions allowing for quick composition of new variations of the same application. The general flow of the application is shown in Figure 26, and a code snippet of a bare-bones `Program.fs` implementation is shown in Listing 45. In simple terms, the application first fetches a list of sessions, then it gets all data from that particular session by calling the correct CloudKit APIs, finally, it creates the charts needed to display a sleep session.

Listing 45 Simple Program.fs

```
let main argv =
    printfn "** Loading sessions..."
    let chosenSession = getSessionList () |> chooseSession

    let samplingData =
        chosenSession.sessionIdentifier
        |> CloudKitProcessor.fetchSamplingData

    if samplingData.Length = 0 then raise (Exception "Empty List")
    let batteryChart = Charts.createBatteryChart samplingData
    batteryChart.Show()
```

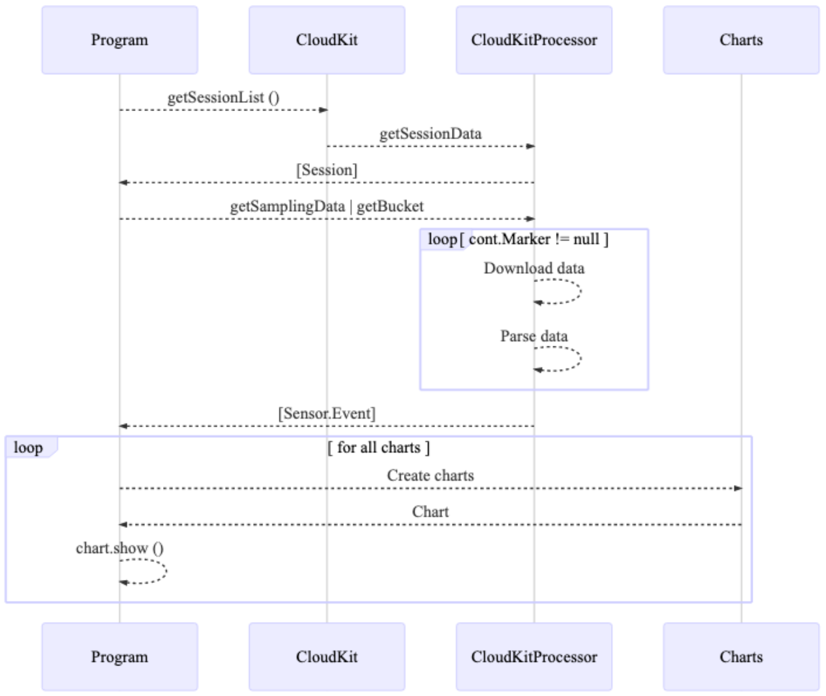


Figure 26: Virga program flow

5.3.5.2 CloudKit.fs

The current Sopor version is configured to use the CloudKit sink which uploads data to iCloud. This is an easy and reliable integration, however, getting the data back from iCloud is a little more tricky. Luckily, Apple provides a REST API that can be used to collect the stored data. Since the iCloud container is essentially a database, it is possible to filter out elements by passing queries in the body of the HTTP call to the back-end server. The CloudKit module in Virga is responsible for fetching data from iCloud.

When first setting up the CloudKit backend for use with Sopor, a schema has to be defined. This schema is shown in Figure 27 and presented in detail in Appendix Section D.3. Figure 27 essentially describes that a `session` record has a foreign key to two record types; 1) a `bucket` containing the raw sensor data as well as an event-count and 2) a `samplingEvent` type consisting of the aggregated number of events as well as the battery level. In addition, there are default system fields, i.e., metadata,[28] which are not shown.

[28]The metadata fields can be found here: https://developer.apple.com/documentation/cloudkit/ckrecord.

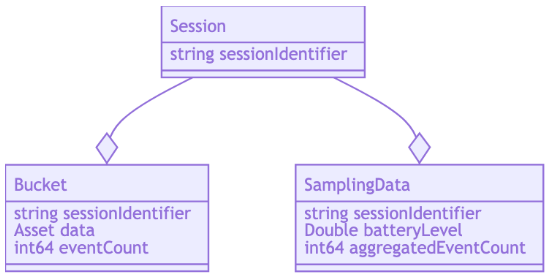


Figure 27: Relationship schema for the records

When fetching data from the container the application needs to perform a `POST` call to the CloudKit back-end, passing the appropriate query. The REST API endpoint is built up with the following elements: *[path]/database/[version]/[container]/[environment]/[database]/records/ query?ckAPIToken="token"*, all of which are described in Table 10.

Table 10: CloudKit parameters

API Parameter	Description
path	The URL to the CloudKit web service.
database	Static parameter. It indicates that we want to access a database.
version	The version of the API. The current version is *1*.
container	The container of the application. This is unique for all applications.
environment	Environment of which the application is running in. It is either *production* or *development*. If not on the App Store this should be `development`.
database	The database of which the data is stored. Possible values are `public`, `private` and `shared`.
records	Static parameter. It indicates that we want to fetch records.
query	Static parameter. It indicates that we want to perform a query on the data.
ckAPIToken	Token used to authenticate the request.

When calling the CloudKit API the response is an array describing the results of the query and the optional filter. The JSON response contains one or two keys; 1) `records` and potentially, 2) `continuationMarker`. The `records` value is an array of data objects described in the CloudKit schema, in general one of the three record types described in Figure 27. The `continuationMarker` is an optional value in the response and if the response contains more than 200 records, a continuation marker is passed which indicates that a follow-up REST call is necessary since there are more records available to be fetched. It is important to note that the `continuationMarker` has a limited time to live and will be invalid after some period of time.[29] When `Virga` sends the API request to the CloudKit back-end the HTTP request body needs to contain multiple fields, essentially telling CloudKit which records it wants to fetch. The different bodies used in the application are described in the Appendix, Section E.3.

Besides the different bodies used in the HTTP Request, the pure implementation of the `CloudKit` module in `Virga` is rather simple and most of it is shown in Listing 46. In addition, there is also a simple HTTP GET request function which is used to get a specific `asset` in the `Bucket` records.

Listing 46 CloudKit module implementation

```
module CloudKit

let fetch queryBody =
    let token = Environment.GetEnvironmentVariable "CLOUDKIT_TOKEN"
    let url = "https://api.apple-cloudkit.com/database/1/
                iCloud.com.bakkertechnologies.osa-tracker-watch.
                watchkitapp.watchkitextension/development/public
                /records/query?ckAPIToken=" + token
    Http.RequestString
        (url, httpMethod = "POST", body = TextRequest queryBody,
         headers =
            [ "Content-Type", "application/json"
              "Accept", HttpContentTypes.Json
              "User-Agent", "PostmanRuntime/7.22.0"
              "Host", "api.apple-cloudkit.com"
            ])
```

5.3.5.3 CloudKitProcessor.fs

The `CloudKitProcessor.fs` module handles business logic by calling the Cloud-Kit module and serializing the API response to the correct types. The defined record types are shown in the Appendix, Section E.4. This module provides three services, fetching all buckets, fetching the sampling data records and fetching a single data bucket. The services are described in the following paragraphs.

Fetching the *Bucket list*: When fetching `Bucket` records stored in iCloud one has to take into account the `Continuation Marker` that may come as a response from the API Call to CloudKit. If present it indicates that there are more records that matches the given query. Due to this, a recursive implementation

[29]The exact number of minutes is not described in the documentation. But through trial and error during development, this is between 30 to 60 minutes.

of the generic version of getting a bucket has been implemented and is shown in Listing 47. Essentially, the function fetches and parses a list of `Bucket` records, and then checks for the continuation marker and if present it recursively fetches them, if not it will return.

Listing 47 Get all Buckets recursive implementation

```
let rec getAllBuckets sID (contMarker: string option) i =
    printf "\rFetching bucket list: %d" i

    match contMarker with
        | Some x -> CloudKit.fetch (CloudKit.fullBodyBucket sID x)
        | None -> CloudKit.fetch (CloudKit.bodyBucketWithFilter sID)
    |> fun x -> (JsonConvert.DeserializeObject<CloudRecord>
                (x, JsonSerializerSettings(
                    MissingMemberHandling =
                        MissingMemberHandling.Ignore)
                ))
    |> fun x -> ( if isNull x.continuationMarker
                  then x.records
                  else x.records @ (getAllBuckets sID
                    (Some x.continuationMarker) (i+1)))
```

Upon return of the `getAllBuckets()` function it returns a list of `Bucket` records as described in the Appendix, Section E.4. These `Bucket` records can then be used to fetch the actual collected data from the Sopor application using the `fields.data.value.downloadURL` value for that `Bucket`, as shown in Listing 48.

Listing 48 Fetching Bucket data asset

```
let fetchBucketData i filterOut (bucket: Bucket) =
    printf "\rGetting bucket data nr: %d" i
    bucket.fields.data.value.downloadURL
    |> CloudKit.fetchBucket
    |> fun x ->
        (JsonConvert.DeserializeObject<SensorDomain.Event list>
            (x, JsonSerializerSettings(
                    MissingMemberHandling =
                        MissingMemberHandling.Ignore)))
    |> fun events ->
        match filterOut with
        | Some filter ->
            events
            |> List.filter (fun x -> (x.sensorName = filter))
        | None -> events
```

Fetching *samplingData* records: In addition to the buckets, there are also `samplingData` records uploaded to iCloud, these records contain useful information with regards to the experiments run in Section 6.1.1. These records are fetched and deserialized to `SamplingDataRecord` types using the `fetchSamplingData` function as shown in Listing 49. The returned *SamplingData list* can be used to create charts as discussed in Section 5.3.5.4.

Listing 49 Fetching SamplingData records

```
let getSamplingDataWithSession sID (cloudRecordStr: string) =
    (JsonConvert.DeserializeObject<SamplingDataRecord>
        (cloudRecordStr, JsonSerializerSettings(
                MissingMemberHandling =
                    MissingMemberHandling.Ignore)))
    |> fun x -> x.records
    |> List.filter (fun x -> x.fields.sessionIdentifier.value = sID)

let fetchSamplingData sID =
    sID
    |> CloudKit.sampleBodyWithSessionID
    |> CloudKit.fetch
    |> getSamplingDataWithSession sID
```

5.3.5.4 Charts.fs

To create charts `Virga` makes use of the `XPlot` package,[30] which is added to the project as shown in Listing 50. This package is an F# wrapper around the Google Chart API.[31] The `Charts.fs` module connects the different record types defined in `Virga` and creates plots using this package.

Listing 50 Adding XPlot.GoogleCharts to the project

```
dotnet add package XPlot.GoogleCharts --version 3.0.1
```

All events created by Sopor have a timestamp associated with it, defined in Unix time,[32] therefore two helper functions have been created in order to convert the Unix timestamp to the format `HH:MM`. This format looks better when generating the charts. Listing 51 shows these two functions. The *milliOrSeconds* function takes a timestamp string, and depending on the length of the string it will return the DateTimeOffset from Unix time seconds or milliseconds. The *getHHMM* function essentially converts the `DateTimeOffset` to the correct format. When

[30]The XPlot package can be found here: https://fslab.org/XPlot/.
[31]The Google Chart API documentation can be found here: https://developers.google.com/chart.
[32]Information about Unix Time can be found here: https://en.wikipedia.org/wiki/Unix_time.

working with the XPlot package it is important to pass in the correct timestamp in order for the chart to make sense.

Listing 51 Convert timestamp to correct format

```
let milliOrSeconds (timestamp:string) =
    let timestampInt64 = (timestamp |> int64)
    let secondsLength = 10 // e.g. 1582991605
    if timestamp.Length = (secondsLength) then
        DateTimeOffset.FromUnixTimeSeconds(timestampInt64)
    else
        DateTimeOffset.FromUnixTimeMilliseconds(timestampInt64)

let getHHMM timestamp =
    let dateTimeOffset = milliOrSeconds timestamp
    let dateTime = dateTimeOffset.UtcDateTime;
    dateTime.ToString("HH:mm")
```

Battery Chart: Using the `samplingData` records fetched from CloudKit a battery chart can be created. The function for creating the chart that is shown in Listing 52. It takes a list of `CloudKitProcessor.SamplingData` records and returns a `Chart` object. Upon return one just has to call the `Show()` method on the `Chart` object. Creating a chart consists of a few steps. First, a few configurations are defined and attached to the `options` value, then piping is used in order to define the variables for the chart itself, and finally, the chart is returned. Once everything is wired up correctly `Virga` will produce battery graphs like the one shown in Figure 28

Listing 52 Creating a battery chart

```
let createBatteryChart (data: CloudKitProcessor.SamplingData List) =
    let axis = Axis(minValue=0, maxValue=100)
    let options =
        Options
            ( title = "Sopor - Sleep Session Battery Life",
              legend = Legend(position = "bottom"), vAxis=axis )

    data
    |> List.map (fun x ->
        (getHHMM x.created.timestamp, x.fields.batteryLevel.value))
    |> List.toSeq
    |> Chart.Line
    |> Chart.WithLabel "Battery Level %"
    |> Chart.WithOptions options
    |> Chart.WithLegend true
    |> Chart.WithSize (1000, 500)
```

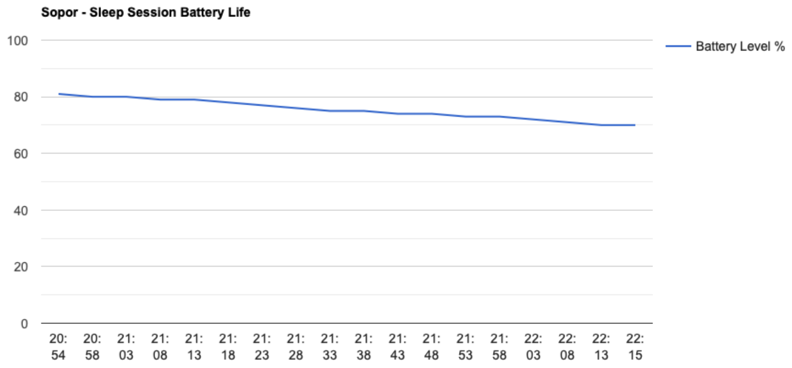


Figure 28: Example of battery events in a chart

In addition to the battery chart function, there are also functions defined to create graphs of a user's heart rate, shown in Figure 29, and the aggregated event count for a session as shown in Figure 30. Figure 31 shows data-points for a complete sleeping session, here it is possible to see when the user is in deep sleep, and when the user is more awake.

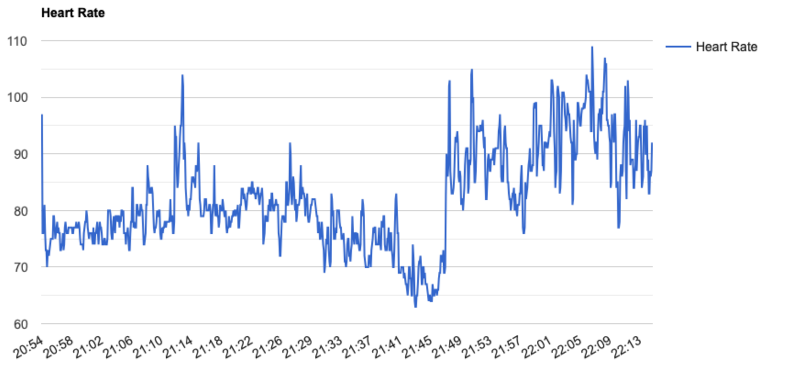


Figure 29: Example of heart rate events chart

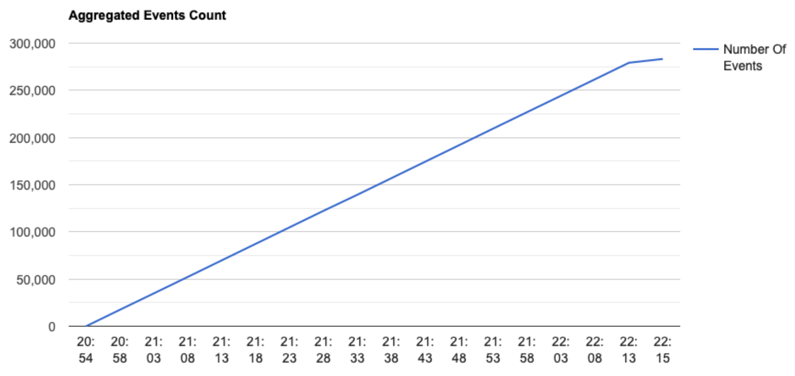


Figure 30: Example of aggregated events count chart

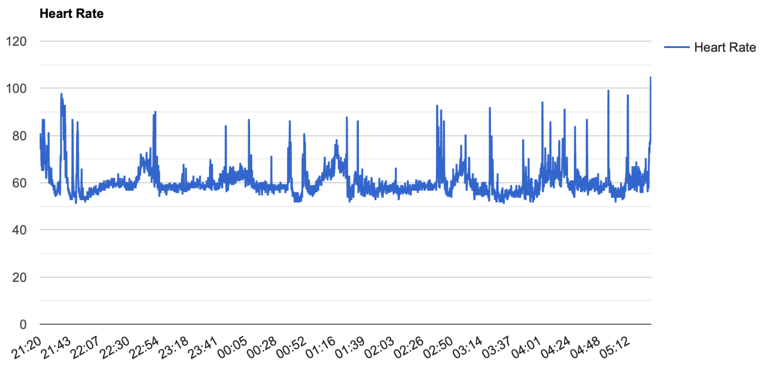


Figure 31: Heart rate graphed for a complete sleep session

Part V

Evaluation and Conclusion

6 Evaluation

In order to evaluate the application, three experiments are performed. The first two regards the actual performance of Sopor and the last one examines its ease of extensibility. Next, an evaluation is performed focusing on whether or not the high-level requirements, defined in Section 4.1, have been met and to which extent the functionality of the application has been fulfilled.

6.1 Practical experiments

To show that Sopor fulfills the requirements defined in Section 4.1 a series of experiments are performed. These practical experiments focus on running the application and analyzing the results and the collected data. The first experiment examines the application's battery usage, the second focuses on the CPU and memory usage during a session and the last one shows whether or not it is easy to extend the application. For the experiments, an Apple Watch Series 3 42mm GPS purchased in October 2017 has been used.

6.1.1 Experiment 1 - Battery usage

This experiment is done to explore whether or not Sopor can collect data for a full sleep session, i.e., eight hours. Battery loss is inevitable while using the application, but as discussed in Section 4.1.2 one requirement should be that the application should not deplete the battery more than 50% in order to allow the user to use the watch during the day as well. This experiment also demonstrates whether or not the application is suitable for collecting data over a longer period of time.

Description

The experiment is done in multiple runs using different configuration settings. Each session collects data for eight hours. The hypothesis is that the battery life is given mostly by the `BatchFrequency` variable and the sampling rate of the accelerometer. The network request performed on each `BatchFrequency`-trigger is likely to affect the results the most. A control run is also performed which includes not running the application when wearing the watch for eight hours while sleeping. This test enables us to see how much battery is drained just by running the operating system and regular tasks on the watch. In addition, the test allows us to calculate the actual contribution with regards to battery drainage of the Sopor application.

The sensors used in this experiment are outlined in Table 11.

Table 11: Sensors used in Experiment 1

Applied sensor wrappers
BatterysensorWatch
AccelerometerSensor
HeartRateSensor
MetaSensor

The only sink used in this experiment is the CloudKit sink without any modifications to the data-set. This is done in order to more easily analyze the data-set upon completion. The *runSink* configuration function is shown in Listing 53.

Listing 53 Sink setup in Experiment 1

```
static private func runSinks(events:[Data], UUID:String){
    let _ = CloudKitSink.runSink(events: events)
}
```

Variables

During the different runs, different variables are changed, these are shown in Table 12. This is done in order to see how different watch settings affect the results. During the different runs, we changed the *BatchFrequency* and the *Accelerometer sampling rate* as the hypothesis is that these two variables contribute the most to battery usage.

Table 12: Variable changes for Experiment 1

Setting	Run 1	Run 2	Run 3	Run 4
BatchFrequency	$\frac{1}{5}$	$\frac{1}{60}$	$\frac{1}{60}$	$\frac{1}{120}$
Accelerometer Sample Rate	60Hz	60Hz	30Hz	30Hz
Battery Sample Rate	5 seconds	5 seconds	5 seconds	5 seconds
Heart rate	Collecting	Collecting	Collecting	Collecting

Experiment setup

The experiment is run during the night in bed at home for eight hours, using the Apple Watch Series 3 42mm Aluminum version. Before the experiment starts the watch is charged to 100% and restarted as to start fresh. During the experiment, the watch is set to *Do Not Disturb* mode and *Theatre Mode* is activated. The application is turned on once in bed and turned off when waking up after eight hours. If the subject sleeps more than eight hours, the data for the first eight hours of the session is used. Each `Run`, shown in Table 12, is performed three times and the average over the three different sessions is used as the result for the specific run.

Results

The results from the different runs are presented in Table 13 and Table 14.

Table 13: Results from Experiment 1

Run	Avg. Battery % depleted	Max Battery % depleted	Min Battery % depleted	Standard deviation
1	76,33%	78%	73%	2,887
2	57,00%	63%	49%	5.888
3	50,66%	54%	45%	4,933
4	45,00%	48%	42%	3,000
Control	9,33%	15%	6%	4,028

Table 14: Number of events collected

Run	Avg. Number of Events collected
1	1.700.133
2	1.695.387
3	871.484
4	876.142

The results clearly show that it is the `BatchFrequency` that has the biggest effect on battery life. `Run 1` depleted nearly 20% more battery than `Run 2`, where the only configuration difference was the `BatchFrequency`. Reducing the frequency of the accelerometer sensor also contributed positively to the battery, depleting ~5.5% less battery with the lower frequency. Reducing the `BatchFrequency` from $\frac{1}{60}$ to $\frac{1}{120}$, i.e., uploading every 120 seconds instead of every 60 seconds, also contributed positively to the battery life, ensuring ~5% less battery drain during the eight hours.

Discussion

The results show that it is possible to use the application for a total of eight hours without completely draining the battery. However, depending on the configuration the battery usage is different. The hypothesis that frequently offloading data increases the battery usage is clearly shown between `Run 1` and `Run 2`. Between `Run 3` and `Run 4` the improved battery life is not as drastic, but it is noticeable. Between `Run 1` and `Run 4` the battery usage decreased with ~31% showing a clear improvement by reducing the frequency of the accelerometer and most importantly reducing the `BatchFrequency`.

During a session, it is difficult to know whether or not the watch is performing other background tasks. If the watch suddenly starts background tasks it will impact the battery life, this is not something we can control. This is one of the reasons why there are large standard deviations in each run. Taking into account that the operating system is running tasks in the background as well, the actual results show that `Sopor` contributes somewhere around ~36% when using `Run 4` and the `Control Run` as the data-points.

It is important to note that the Apple Watch used in the experiments has been heavily used over a few years, meaning the battery has become weaker, i.e., not having the maximum capacity when performing the experiments as compared to a brand new watch. This gives a better insight into the real-life situation for most users of the application. Most people will use the application on a watch that has been used for some time. A newer Apple Watch would probably have depleted less battery than the watch used in the experiment.

Conclusion

The goal of Experiment 1 is to show that the watch can function during a complete sleep session and show that the application uses less than 50% battery. The results of the experiment show that the requirements of the application have been met. The configuration in `Run 3` and `Run 4` shows that it is possible to have a configuration that depletes less than ~50% of the battery. Given the age of the test device, this is promising. A new version of the Apple Watch would more than likely have better performance and decreased resource usage.

6.1.2 Experiment 2 - CPU usage and memory usage

When running the application on the watch it is important to reduce the amount of CPU and memory the application uses. If the application uses too much CPU time over a given period, it will be killed or suspended by the operating system due to excessive resource usage [74]. In addition, it is important to reduce the memory usage to a minimum as the application may be killed as well when using too much memory. Finally, the experiment shows whether or not there are any memory leaks in the application.

This experiment gives insight into how well the application performs with regards to the two variables, CPU and memory usage. While no numbers are given by Apple with regards to the CPU % and memory usage, Xcode gives an indication of CPU usage through a gauge chart as shown in Figure 32 and Figure 33.

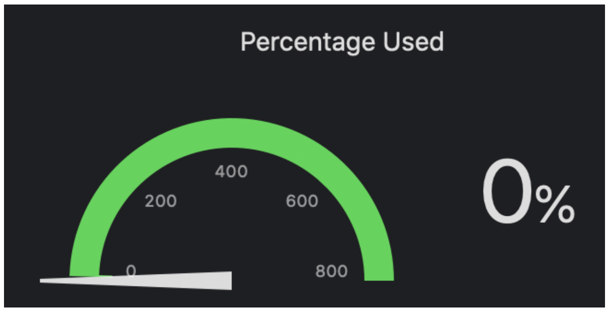


Figure 32: Xcode CPU gauge

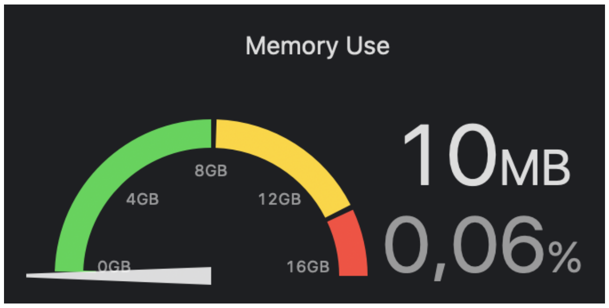


Figure 33: Xcode memory gauge

Description

The experiment is performed using Xcode and the built-in tools of the IDE. The same Series 3 Apple Watch as used in Experiment 1 is used. Newer versions of the Apple Watch have increased CPU and memory capacity, meaning that if the Series 3 Apple Watch can tackle the load, it is reasonable to think that newer Apple Watches can handle the load as well.

For this experiment, we hypothesize that the CPU usage will be constantly at a reasonable level with spikes every time the Sink is called, due to the CPU intensive operations that are performed in the sinks. The memory usage should be steadily increasing between batch uploads and reduced once data is offloaded.

Variables

In this experiment, the best configuration from Experiment 1 is used (`Run 4`), the configuration used in the experiment is shown in Table 15. In addition to being the best configuration from Experiment 1, the reduced batch frequency gives a better understanding and insight into the memory usage in the application.

Table 15: Apple Watch configuration for Experiment 2

Setting	Configuration
BatchFrequency	120 seconds
Accelerometer Sample Rate	30Hz
Battery Sample Rate	5 seconds
Heart rate	Collecting

Experiment setup

The experiment is performed three times, every time the application runs for one hour, collecting as it would do normally using the `CloudKitSink`. One hour is enough to understand the CPU and memory usage of the application. As with Experiment 1, the watch is charged to 100% and restarted before starting the experiment. During the experiment, the watch is set to *Do Not Disturb* mode, and *Theatre Mode* is activated. Each run is performed once with the same configuration each time.

Results

The results from the experiment are shown in Table 16. The results show that there is no excessive CPU or memory usage in the application, and at no point did the application receive any warnings from the operating system to reduce CPU or memory usage, which indicates that the application is behaving according to Apple's rules.

Table 16: Results of Experiment 2

Run	Avg. CPU usage	Max CPU usage	Max memory usage
1	3%	71%	18,4MB
2	4%	49%	17,9MB
3	4%	69%	17,8MB

There are spikes in CPU usage every `BatchFrequency` seconds as shown in Figure 34, indicating that the hypothesis presented earlier is correct. Every time the watch has to perform a network operation the CPU usage increases. The spikes are also caused by the amount of processing the watch has to perform to make sure all data-points are ready to be uploaded. By using additional sinks one can assume that both the average CPU usage and the maximum CPU usage would increase.

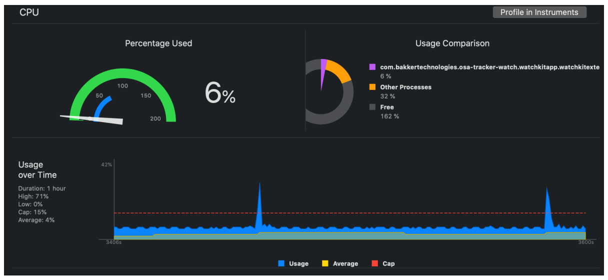


Figure 34: CPU spikes every BatchFrequency

An interesting observation is the memory usage of the application, it has a very predictable graph as shown in Figure 35. The memory usage keeps increasing until it can offload the data to CloudKit. Once offloaded, the cycle repeats.

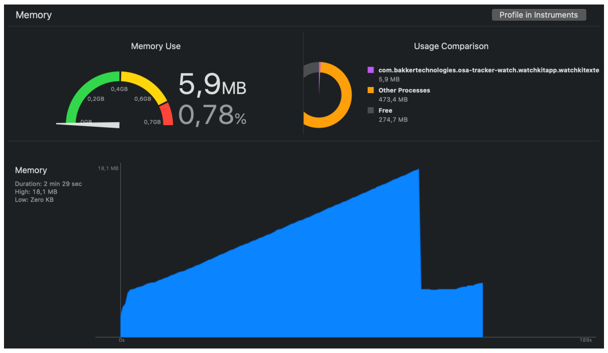


Figure 35: Predictable memory usage

Discussion

The application is well within the limits of the operating system when running. The low CPU usage indicates that the application is able to function well over long a period of time. Figure 34 shows a red dotted line which is a line added by the Xcode IDE that indicates that an average CPU usage above this line would result in receiving a high CPU usage warning. `Sopor` is only above this line whenever it makes a network request which does not contribute enough to impact the average CPU usage. There is also no indication of memory leaks in the application which is a very good thing. In an application like `Sopor`, a tiny memory issue could be fatal as it would build up over time and eventually crash the application.

Conclusion

The purpose of Experiment 2 is to show that the application has an average low CPU and memory usage when running on-device, as well as showing that what happens around the uploading of the data causes spikes in the CPU usage, these have all been shown in the experiment. At no point during the experiment did the application receive a low memory warning or indications of excessive CPU usage. The results show that the application satisfies the CPU and memory requirements presented in Section 4.1.

6.1.3 Experiment 3 - Extensibility

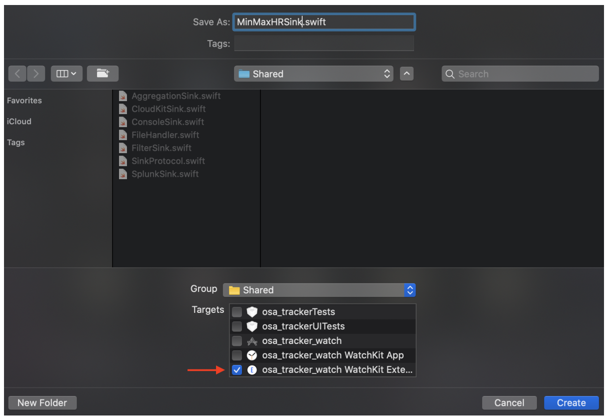
The goal of the following experiment is to show that `Sopor` can easily be extended. In this experiment, the application should be extended with a new sink. In order to show that it is easy to extend, a step by step guide is created. This should be sufficient to show that the application can be extended with multiple new features. According to [57] a system is extensible if new features can safely be added without having global knowledge of the system. This is what the experiment aims to demonstrate. This experiment also demonstrates that it is possible to chain multiple sinks together.

Description and setup

The sink to be created is a sink that finds the maximum and minimum heart rate for a given batch. It should then remove all other events, only keeping the two events collected. The experiment is performed using the same codebase as used in Experiments 1 and 2 which is the final version of the code in the thesis. In the following part, the experiment is performed and all the steps are documented as a step-by-step guide.

Step-by-step guide

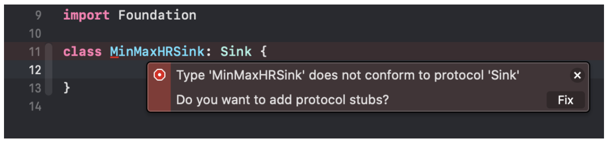
1. When first creating a new `Sink` a new Swift file has to be created. We create a file named `MinMaxHRSink.swift`. When creating the file make sure to add the file to the project's *Shared* directory and that and the correct *target* (`osa_tracker_watch WatchKit Extension`) is checked, as shown in Figure 36.



```
Save As: MinMaxHRSink.swift
Tags:
Shared

AggregationSink.swift
CloudKitSink.swift
ConsoleSink.swift
FileHandler.swift
FilterSink.swift
SinkProtocol.swift
SplunkSink.swift

Group   Shared
Targets [ ] osa_trackerTests
        [ ] osa_trackerUITests
        [ ] osa_tracker_watch
        [ ] osa_tracker_watch WatchKit App
        [✓] osa_tracker_watch WatchKit Exte...
```

Figure 36: Step 1

2. Next, we need to create the class and make it conform to the `Sink` protocol. We add the class definition and make it adhere to the Sink protocol. In Xcode, we can then build the project, and the compiler warns us that we need to add the stubs of the Sink protocol, that can be done by clicking *fix* as shown in Figure 37. Once hitting *fix*, Xcode updates the editor with the static `runSink()` method.



```
import Foundation

class MinMaxHRSink: Sink {

}
```

```
Type 'MinMaxHRSink' does not conform to protocol 'Sink'
Do you want to add protocol stubs?
```

Figure 37: Step 2

3. Step three involves finding the min and max heart rate of the batch. This is where we implement the actual logic of the sink. The implemented idea involves looping through all events in the batch, serializing them from a string object to a JSON object, and checking for min and max. Finally, it returns only the two heart-rate events. The complete method is shown in Listing 54.

Listing 54 The runSink method for MinMaxHRSink

```
static func runSink(events: [Data]) -> [Data] {
        var min, max = -1
        var minEvent:Data? = nil
        var maxEvent:Data? = nil

        for event in events{
            let serJson = try? JSONSerialization.jsonObject(
                    with: event, options: []) as? [String:AnyObject]

            if("Heart Rate" == serJson!["sensorName"] as! String){
                let hr = serJson!["event"]!["heartRate"] as! Int
                // Check if the HR is min or max
                if(hr < min || min == -1){
                    minEvent = event
                    min = hr
                }else if(hr > max || max == -1){
                    maxEvent = event
                    max = hr
                }
            }
        }
        return [minEvent!, maxEvent!]
    }
```

4. After the class and its methods are implemented, the sink configuration needs to be changed. In this experiment, we add the *MinMaxHRSink*'s `runSink()` method before the *CloudKitSink*'s `runSink()` method, resulting in only storing the minimum and maximum heart-rate for the batch. The final configuration is shown in Listing 55.

Listing 55 Changed configuration for Experiment 3

```
static func runSinks(events:[Data], UUID:String)
    {
        let hrMinMaxEvents = MinMaxHRSink.runSink(events: events)
        let _ = CloudKitSink.runSink(events: hrMinMaxEvents,
                                     sessionIdentifier: UUID)
    }
```

5. Finally, build and run the project. That's it.

Results

The step-by-step guide from the experiment shows that only five steps are required to extend the application with more sinks. This without having a deep understanding of how the sensors collect data. The most challenging part of the experiment is creating the actual logic of the sink itself. By adhering to the contract and protocols that are described in Section 5.2.3 it is simple to extend the application as shown in this experiment.

Discussion

While it is easy to add and remove sinks, the developer still needs some knowledge of the actual representation of the events. Therefore, users would need to know details about how the events are structured in the JSON format. A developer also has to have some knowledge about the sink-structure, when working with sinks, however, this is to be expected when working with a sink-model. More knowledge about other items than the JSON-structure and sink-model is not required for a developer to start extending the application with more sinks.

Conclusion

The goal of the experiment is to show that there is no need for a developer to have a complete understanding of the application in order to write features that will extend the functionality of the application. This experiment has shown that it is easy to write modules extending the functionality of the application without having global knowledge of the system.

6.2 Examination of requirements

This section concerns examining the high-level requirements defined in Section 4.1 and that the functionality of the application works as expected. When examining the requirements the results from the experiments performed in Section 6.1 are taken into consideration when performing the evaluation. The requirements are discussed individually and a summary is given at the end of the section.

Extensibility

Allowing `Sopor` to be extensible is an important aspect of the application. This requirement is shown to be met in Section 6.1.3, which shows that implementing a new `Sink` can easily be done using a few simple steps. Having global knowledge of the application is not required when extending the application. However, a user needs to have some basic understanding of the architecture and the event-structure when developing. The architecture and event-structure are described in Section 4 and should be easy to follow. Based on this, this requirement is seen as met.

Resource efficiency

The resource efficiency requirement concerned battery-, networking-, CPU-, and memory usage. When evaluating this requirement Experiment 1 and 2 are taken into consideration. From Experiment 1, insight is given into how much battery is depleted for a set of configurations. When outlining the requirement, it was decided that the watch should not deplete more than 50% during the night. Even the most aggressive configuration left the device with more than 20% battery left on average. The best configuration used 45% of battery on average. For an

Apple Watch which has been used for more than two years, this is impressive and is a good indication that the requirement has been met.

The networking requirement is somewhat more tricky to give a good indication as to how well it has been met. However, using JSON as the data format reduced the file uploads size with around 35% as shown in Section 4.4. In addition, using the CloudKit API allows the watch to have some control over the file upload. One improvement that has not been implemented is to zip the file before uploading, this could have reduced the file size a lot since there are a lot of repetitive sections in each uploaded file. In addition to zip-ing the file, the events generated could have been different abbreviations, thus reducing the size of each event. When it comes to the second sub-requirement, delay-tolerant networking, this has not been implemented, but a solution to the problem is outlined in Section 5.3.3.

Finally, the CPU and memory usage of the application is examined in Section 6.1.2, which showed that `Sopor` does not use excessive memory and CPU. In addition, there is no trace of memory leaks in the application. Therefore, this requirement is seen as met.

Data collection

The data collection part is satisfied in the application, the application is able to collect data from a variety of sensors on the watch. In the future, implementing other sensors would be beneficial to collect even more data. As the watches become more powerful one can assume that the watch is able to connect more sensors while still having reasonable resource efficiency.

Privacy

In the current implementation of `Sopor`, the application does not store data that can be used to track which user performed a given session, this makes the work of correctly processing the data somewhat difficult if the application gets more than a few users. The current implementation does not allow users to delete their data in an easy way. This needs to be improved if the application should be used by more users. The application does not take the current GDPR[ˆgdpr_link] rules into account, thus before the application can be used outside of a controlled group, a solution to allow users to delete and give explicit consent to collect user data has to be implemented.

In addition, if the application is to be used in a patient-medical doctor scenario, there needs to be some way for the patient to get the data and send the dataset to the doctor. This has not been implemented, but is not in the scope of this thesis.

Storage

Is was clear early on in the development cycle that storing full sessions on the Apple Watch would pose a challenge since it would fill up the available storage pretty quickly. However, using the method of offloading the data to a secondary storage location proved to be an efficient way to reduce the storage needs on the device. Two storage options have been implemented, the `CloudKitSink` using the CloudKit APIs, and the `SplunkSink` using regular `HTTP GET` calls to send the data to the Splunk server. As discussed in Section 5.3.3, it is possible to store data on the watch if the network connection is down, thus allowing for intermittent and temporary watch storage if necessary.

Stakeholders

In Section 4.1.6, three potential users are described: 1) end-users, 2) researchers and medical doctors, and 3) developers. It is a requirement that all these users should be able to use the application and make use of the data collected. For the end-users, the focus is on the user-interface. The application's user interface is simplistic and not hard to use. For group 2, researchers and medical doctors, it is important to be able to retrieve and process the data. This requirement is met by creating `Virga` the data processing application that can be used in conjunction with `Sopor`. `Virga` allows researchers and medical doctors to get access to the data-points generated by `Sopor`. `Sopor` also uses a data format and a data model that is easy to understand, the common JSON structure makes it easy to understand what a single data-point describes. The last group of stakeholders is the developers. A developer should be able to understand the application flow and the high-level design. This requirement is also seen as met since we can point to Section 4.3.3 to show the high-level design and all the components are discussed in Section 5.2. Finally, Experiment 3, in Section 6.1.3, shows that it is simple to extend the application. Therefore, this thesis meets the requirements with regards to the stakeholders outlined in Section 4.1.6.

Summary of requirements

Table 17 shows a summary of all the requirements and whether or not the requirement has been met.

Table 17: Summary of requirements

Requirement	Summary
Extensibility	This requirement has been met, as shown in Section 6.1.3.
Resource Efficiency	Requirement has been met, as shown in Section 6.1.1 and Section 6.1.2.
Data Collection	The experiments performed in Section 6.1.1 and Section 6.1.2 shows that this requirement has been met.
Privacy	This requirement has partially been met. In needs further work.
Storage	Multiple solutions to this requirement has been discussed and implemented.
Stakeholders	Requirement has been met.

7 Conclusion

7.1 Summary

This thesis is motivated by the ability to extend the CESAR project by creating a watch application that can be used to collect sensor data from a wrist-worn device. The goal is that the application should lay the groundwork and to ensure that a foundation is created for the future development of the CESAR project. By creating an application that allows different sensors to send data to a unified location and an application that can download and parse the data the goal of this thesis is met.

To achieve the goals, a watch application is designed and implemented. All identified concerns are separated, implemented, and finalized in an application named `Sopor`. The application allows users to record their sleep with little effort, thus having a low barrier to entry. The design chapter outlines five areas that are implemented: 1) Session recording, 2) Sensors, 3) Event Management and Storage, 4) Sink, and 5) User interface. The Session Recording is implemented using a controller class and a session class, these two interacting closely together. Multiple sensor wrappers are also implemented, giving access to valuable test-data for the experiments. The Event Management and Storage are handled by the controller class, making sure that the data collected is able to be accessed by the sinks. A sink pattern is implemented, and a contract is established between the session controller and the respective sinks. The contract states that once the sink has received the events it is the responsibility of the sink to make sure the data is stored correctly. Finally, a simple user interface is implemented allowing the user to see some data-points and information during the sleep session.

In addition to `Sopor`, an application named `Virga` is created. It allows the user to download data collected by `Sopor` which is stored in iCloud. By calling the CloudKit API and using the XPlot F# library the application is able to download the assets stored in iCloud and create graphs showing the data-points. This application is necessary to create in order to be able to retrieve and visualize the data. While there are no experiment testing `Virga`'s features, it is used heavily in Experiment 1 to fetch the data-points and to generate the graphs needed to answer whether or not `Sopor` satisfies the outlined requirements.

By performing Experiment 1, we show that the application is able to collect data over an eight hour period and that there are configurations that allow the watch to retain more than 50% of the battery at the end of a sleep session. The results show that reducing the `BatchFrequency` contributes a lot to the reduction in battery depletion, by setting the `BatchFrequency` variable to $\frac{1}{120}$, i.e., 120 seconds, the application is able to maintain more than 50% battery after eight hours. Experiment 2 shows that there are no memory leaks or excessive CPU usage. The experiment confirms the hypothesis that there are CPU spikes every time the application uploads data to CloudKit. The memory graph is shown to be predictable, building up towards every `BatchFrequency` seconds and then quickly reducing whenever data is offloaded. The final experiment, Experiment 3, gives an indication as to how easy it is to extend the application. The results show that a few steps are needed to add another sink. Being extensible allows other developers to continue the work, easily improve the application, and add new features.

7.2 Open problems

The thesis leaves a few problems which still need to be resolved and improved upon before the application can be pushed to production. The first issue that needs to be tackled is allowing the setup of Bluetooth sensors. There are multiple solutions to this problem, however, with the foundation that is created with `Sopor`, it should not be an issue to implement a sensor once the connectivity to the device is solved. This allows other sensors to contribute with data-points which would be useful for data processing.

Another open problem is how to pick up sounds from the environment and the user. As elaborated upon in Section 5.2.1.6 using the built-in microphone on the watch could be impractical, as the user might have their hand and wrist under their pillow, thus muffling any sounds created. The proposed solution to use a phone as the recording device is an open problem to implement and correlate data-points with the data from `Sopor`.

During Experiment 1, some bugs were found, which have yet to be fixed. One of which is that the workout session is not always stopped. Some time was given to solve this problem, but no solution has been found. It could very well be an issue with the API itself, this needs further investigation.

Finally, the last open problem to tackle is how to store the data in a privacy-friendly way. The only individuals who should have access to the data-points are the user and the medical professional once the correct approvals have been given. This problem is not tackled in this thesis, and needs to be designed and implemented before the application can be used in a medical setting.

7.3 Future work

There are many different roads the work from this thesis can take, there are tasks, experiments, and improvements that have been left for the future due to lack of time. Since one experiment often takes multiple 8-hour sessions, or multiple individuals to complete there is still much that can be investigated. In addition, some features would have taken too much time to implement for this thesis, thus they are left for future developers to implement. The following is a list of items that future developers should focus on:

1. It would be interesting to use machine learning on the data-set collected by the application. As discussed in Section 5.3.2, there are frameworks that allows labeled data-sets to train a machine learning model. This can be implemented in different ways, using `Virga` to download the data

and process locally on a machine, or improve `Sopor` to handle real-time machine learning processing of the collected data.

2. As discussed in Section 7.2, there is still no solution for storing the data in a privacy friendly way. In addition, the data should be stored in compliance with GDPR rules. This is important as the data collected and processed by the application contains sensitive information about the user. The application needs to implement the ability for the user to give consent to the application to allow it to process the collected data and allow a user to remove their data.
3. Together with medical professionals, the application should be tested in a controlled environment allowing data-points to be correlated with medical data-collection devices. Since medical equipment has high accuracy, it will be interesting to see how sensors on the watch compare to the equipment used by professionals.
4. Left for the future is also the implementation of more sensor wrappers. When new sensors become available on the watch, wrappers for these need to be implemented to allow data to be collected from them. The task to design and implement Bluetooth sensors is one that can be tackled immediately.

The designed, implemented, and tested applications, `Sopor` and `Virga`, creates a foundation for future work on the Apple platforms for the CESAR project. Left for the future are multiple open problems and suggestions for future work. Researchers can take the work from this thesis and further move our understanding of how wrist-worn devices can be used in a medical setting.

8 References

[1] U.S. Department of Health and Human Services, “NHLBI: Health Information for the Public,” 2012 [Online]. Available: https://www.nhlbi.nih.gov/health-topics/sleep-apnea. [Accessed: 10-Mar-2019]

[2] Mayo Clinic, “Sleep apnea - Diagnosis and treatment - Mayo Clinic” [Online]. Available: https://www.mayoclinic.org/diseases-conditions/sleep-apnea/diagnosis-treatment/drc-20377636. [Accessed: 10-Mar-2019]

[3] American Sleep Apnea Association, “Sleep Apnea Information for Clinicians,” *SleepApnea.org*, Jan. 2017 [Online]. Available: https://www.sleepapnea.org/learn/sleep-apnea-information-clinicians/

[4] World Health Organization, Ed., *Global surveillance, prevention and control of chronic respiratory diseases: A comprehensive approach.* Geneva: WHO, 2007.

[5] ResMed Inc., “Nearly 1 Billion People Worldwide Have Sleep Apnea, International Sleep Experts Estimate.” 21-May-2018 [Online]. Available: http://investors.resmed.com/investor-relations/events-and-presentations/press-releases/press-release-details/2018/Nearly-1-Billion-People-Worldwide-Have-Sleep-Apnea-International-Sleep-Experts-Estimate/default.aspx. [Accessed: 10-Mar-2019]

[6] D. Bohn, “Will Apple Watch sleep tracking in watchOS 7 Sherlock third party apps?” The Verge, 10-Mar-2020 [Online]. Available: https://www.theverge.com/tech/2020/3/10/21172386/apple-watch-sleep-tracking-sherlocking. [Accessed: 16-Apr-2020]

[7] S. P. Gjøby, “Extensible data acquisition tool for Android,” 2016 [Online]. Available: https://www.duo.uio.no/handle/10852/53004. [Accessed: 12-Mar-2019]

[8] S. H. Fairclough, “Fundamentals of physiological computing,” *Elsevier*, Oct. 2008 [Online]. Available: https://www.cs.tufts.edu/~jacob/250hci/Fairclough2009Fundamentalsofphysiologicalcomputing-annotated.pdf. [Accessed: 02-Oct-2019]

[9] C. Tardi, “Moore’s law,” *Investopedia.* 05-Sep-2019 [Online]. Available: https://www.investopedia.com/terms/m/mooreslaw.asp. [Accessed: 02-Oct-2019]

[10] S. Tibken, “CES 2019: Moore’s Law is dead, says Nvidia’s CEO - CNET,” 09-Jan-2019. [Online]. Available: https://www.cnet.com/news/moores-law-is-dead-nvidias-ceo-jensen-huang-says-at-ces-2019/. [Accessed: 16-Apr-2020]

[11] P. D. Bugajski, “Extensible data streams dispatching tool for Android,” 2017 [Online]. Available: https://www.duo.uio.no/handle/10852/60350. [Accessed: 12-Mar-2019]

[12] Shimmer, “Shimmer Java/Android API.” [Online]. Available: https://www.shimmersensing.com/products/shimmer-android-id. [Accessed: 31-Oct-2019]

[13] Garmin Ltd., “Fēnix® 5 - Multisport GPS Watch,” *Garmin.* [Online]. Available: https://buy.garmin.com/en-US/US/p/552982. [Accessed: 10-Mar-2019]

[14] Sport Tiedje GmbH., “Garmin Sport Watch Fenix 5X Sapphire.” [Online]. Available: https://www.sport-tiedje.co.uk/garmin-sport-watch-fenix-5x-sapphire-ga-010-01733-01. [Accessed: 31-Oct-2019]

[15] R. Llamas, J. Ubrani, and M. Shirer, “Worldwide Wearables Market Ticks Up 5.5% Due to Gains in Emerging Markets, Says IDC,” *Bloomberg*, 04-Sep-2018. [Online]. Available: https://www.bloomberg.com/press-releases/2018-09-04/worldwide-wearables-market-ticks-up-5-5-due-to-emerging-markets-says-idc. [Accessed: 10-Mar-2019]

[16] International Data Corporation, “IDC Reports Strong Growth in the Worldwide Wearables Market, Led by Holiday Shipments of Smartwatches, Wrist Bands, and Ear-Worn Devices,” *IDC: The premier global market intelligence company.* 09-Mar-2019 [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS44901819. [Accessed: 10-Mar-2019]

[17] Fitbit Inc., “Fitbit Ionic™ Watch.” [Online]. Available: https://www.fitbit.com/ionic. [Accessed: 10-Mar-2019]

[18] Fitbit Inc., “Press kit.” [Online]. Available: https://investor.fitbit.com/press/press-kit/default.aspx. [Accessed: 31-Oct-2019]

[19] C. Grugan, “Writing JS apps for Fitbit Ionic,” *JavaScript January.* Jan-2018 [Online]. Available: https://www.javascriptjanuary.com/blog/writing-js-apps-for-fitbit-ionic. [Accessed: 10-Mar-2019]

[20] “Max sampling rate of heart rate sensor of Ionic?” Aug-2017 [Online]. Available: https://community.fitbit.com/t5/SDK-Development/Max-sampling-rate-of-heart-rate-sensor-of-Ionic/m-p/2166062#M53. [Accessed: 10-Mar-2019]

[21] A. Pressman, “How Fitbit Plans to Boost Falling Sales,” *Fortune.* 27-Feb-2018 [Online]. Available: http://fortune.com/2018/02/27/fitbit-ionic-stock-price/. [Accessed: 10-Mar-2019]

[22] Apple Inc., “Apple Watch Series 4 - Technical Specifications.” 02-Apr-2020 [Online]. Available: https://support.apple.com/kb/SP778?locale=en_GB. [Accessed: 15-Apr-2020]

[23] Apple Inc., “Apple Watch Series 5 - Titanium Case.” [Online]. Available: https://www.apple.com/shop/buy-watch/apple-watch/titanium-case. [Accessed: 31-Oct-2019]

[24] S. Caldwell, “How to measure heart rate variability (HRV) on your Apple Watch,” *iMore.* 29-Apr-2018 [Online]. Available: https://www.imore.com/how-measure-heart-rate-variability-hrv-your-apple-watch. [Accessed: 27-Mar-2019]

[25] P. Cao, “IDC: Apple continues to lead wearable space, 10.4 million Apple Watches sold in Q4 2018,” *9to5Mac*. 05-Mar-2019 [Online]. Available: https://9to5mac.com/2019/03/05/idc-apple-watch-wearables-figures-q418/. [Accessed: 10-Mar-2019]

[26] A. Shcherbina *et al.*, “Accuracy in Wrist-Worn, Sensor-Based Measurements of Heart Rate and Energy Expenditure in a Diverse Cohort,” *Journal of Personalized Medicine*, vol. 7, no. 2, May 2017 [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5491979/. [Accessed: 10-Mar-2019]

[27] C. McGarry, “Who Has the Most Accurate Heart Rate Monitor?” *Tom’s Guide*. 11-Oct-2018 [Online]. Available: https://www.tomsguide.com/us/heart-rate-monitor,review-2885.html. [Accessed: 10-Mar-2019]

[28] Apple Inc., “How to use your Apple Watch.” [Online]. Available: https://support.apple.com/en-us/HT205552. [Accessed: 31-Oct-2019]

[29] P. Hegarty, “Overview of iOS.” Stanford University; University Lecture, 11-Nov-2017 [Online]. Available: https://www.youtube.com/watch?v=z9IXfYHhKYI&index=1&list=PL_l7vS8VbNDFBiKIL3fEQhkKXTYsncsvN. [Accessed: 12-Mar-2019]

[30] R. Barnes, “Apple iOS Architecture.” [Online]. Available: https://www.tutorialspoint.com/apple-ios-architecture. [Accessed: 31-Oct-2019]

[31] Apple Inc., “Core os layer.” [Online]. Available: https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CoreOSLayer/CoreOSLayer.html. [Accessed: 28-Oct-2019]

[32] A. M. Wood, “Cocoa and Cocoa Touch: How to Get Started Build Mac and iOS Apps,” 13-Feb-2018. [Online]. Available: https://www.whoishostingthis.com/resources/cocoa/. [Accessed: 08-Apr-2020]

[33] Apple Inc., “WKExtensionDelegate - WatchKit - Apple Developer Documentation.” [Online]. Available: https://developer.apple.com/documentation/watchkit/wkextensiondelegate. [Accessed: 10-Mar-2019]

[34] Apple Inc., “Managing Your App’s Life Cycle - Apple Developer Documentation.” [Online]. Available: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle. [Accessed: 22-Aug-2019]

[35] Apple Inc., “Managing Your App’s Life Cycle - Apple Developer Documentation,” 14-Apr-2020. [Online]. Available: https://developer.apple.com/documentation/uikit/app_and_environment/managing_your_app_s_life_cycle. [Accessed: 14-Apr-2020]

[36] Apple Inc., “Creating Independent watchOS Apps - Apple Developer Documentation,” 08-Apr-2020. [Online]. Available: https://developer.apple.com/

documentation/watchkit/creating_independent_watchos_apps. [Accessed: 08-Apr-2020]

[37] Apple Inc., "TransferFile - WCSession - Apple Developer Documentation," 15-Jan-2020. [Online]. Available: https://developer.apple.com/documentation/watchconnectivity/wcsession/1615667-transferfile. [Accessed: 15-Jan-2020]

[38] Open Planet Software, "I have recordings on my Apple Watch that have not transferred to my iPhone. – Product Support," 08-Jan-2018. [Online]. Available: https://openplanet.zendesk.com/hc/en-gb/articles/115004471413-I-have-recordings-on-my-Apple-Watch-that-have-not-transferred-to-my-iPhone-. [Accessed: 15-Jan-2020]

[39] Apple Inc., "Apple Watch Series 5 - Technical Specifications." 19-Sep-2019 [Online]. Available: https://support.apple.com/kb/SP808?locale=en_US. [Accessed: 31-Oct-2019]

[40] Apple Inc., "Energy Efficiency Guide for iOS Apps - Fundemental Concepts." 2018 [Online]. Available: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/FundamentalConcepts.html. [Accessed: 31-Oct-2019]

[41] Apple Inc., "Energy Efficiency Guide for iOS Apps: Energy Efficiency and the User Experience," 2018. [Online]. Available: https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/index.html. [Accessed: 14-Apr-2020]

[42] Apple Inc., "Human Interface Guidelines - Requesting Permission." [Online]. Available: https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/. [Accessed: 01-Nov-2019]

[43] Apple Inc., "The Basics — The Swift Programming Language (Swift 5.2)," 24-Mar-2020. [Online]. Available: https://docs.swift.org/swift-book/LanguageGuide/TheBasics.html. [Accessed: 16-Apr-2020]

[44] A. Inc., "Extensions — The Swift Programming Language (Swift 5.2)," 24-Mar-2020. [Online]. Available: https://docs.swift.org/swift-book/LanguageGuide/Extensions.html. [Accessed: 08-Apr-2020]

[45] R. Mejia, "Declarative and Imperative Programming using SwiftUI and UIKit," 08-Apr-2020. [Online]. Available: https://medium.com/flawless-app-stories/declarative-and-imperative-programming-using-swiftui-and-uikit-c91f1f104252. [Accessed: 08-Apr-2020]

[46] Apple Inc., "SwiftUI Tutorials - Apple Developer Documentation," 2019. [Online]. Available: https://developer.apple.com/tutorials/swiftui/. [Accessed: 08-Apr-2020]

[47] R. F. Tinder, *Relativistic Flight Mechanics and Space Travel: A Primer for Students, Engineers and Scientists*, 1st ed. Morgan; Claypool Publishers, 2006.

[48] E. Hsiao, "Introduction to Apple WatchKit with Core Motion — Tracking Jumping Jacks," 17-Feb-2018. [Online]. Available: https://heartbeat.fritz.ai/introduction-to-apple-watchkit-with-core-motion-tracking-jumping-jacks-259ee80d1210. [Accessed: 08-Apr-2020]

[49] C. McFadden, "Gyroscopes: What they are, how they work and why they are important," 04-Sep-2017. [Online]. Available: https://interestingengineering.com/gyroscopes-what-they-are-how-they-work-and-why-they-are-important. [Accessed: 08-Apr-2020]

[50] J. Su, "Apple Watch 4 Is Now An FDA Class 2 Medical Device: Detects Falls, Irregular Heart Rhythm," 14-Sep-2018. [Online]. Available: https://www.forbes.com/sites/jeanbaptiste/2018/09/14/apple-watch-4-is-now-an-fda-class-2-medical-device-detects-falls-irregular-heart-rhythm/. [Accessed: 08-Apr-2020]

[51] R. E. Berg, "Electromechanical transducer," 24-Oct-2018. [Online]. Available: https://www.britannica.com/technology/electromechanical-transducer. [Accessed: 16-Apr-2020]

[52] Apple Inc., "Measure noise levels with Apple Watch - Apple Support," 08-Apr-2020. [Online]. Available: https://support.apple.com/en-gb/guide/watch/apd00a43a9cb/watchos. [Accessed: 08-Apr-2020]

[53] G. Riches, R. Martinez, J. Maison, M. Klosterman, and M. Griffin, *Apple Watch for Developers Advice & Techniques from Five Top Professionals*. Apress, 2015.

[54] R. Metz, "Using Your Ear to Track Your Heart," 01-Aug-2014. [Online]. Available: https://www.technologyreview.com/2014/08/01/171915/using-your-ear-to-track-your-heart/. [Accessed: 08-Apr-2020]

[55] M. Chin, "Apple Watches may soon detect blood oxygen levels." The Verge, 08-Apr-2020 [Online]. Available: https://www.theverge.com/2020/3/9/21171483/apple-watch-ios-14-code-blood-oxygen-features-update. [Accessed: 08-Apr-2020]

[56] Apple Inc., "App Review - App Store - Apple Developer." [Online]. Available: https://developer.apple.com/app-store/review/. [Accessed: 08-Apr-2020]

[57] C. Szyperski, "Independently Extensible Systems - Software Engineering Potential and Challenges," in *In Proceedings of the 19th Australasian Computer Science Conference*, 1996.

[58] M. Swider, "Apple Watch battery life: how many hours does it last?" 24-Apr-2015. [Online]. Available: https://www.techradar.com/news/wearables/apple-watch-battery-life-how-many-hours-does-it-last-1291435. [Accessed: 06-Nov-2019]

[59] I. Buchmann, "Fast and ultra-fast chargers - battery university," 12-Apr-2019. [Online]. Available: https://batteryuniversity.com/learn/article/ultra_fast_chargers. [Accessed: 06-Apr-2020]

[60] J. Clarke, “How to Use a Morning Routine to Be More Productive,” 17-Mar-2020. [Online]. Available: https://www.verywellmind.com/morning-routine-4174576. [Accessed: 06-Apr-2020]

[61] E. Boersma, “Memory leak detection - How to find, eliminate, and avoid,” 09-Jan-2020. [Online]. Available: https://raygun.com/blog/memory-leak-detection/. [Accessed: 06-Apr-2020]

[62] “General Data Protection Regulation (GDPR) – Official Legal Text,” 04-May-2016. [Online]. Available: https://gdpr-info.eu/. [Accessed: 06-Apr-2020]

[63] J. Chen, “Stakeholder definition,” 04-Mar-2020. [Online]. Available: https://www.investopedia.com/terms/s/stakeholder.asp. [Accessed: 25-Apr-2020]

[64] P. A. Laplante, *What Every Engineer Should Know about Software Engineering*. Routledge, 2007 [Online]. Available: https://www.xarg.org/ref/a/0849372283/. [Accessed: 17-Feb-2020]

[65] Apple Inc., “Creating Independent watchOS Apps - Apple Developer Documentation,” 08-Feb-2020. [Online]. Available: https://developer.apple.com/documentation/watchkit/creating_independent_watchos_apps. [Accessed: 08-Feb-2020]

[66] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, “Extensible Markup Language (XML) 1.0,” 02-Oct-2017. [Online]. Available: https://www.w3.org/TR/1998/REC-xml-19980210. [Accessed: 06-Apr-2020]

[67] A. Seaborne, “SPARQL 1.1 Query Results CSV and TSV Formats,” 21-Mar-2013. [Online]. Available: https://www.w3.org/TR/sparql11-results-csv-tsv/. [Accessed: 08-Jan-2020]

[68] S. Tan and A. Shortlidge, “What’s New in HealthKit - WWDC 2015 - Videos - Apple Developer,” 2015. [Online]. Available: https://developer.apple.com/videos/play/wwdc2015/203/. [Accessed: 27-Mar-2020]

[69] E. S. Skarpsno, P. J. Mork, T. I. L. Nilsen, and A. Holtermann, “Sleep positions and nocturnal body movements based on free-living accelerometer recordings: association with demographics, lifestyle, and insomnia symptoms,” *Nat Sci Sleep*, vol. 9, pp. 267–275, 2017 [Online]. Available: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5677378/. [Accessed: 21-Mar-2020]

[70] D. Kyslenko, “Swift - Recording audio on WatchOS over AVAudioRecorder - Stack Overflow,” 21-Mar-2020. [Online]. Available: https://stackoverflow.com/questions/56407339/recording-audio-on-watchos-over-avaudiorecorder. [Accessed: 21-Mar-2020]

[71] Apple Inc., “CloudKit Database - CloudKit - Apple Developer Documentation,” 30-Jan-2020. [Online]. Available: https://developer.apple.com/documentation/cloudkitjs/cloudkit/database. [Accessed: 30-Jan-2020]

[72] Splunk Inc., “Splunk® Enterprise.” [Online]. Available: https://www.splunk.com/en_us/software/splunk-enterprise.html. [Accessed: 30-Jan-2020]

[73] O. Walch, Y. Huang, D. Forger, and C. Goldstein, “Sleep stage prediction with raw acceleration and photoplethysmography heart rate data derived from a consumer wearable device,” *Sleep*, vol. 42, no. 12, Aug. 2019 [Online]. Available: https://doi.org/10.1093/sleep/zsz180

[74] Apple Inc., “Running Workout Sessions - Apple Developer Documentation.” [Online]. Available: https://developer.apple.com/documentation/healthkit/workouts_and_activity_rings/running_workout_sessions?language=objc. [Accessed: 22-Mar-2020]

Appendices

A Cloning the project

The projects presented in this thesis are open source and can freely be cloned from GitHub. Both Sopor[33] and Virga[34] are available to download. This is done as shown in Listing 56.

Listing 56 Clone projects from GitHub

```
# Clone Sopor
git clone https://github.com/HaakonBakker/master_thesis_dev.git

# Clone Virga
git clone https://github.com/HaakonBakker/sopor-event-processor.git
```

B Creating a new Sensor class

A new sensor can be added to the project using the steps outlined in this part.

B.1 Add the file to the project

In Figure 38, three steps has to be taken to add the file to the project. Give the file the correct name (1), and make sure the WatchKit Extension is targeted (2) and add it to the correct directory (3).

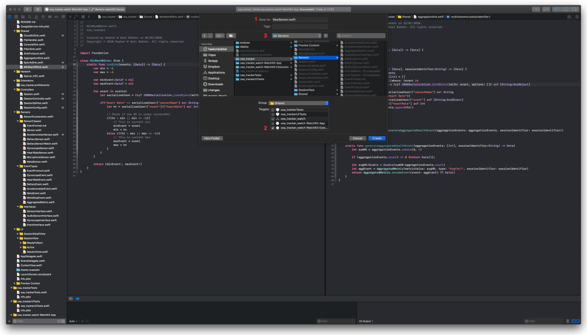


Figure 38: Adding the file to the project

[33]https://github.com/HaakonBakker/master_thesis_dev
[34]https://github.com/HaakonBakker/sopor-event-processor

B.2 Create the class

Creating the class and make it inherit from the base Sensor class is done as shown in Listing 57. From here methods from the base Sensor class[35] can be overridden. The following two methods have to be overridden: 1) the *startSensor()* method and 2) the *stopSensor()* method, these have also been added to Listing 57.

Listing 57 A NewSensor class

```
class NewSensor:Sensor{
    override func startSensor(session:Session) -> Bool{
        return true
    }

    override func stopSensor() -> Bool{
        return true
    }
}
```

B.3 Add the sensor to the configuration

First, you need to add the new Sensor class to the SensorEnumeration enum file, defined in `SensorEnumeration.swift`. Next, add the NewSensor class to the list of sensors defined in the `SensorConfiguration.swift`, as shown in Listing 58.

Listing 58 Adding the NewSensor to the sensorList

```
let sensorList =
    [GyroscopeSensor(sessionIdentifier: SESSION_UUID),
    MicrophoneSensor(sessionIdentifier: SESSION_UUID),
    BatterySensor(samplingRate: 5.0,
                  sessionIdentifier: SESSION_UUID),
    MetaSensor(sessionIdentifier: SESSION_UUID),
    NewSensor(sensorEnum: SensorEnumeration.NewSensor,
              sessionIdentifier: SESSION_UUID)]
```

B.4 Build and run

Finally, you have to build and run the project to take advantage of the newly created sensor.

C Creating a new Sink class

This is shown and described in detail Section 6.1.3.

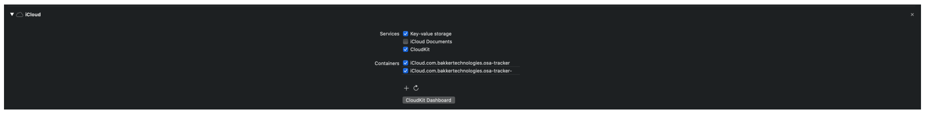
[35] Defined in the *Sensor.swift* file.

D Setting up CloudKit

To get CloudKit to work in the application there are a few steps that need to be taken.

D.1 Activate the iCloud capability

In Xcode, you have to activate the iCloud capability for the application. Next, the CloudKit service needs to be checked and the correct CloudKit containers need to be added see Section D.2. It should look something like Figure 39 depending on which version of Xcode is running.



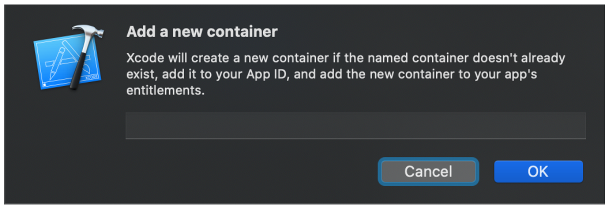
Services: ✓ Key-value storage; iCloud Documents; ✓ CloudKit

Containers: ✓ iCloud.com.bakkertechnologies.osa-tracker; ✓ iCloud.com.bakkertechnologies.osa-tracker-

Figure 39: Activate iCloud and CloudKit in Xcode

D.2 Create an CloudKit container

Also in Xcode, you have to create the containers for the application. This is done by hitting the "+" icon shown in Figure 39. Next, you need to fill in the name for the container in the dialog shown in Figure 40.



Add a new container

Xcode will create a new container if the named container doesn't already exist, add it to your App ID, and add the new container to your app's entitlements.

Figure 40: Creating a new container

D.3 Create the schema in CloudKit

By clicking the *CloudKit Dashboard* button shown in Figure 39 you will be taken to the CloudKit Dashboard where you will be asked to log in. Next, find the container you added in Section D.2. Click the "Schema" button shown in Figure 41.

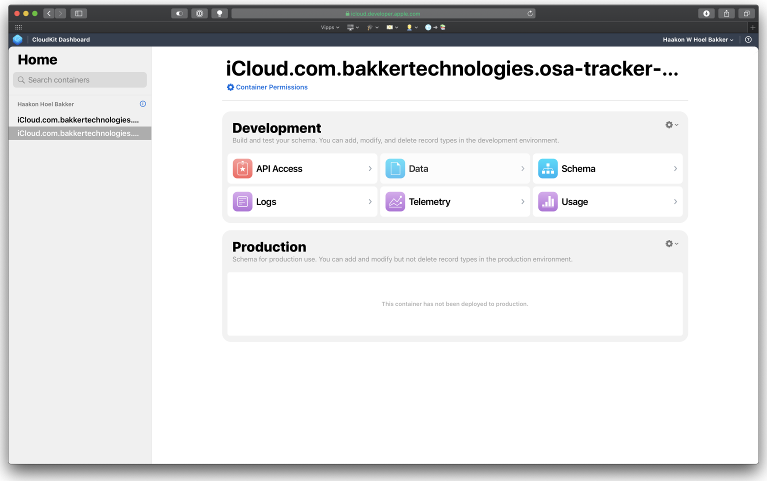
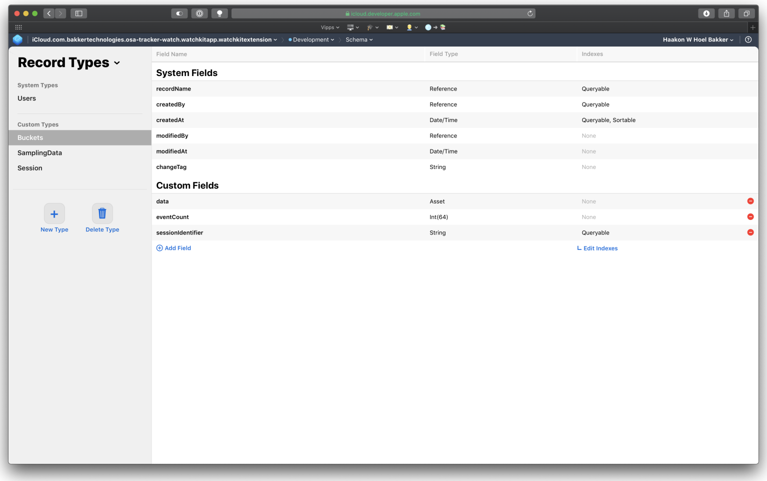

CloudKit Dashboard
Haakon W Hoel Bakker
Home
Search containers
Haakon Hoel Bakker
iCloud.com.bakkertechnologies....
iCloud.com.bakkertechnologies....
iCloud.com.bakkertechnologies.osa-tracker-...
Container Permissions
Development
Build and test your schema. You can add, modify, and delete record types in the development environment.
API Access
Data
Schema
Logs
Telemetry
Usage
Production
Schema for production use. You can add and modify but not delete record types in the production environment.
This container has not been deployed to production.

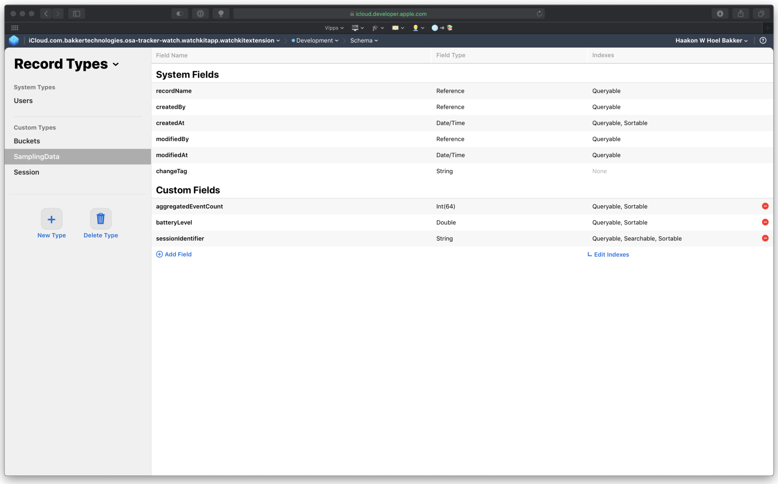
Figure 41: CloudKit overview

Next, a few record types need to be created. Start creating by clicking "New Type". The three record types need to be created: 1) "Buckets", 2) "Sampling-Data" and 3) "Session". These three records need to be configured as shown in Figure 42, Figure 43 and Figure 44. Make sure to set the indexes and the field types correctly.



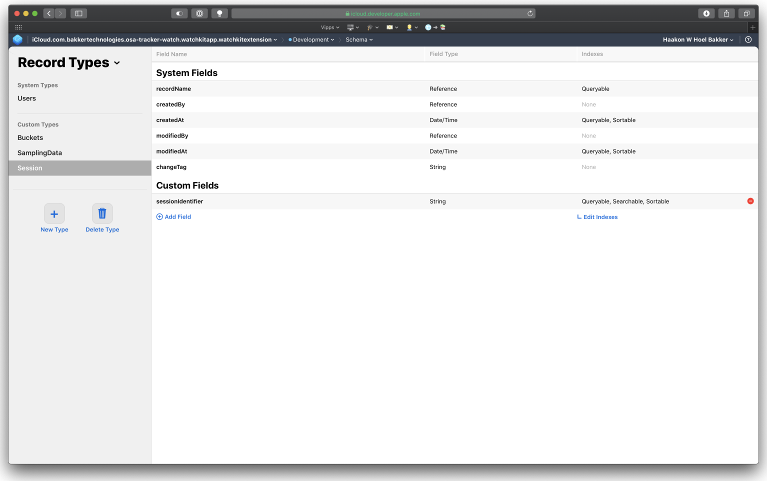
Field Name	Field Type	Indexes
System Fields		
recordName	Reference	Queryable
createdBy	Reference	Queryable
createdAt	Date/Time	Queryable, Sortable
modifiedBy	Reference	None
modifiedAt	Date/Time	None
changeTag	String	None
Custom Fields		
data	Asset	None
eventCount	Int(64)	None
sessionIdentifier	String	Queryable

Figure 42: Bucket record type



Field Name	Field Type	Indexes
System Fields		
recordName	Reference	Queryable
createdBy	Reference	Queryable
createdAt	Date/Time	Queryable, Sortable
modifiedBy	Reference	Queryable
modifiedAt	Date/Time	Queryable
changeTag	String	None
Custom Fields		
aggregatedEventCount	Int(64)	Queryable, Sortable
batteryLevel	Double	Queryable, Sortable
sessionIdentifier	String	Queryable, Searchable, Sortable

Figure 43: SamplingData record type

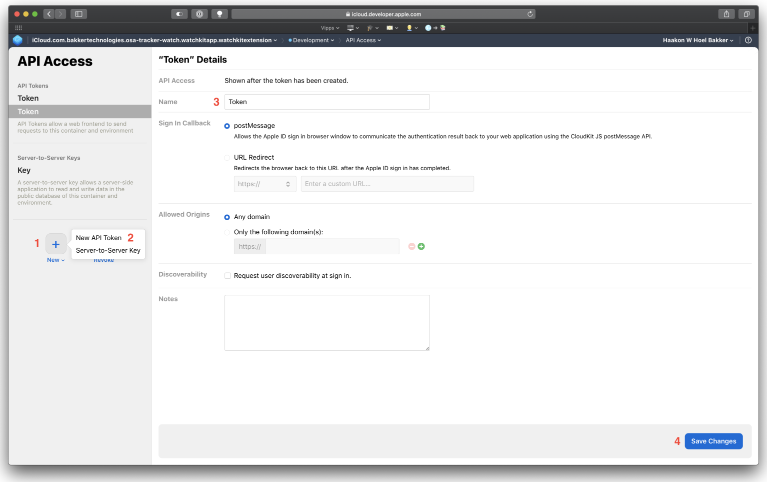


Field Name	Field Type	Indexes
System Fields		
recordName	Reference	Queryable
createdBy	Reference	None
createdAt	Date/Time	Queryable, Sortable
modifiedBy	Reference	None
modifiedAt	Date/Time	Queryable, Sortable
changeTag	String	None
Custom Fields		
sessionIdentifier	String	Queryable, Searchable, Sortable

Figure 44: Session record type

D.4 Getting the CloudKit token

At the main dashboard, shown in Figure 41, click the "API Access" button. Next, follow the steps outlined in Figure 45. (1) Click "New". (2) Click "New API Token". (3) Give the token a name. Leave the rest as is. Hit "Save Changes". Your API key will become visible, copy this and store securely.


icloud.developer.apple.com

iCloud.com.bakkertechnologies.osa-tracker-watch.watchkitapp.watchkitextension › Development › API Access

Haakon W Hoel Bakker

API Access

API Tokens
Token
Token
API Tokens allow a web frontend to send requests to this container and environment

Server-to-Server Keys
Key
A server-to-server key allows a server-side application to read and write data in the public database of this container and environment.

1 New · Revoke
New API Token 2
Server-to-Server Key

"Token" Details

API Access: Shown after the token has been created.

Name 3: Token

Sign In Callback:
[✓ postMessage]
Allows the Apple ID sign in browser window to communicate the authentication result back to your web application using the CloudKit JS postMessage API.
URL Redirect
Redirects the browser back to this URL after the Apple ID sign in has completed.
https:// · Enter a custom URL...

Allowed Origins:
[✓ Any domain]
Only the following domain(s):
https://

Discoverability: ☐ Request user discoverability at sign in.

Notes:

4 Save Changes

Figure 45: Steps for creating CloudKit token

D.5 CloudKit documentation

The documentation for getting CloudKit JS is available at Apple's website[36] as well as the CloudKit Web Services[37] webpage.

E Virga

E.1 Set up Virga

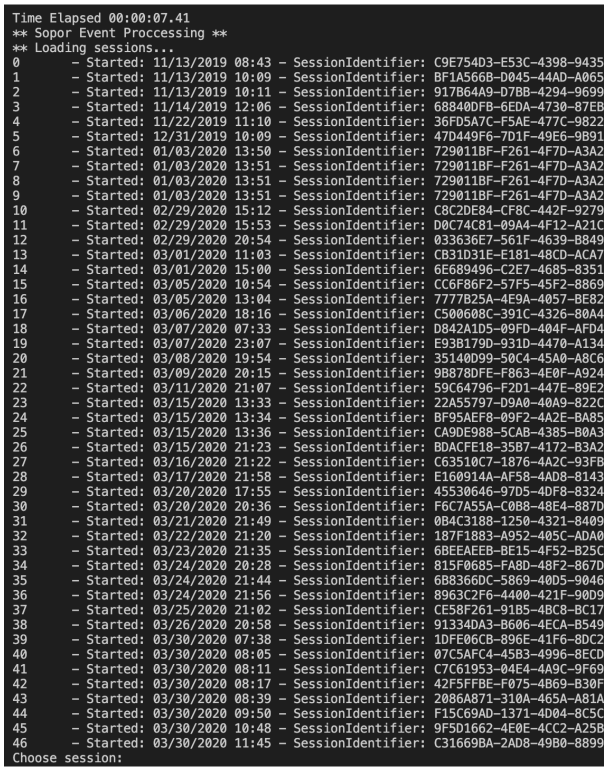
Virga requires the .NET framework and F# to be installed. When checked out from the GitHub repo, the steps showed in Listing 59 can be performed to run the project. Running those commands will result in an output like the one shown in Figure 46. You need to export the CloudKit token to the shell. Getting the token is shown in Section D.4. Setting the token as an environment variable allows the application to access the token without it being visible in code.

Listing 59 Running Virga from the terminal

```
cd virga/
export CLOUDKIT_TOKEN=YourCloudKitToken
dotnet build
dotnet run
```

[36]https://developer.apple.com/documentation/cloudkitjs

[37]https://developer.apple.com/library/archive/documentation/DataManagement/Conceptual/CloudKitWebServicesReference/index.html#//apple_ref/doc/uid/TP40015240



```
Time Elapsed 00:00:07.41
** Sopor Event Proccessing **
** Loading sessions...
0       - Started: 11/13/2019 08:43 - SessionIdentifier: C9E754D3-E53C-4398-9435
1       - Started: 11/13/2019 10:09 - SessionIdentifier: BF1A566B-D045-44AD-A065
2       - Started: 11/13/2019 10:11 - SessionIdentifier: 917B64A9-D7BB-4294-9699
3       - Started: 11/14/2019 12:06 - SessionIdentifier: 68840DFB-6EDA-4730-87EB
4       - Started: 11/22/2019 11:10 - SessionIdentifier: 36FD5A7C-F5AE-477C-9822
5       - Started: 12/31/2019 10:09 - SessionIdentifier: 47D449F6-7D1F-49E6-9B91
6       - Started: 01/03/2020 13:50 - SessionIdentifier: 729011BF-F261-4F7D-A3A2
7       - Started: 01/03/2020 13:51 - SessionIdentifier: 729011BF-F261-4F7D-A3A2
8       - Started: 01/03/2020 13:51 - SessionIdentifier: 729011BF-F261-4F7D-A3A2
9       - Started: 01/03/2020 13:51 - SessionIdentifier: 729011BF-F261-4F7D-A3A2
10      - Started: 02/29/2020 15:12 - SessionIdentifier: C8C2DE84-CF8C-442F-9279
11      - Started: 02/29/2020 15:53 - SessionIdentifier: D0C74C81-09A4-4F12-A21C
12      - Started: 02/29/2020 20:54 - SessionIdentifier: 033636E7-561F-4639-B849
13      - Started: 03/01/2020 11:03 - SessionIdentifier: CB31D31E-E181-48CD-ACA7
14      - Started: 03/01/2020 15:00 - SessionIdentifier: 6E689496-C2E7-4685-8351
15      - Started: 03/05/2020 10:54 - SessionIdentifier: CC6F86F2-57F5-45F2-8869
16      - Started: 03/05/2020 13:04 - SessionIdentifier: 7777B25A-4E9A-4057-BE82
17      - Started: 03/06/2020 18:16 - SessionIdentifier: C500608C-391C-4326-80A4
18      - Started: 03/07/2020 07:33 - SessionIdentifier: D842A1D5-09FD-404F-AFD4
19      - Started: 03/07/2020 23:07 - SessionIdentifier: E93B179D-931D-4470-A134
20      - Started: 03/08/2020 19:54 - SessionIdentifier: 35140D99-50C4-45A0-A8C6
21      - Started: 03/09/2020 20:15 - SessionIdentifier: 9B878DFE-F863-4E0F-A924
22      - Started: 03/11/2020 21:07 - SessionIdentifier: 59C64796-F2D1-447E-89E2
23      - Started: 03/15/2020 13:33 - SessionIdentifier: 22A55797-D9A0-40A9-822C
24      - Started: 03/15/2020 13:34 - SessionIdentifier: BF95AEF8-09F2-4A2E-BA85
25      - Started: 03/15/2020 13:36 - SessionIdentifier: CA9DE988-5CAB-4385-B0A3
26      - Started: 03/15/2020 21:23 - SessionIdentifier: BDACFE18-35B7-4172-B3A2
27      - Started: 03/16/2020 21:22 - SessionIdentifier: C63510C7-1876-4A2C-93FB
28      - Started: 03/17/2020 21:58 - SessionIdentifier: E160914A-AF58-4AD8-8143
29      - Started: 03/20/2020 17:55 - SessionIdentifier: 45530646-97D5-4DF8-8324
30      - Started: 03/20/2020 20:36 - SessionIdentifier: F6C7A55A-C0B8-48E4-887D
31      - Started: 03/21/2020 21:49 - SessionIdentifier: 0B4C3188-1250-4321-8409
32      - Started: 03/22/2020 21:20 - SessionIdentifier: 187F1883-A952-405C-ADA0
33      - Started: 03/23/2020 21:35 - SessionIdentifier: 6BEEAEEB-BE15-4F52-B25C
34      - Started: 03/24/2020 20:28 - SessionIdentifier: 815F0685-FA8D-48F2-867D
35      - Started: 03/24/2020 21:44 - SessionIdentifier: 6B8366DC-5869-40D5-9046
36      - Started: 03/24/2020 21:56 - SessionIdentifier: 8963C2F6-4400-421F-90D9
37      - Started: 03/25/2020 21:02 - SessionIdentifier: CE58F261-91B5-4BC8-BC17
38      - Started: 03/26/2020 20:58 - SessionIdentifier: 91334DA3-B606-4ECA-B549
39      - Started: 03/30/2020 07:38 - SessionIdentifier: 1DFE06CB-896E-41F6-8DC2
40      - Started: 03/30/2020 08:05 - SessionIdentifier: 07C5AFC4-45B3-4996-8ECD
41      - Started: 03/30/2020 08:11 - SessionIdentifier: C7C61953-04E4-4A9C-9F69
42      - Started: 03/30/2020 08:17 - SessionIdentifier: 42F5FFBE-F075-4B69-B30F
43      - Started: 03/30/2020 08:39 - SessionIdentifier: 2086A871-310A-465A-A81A
44      - Started: 03/30/2020 09:50 - SessionIdentifier: F15C69AD-1371-4D04-8C5C
45      - Started: 03/30/2020 10:48 - SessionIdentifier: 9F5D1662-4E0E-4CC2-A25B
46      - Started: 03/30/2020 11:45 - SessionIdentifier: C31669BA-2AD8-49B0-8899
Choose session:
```

Figure 46: Running Virga

E.2 SessionDomain record types

To deserialize the JSON from CloudKit, the module shown in Listing 60 is created. The `Event` type encapsulates the `EventData` type.

Listing 60 The Sensor event record type

```
module SensorDomain

type EventData = {
    x: double
    y: double
    z: double
    batteryLevel: double
    batteryState : int
    heartRate : double
    unit : string
}

type Event = {
    sessionIdentifier : string
    timestamp : string
    sensorName : string
    event: EventData
}
```

E.3 Body queries for the CloudKit API

This part describes the different HTTP request bodies used in the CloudKit API calls.

E.3.1 Body for Session records

This body shown in Listing 61 is used to query all `Session` records.

Listing 61 Session body

```
let sessionBody = """{
    "query": {
        "recordType": "Session",
        "sortBy": [
            {
                "systemFieldName": "createdTimestamp",
                "ascending": true
            }
        ]
    }
}"""
```

E.3.2 Body for full Bucket records

Listing 62 Bucket body with continuation marker

```
let fullBodyBucket sID contMarker =
    sprintf  """{
            "query": {
                "filterBy": [
                    {
                        "fieldName": "sessionIdentifier",
                        "comparator": "EQUALS",
                        "fieldValue": {
                            "value": "%s",
                            "type": "STRING"
                        }
                    }
                ],
                "recordType": "Buckets",
                "sortBy": [
                    {
                        "systemFieldName": "createdTimestamp",
                        "ascending": true
                    }
                ]
            },
            "continuationMarker": "%s"
        }""" sID contMarker
```

E.3.3 Body for Bucket records

Listing 63 Bucket body

```
let bodyBucketWithFilter sID =
    sprintf  """{
                    "query": {
                        "filterBy": [
                            {
                                "fieldName": "sessionIdentifier",
                                "comparator": "EQUALS",
                                "fieldValue": {
                                    "value": "%s",
                                    "type": "STRING"
                                }
                            }
                        ],
                        "recordType": "Buckets",
                        "sortBy": [
                            {
                                "systemFieldName": "createdTimestamp",
                                "ascending": true
                            }
                        ]
                    }
                }""" sID
```

E.3.4 Body for SamplingData records

Listing 64 Sample body with Session ID

```
let sampleBodyWithSessionID sID =
    sprintf """{
            "query": {
                "filterBy": [
                    {
                        "fieldName": "sessionIdentifier",
                        "comparator": "EQUALS",
                        "fieldValue": {
                            "value": "%s",
                            "type": "STRING"
                        }
                    }
                ],
                "recordType": "SamplingData",
                "sortBy": [
                    {
                        "systemFieldName": "createdTimestamp",
                        "ascending": true
                    }
                ]
            }
        }""" sID
```

E.4 Data types for the CloudKitProcessor.fs module

When serializing the JSON response `F#` needs to serialize to given data types to keep the strong typing paradigm. Unfortunately, a type cannot be defined directly inside a type record such as shown in Listing 65, all complex types have to be divided down to its simple types as shown in Listing 66 and Listing 67.

Listing 65 Types should be able to be defined this way

```
type Buckets =
    { bucket :
        {
            recordName: string
            recordType: string
            fields:
            {
                data:
                {
                    fileChecksum: string
                    size: int
                    downloadURL: string
                }
                eventCount: {value: int}
                sessionIdentifier: {value: string}
            }
        }
    }
```

Listing 66 How complex types actually are defined (1/2)

```
// Simple Value Records
type DataInt =
    { value: int }

type DataStr =
    { value: string }

type DataDouble =
    { value: double }

// Bucket Types:
type CKAssetValue =
    { fileChecksum: string
      size: int
      downloadURL: string }

type CKAsset =
    { value: CKAssetValue }

type Fields =
    { data: CKAsset
      eventCount: DataInt
      sessionIdentifier: DataStr }

type Bucket =
    { recordName: string
      recordType: string
      fields: Fields }

type CloudRecord =
    { records: Bucket list
      continuationMarker: string }
```

Listing 67 How complex types actually are defined (2/2)

```
// SamplingData Types:
type SamplingDataFields =
    { aggregatedEventCount: DataInt
      sessionIdentifier: DataStr
      batteryLevel: DataDouble }

type Created =
    { timestamp: string
      userRecordName: string
      deviceID: string }

type SamplingData =
    { recordName: string
      recordType: string
      fields: SamplingDataFields
      created: Created }

type SamplingDataRecord =
    { records: SamplingData list }

// Session
type SessionData =
    { recordName: string
      recordType: string
      fields: SamplingDataFields
      created: Created }

type SessionDataRecord =
    { records: SessionData list }
```

F Data from experiments

F.1 Experiment 1

F.1.1 Data-points

The data-points from every sleep session for experiment 1 are presented in Table 18 and Table 19.

Table 18: Experiments 1 results 1/2

Run	Start %	% after 8h	% Depleted	# events 8h	Date
1 - Take 1	100%	27%	73%	1700184	Mar 08-09, 20
1 - Take 2	100%	22%	78%	1702165	Mar 09-10, 20

Run	Start %	% after 8h	% Depleted	# events 8h	Date
1 - Take 3	100%	22%	78%	1698051	Mar 11-12, 20
2 - Take 1	100%	51%	49%	1686210	Mar 15-16, 20
2 - Take 2	100%	41%	59%	1698218	Mar 16-17, 20
2 - Take 3	100%	37%	63%	1701734	Mar 17-18, 20
3 - Take 1	100%	47%	53%	875413	Mar 21-22, 20
3 - Take 2	100%	46%	54%	868747	Mar 22-23, 20
3 - Take 3	100%	55%	45%	870293	Mar 23-24, 20
4 - Take 1	100%	58%	42%	871380	Mar 24-25, 20
4 - Take 2	100%	52%	48%	880367	Mar 25-26, 20
4 - Take 3	100%	55%	45%	876680	Mar 26-27, 20
Control - 1	100%	93%	7%	0	Mar 28-29, 20
Control - 2	100%	85%	15%	0	Mar 29-30, 20
Control - 3	100%	94%	6%	0	Mar 30-31, 20

Table 19: Experiments 1 results 2/2

Run	Avg Battery % Depleted	Max Battery % Depleted	Min Battery % Depleted	Standard deviation
1	76,33%	78%	73%	2,887
2	57,00%	63%	49%	5.888
3	50,66%	54%	45%	4,933
4	45,00%	48%	42%	3,000
Control	9,33%	15%	6%	4,028

F.1.2 Battery graphs

In table Table 20, Table 21, Table 22, Table 23 the collected battery graphs for the different runs are presented.

Table 20: Battery graphs for Run 1

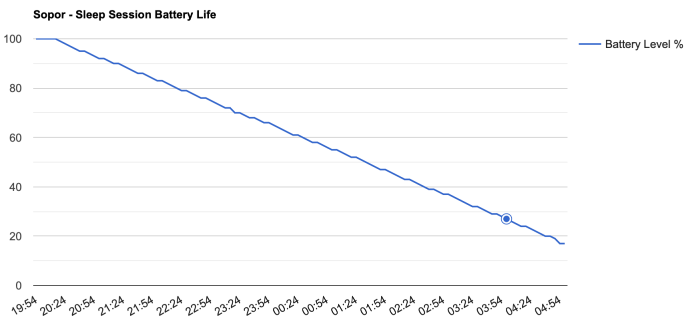
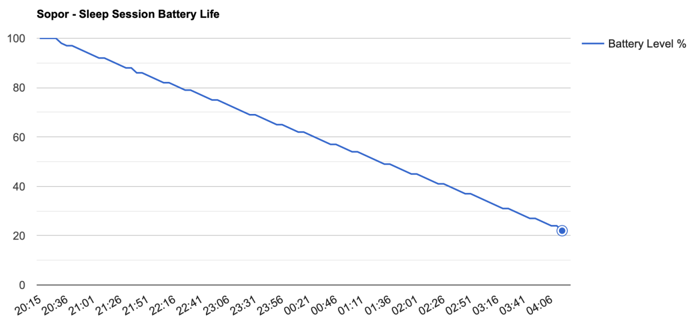
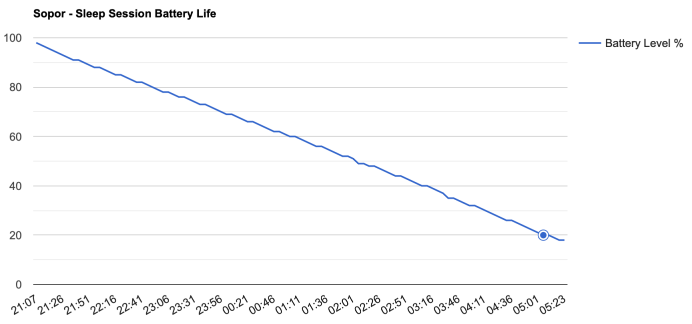
Take	Graph
1	
2	
3	

Table 21: Battery graphs for Run 2

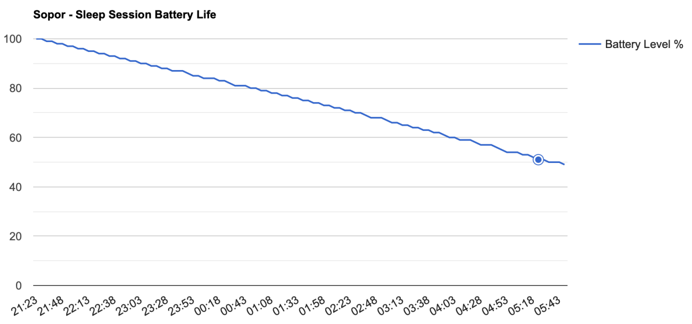
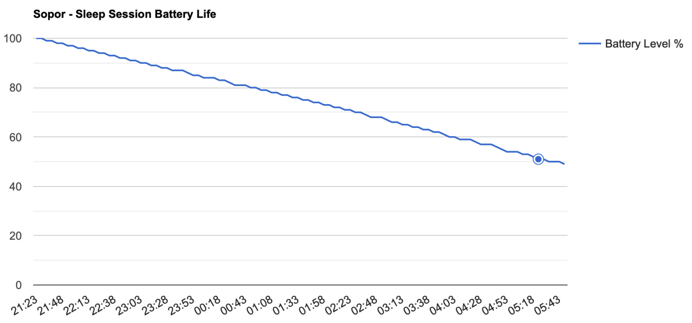
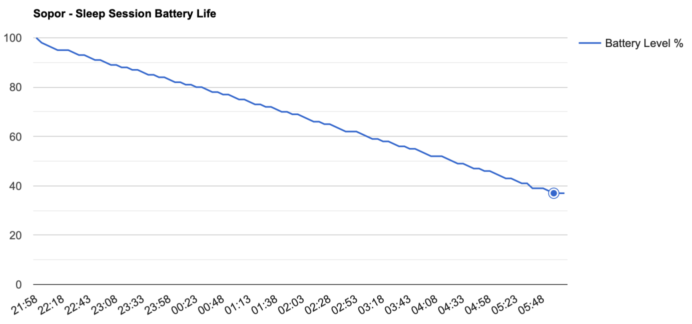
Take	Graph
1	
2	
3	

Table 22: Battery graphs for Run 3

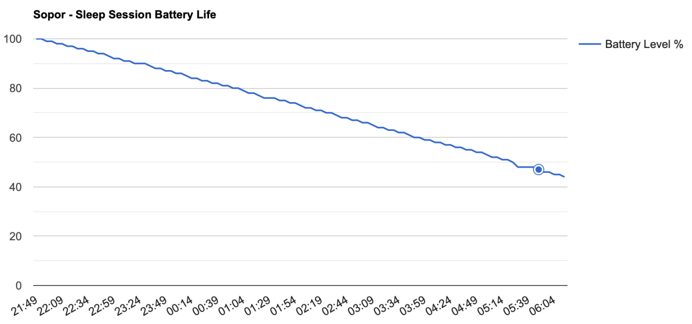
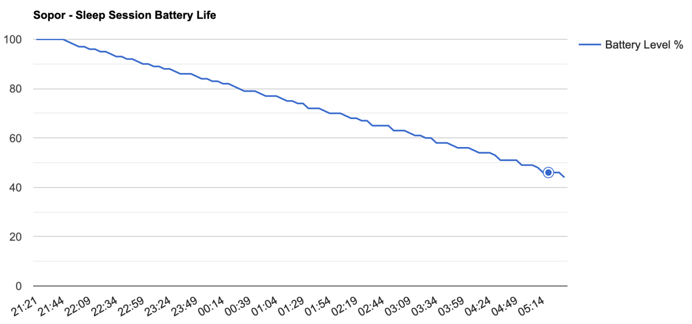
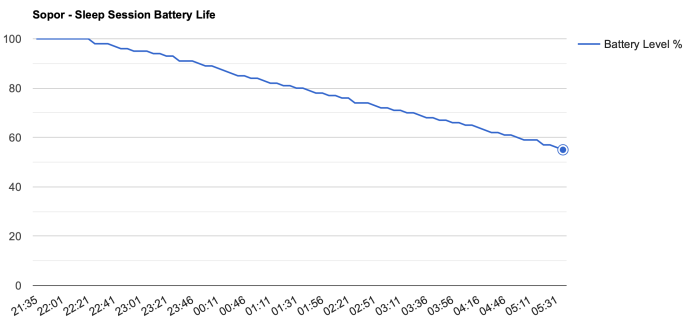
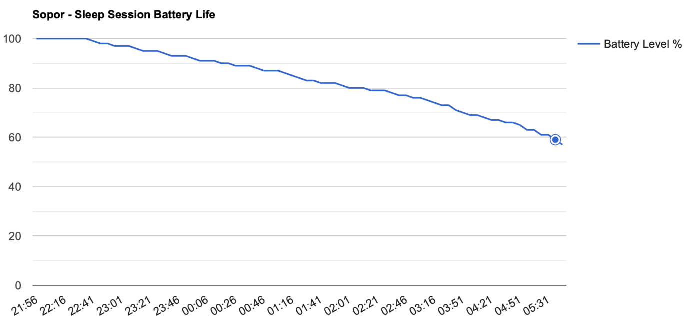
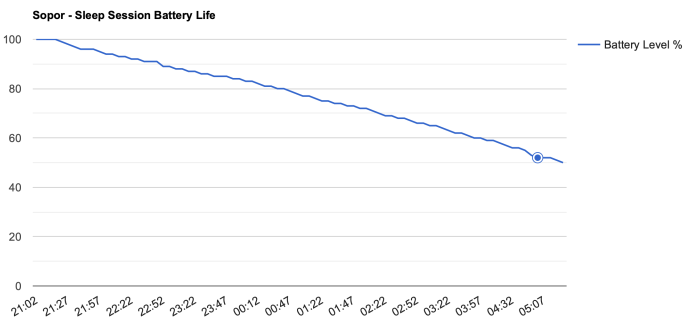
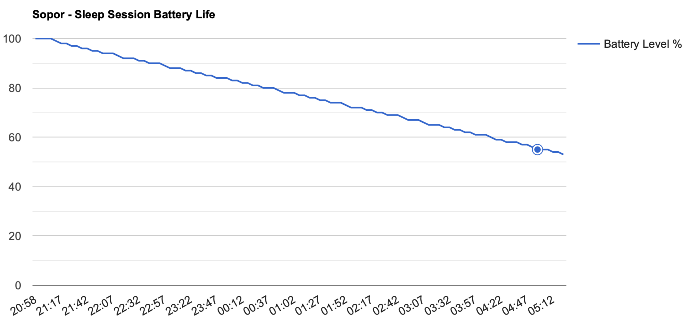
Take	Graph
1	
2	
3	

Table 23: Battery graphs for Run 4

Take	Graph
1	
2	
3	

F.2 Experiment 2

The results from experiment two are presented in Table 24.

Table 24: Results of Experiment 2

Run	Avg. CPU Usage	Max CPU usage	Max memory usage
1	3%	71%	18,4MB
2	4%	49%	17,9MB
3	4%	69%	17,8MB
AVG	3,667%	63%	18,03MB