An Android Wear Framework for Sensor Data and Network Interfaces

Bachelor Thesis

by

Bashkim Berzati

born in
Leverkusen

submitted to

Technology of Social Networks Lab
Jun.-Prof. Dr.-Ing. Kalman Graffi
Heinrich-Heine-Universität Düsseldorf

June 2017

Supervisor:
Andre Ippisch, M. Sc.

Abstract

Today there are many wearable devices on the market, which simplify our everyday life. This technology includes devices like activity trackers, smart glasses and smart watches. Mostly they are connected with our mobile phone, helping us to make communication in our public life easier and faster. Actual wearable devices feature an abundance of sensors and network interfaces, that mostly remain unchallenged due to the habits of the average user. Thus the question arises how to use this hidden potential, particularly with regard to improve the functionality and user experience of handset applications. Furthermore this research will try to explore the possibilities and limits current Wear devices face.

In this thesis we present a Framework for the Android operating system that focuses on sensor and network interfaces mainly. Therefore we created two applications, where one application is running on the mobile device while the second application is running on the wearable one. The communication between the devices is defined in the way, that the mobile application can request a service on the Wear application, while the wearable on the other hand is responsible for answering this request.

As a conclusion we can say that the wearable technology provides a huge amount of opportunities that can be accessed, since there were nearly no limits during the development. Due to the different types of services created for the Wear application we achieved the goal of providing a solid and wide base of information stored on the mobile phone, where the origin of the data is the corresponding Wear device. This broad knowledge base is urgently needed for decision making, especially if we want to influence the functionality and user experience of mobile applications as good as possible.

The recent update history regarding Android Wear reveals, that wearable devices are becoming more autonomous. This progress will ensure them an important position in the mobile communication world, thus working and integrating them into current projects is an interesting and future proof decision.

Acknowledgments

I would like to thank all people who supported me during this work.

A special thanks goes to my supervisor Andre Ippisch. He was a great support whenever I ran into a trouble spot or had a question about my research or writing.

Finally I would like to express my deepest gratitude to all people providing their work and knowledge for free.

Contents

List of Figures xi

List of Tables xiii

List of Listings xv

1 Introduction 1

1.1 Motivation . . . 1
1.2 Related Work . . . 1
1.3 Outline . . . 2

2 Fundamentals 3

2.1 The Android Operating System . . . 3
2.1.1 Linux Kernel . . . 3
2.1.2 Application Components . . . 4
2.1.3 Permissions . . . 5
2.1.4 Process and Thread . . . 5
2.2 The Android Wear Network . . . 6
2.2.1 Google Play Services . . . 7
2.2.2 WearableAPI . . . 7
2.2.3 Google Cloud-Node . . . 7
2.3 Installation . . . 9
2.3.1 Manual . . . 9
2.3.2 Companion Application . . . 10
2.4 Opportunistic Networks . . . 10

3 Demands and Design 11

3.1 Demands . . . 11

3.2 Design . . . 12
3.2.1 Wearable Application . . . 12
3.2.2 Mobile Application . . . 14

4 Implementation **15**
4.1 Development . . . 15
4.2 Google Wearable-API . . . 15
4.2.1 DataAPI . . . 16
4.2.2 DataItem . . . 17
4.2.3 Asset . . . 18
4.2.4 The *onDataChanged* - Method . . . 20
4.2.5 MessageAPI . . . 21
4.2.6 Connect to the Android Wearable Data Layer API . . . 23
4.3 Structure . . . 24
4.4 Library . . . 25
4.4.1 DataTransmitter . . . 25
4.4.2 SensorDataCollectorService . . . 25
4.4.3 BluetoothScanner . . . 27
4.4.4 WiFiScanner . . . 28
4.4.5 WiFiConnectionManager . . . 29
4.4.6 WiFiHotspotManager . . . 31
4.4.7 AudioRecorderManager . . . 32
4.4.8 LocationScanner . . . 34
4.4.9 ControlelementListener . . . 35
4.5 Wearable Application . . . 35
4.6 Mobile Application . . . 36

5 Evaluation **37**
5.1 Test Devices . . . 37
5.2 Network Performance . . . 38
5.3 CPU and Memory Usage . . . 39
5.4 Battery Life . . . 42

6 Conclusion and Future Work **45**
6.1 Conclusion . . . 45
6.2 Future Work . . . 46

Bibliography **49**

List of Figures

2.1 WearNetwork-Example 8
2.2 Overview WearableAPI 9

5.1 Synchronize 28 Byte Items 38
5.2 Synchronize 10028 Byte Items 39
5.3 CPU and Memory Usage of the Huawei Y3 40
5.4 CPU and Memory Usage of the Samsung S6 40
5.5 CPU and Memory Usage of the Sony SmartWatch3 41
5.6 CPU and Memory Usage of the Smartwatch with enlarged packages 41
5.7 Battery usage while synchronizing 42
5.8 Battery usage while not synchronizing 43

List of Tables

2.1 Permissions 5
2.2 Android Platform Versions 6
2.3 Wearable-API-Collection 8

4.1 Supported Sensors 26
4.2 Accuracy Status 26

5.1 Devices used for Testing 37
5.2 DataItems needed to answer a Request 44

List of Listings

4.1 Implementation of the DataAPI 16
4.2 Working with the Asset Structure 19
4.3 Implementation of the MessageAPI 21
4.4 Connecting to the Wearable Data Layer 23

Chapter 1

Introduction

1.1 Motivation

Today there are many wearable devices on the market, which simplify our everyday life. This technology includes devices like activity trackers (e.g Fitbit), smart glasses (e.g Google Glasses) and smartwatches (e.g Sony Smartwatch). Mostly they are connected with our smartphone, helping us to make communication in our public life easier and faster.
Actual wearable devices feature an abundance of sensors and network interfaces, that mostly remain unchallenged due to the habits of the average user, which is characterized by getting notifications of incoming messages or receiving weather news mainly. Thus the question arises how to use this hidden potential, particularly with regard to improve the functionality and user experience of handset applications. Furthermore this research will try to explore the possibilities and limits current Wear devices bring, to be able to form an opinion about their capabilities.

1.2 Related Work

Andre Ippisch presented an implementation of a Multilayer Framework for the Android operating system that uses the principle of Opportunistic Networking for local data exchange [Ipp15]. In the course of this thesis we will explain how the additional data received by the wearable device can improve the functionality of this Framework.

1.3 Outline

This chapter aims at clarifying the motivation to work with wearable devices and to implement applications that make use of the various sensor and network interfaces provided. The remainder of this thesis is structured as follows.

Chapter 2 will define and explain the fundamental knowledge that is necessary to understand this thesis. Section 2.1 will explain the basics on which the Android Operating System is build on and will discuss important components and characteristics that should be minded while developing Android applications. Section 2.2 is going to demonstrate the topology of the network our devices forge, show what is necessary to build this network and what possibilities will be provided for each device by entering it. In Section 2.3 we will guide you through the installation which is the first step for further developing. The last Section 2.4 will give a brief idea of the term Opportunistic Networks.

In Chapter 3 we will reveal the reason that brought the idea of developing this thesis and define demands in Section 3.1 that need to be met for future work. The following Section 3.2 will determine a design realizing all demands given.

Chapter 4 will explain the implementation in detail. Section 4.2 is going to illustrate how to take part in the network and determine the future task each device will fulfill by using the capabilities provided. Section 4.3 will demonstrate the structure of the framework build. Each of these structure components will then be intensively described in the following Sections.

Chapter 5 will take a look at the qualities of the established network and how the participation affects the devices, which are introduced in Section 5.1. Section 5.2 will explain the test environment build, that concentrates on the network parameters bandwidth and connections per second. Section 5.3 will monitor the applications CPU and Memory usage, while Section 5.4 protocols the battery status during the different test runs. At the end of this Chapter we will evaluate the result collected from the different tests created and summarize the most important facts that should be minded while working in the Android Wear Environment.

The last Chapter 6 will deliver a conclusion based on the results gained by developing this thesis. We will evaluate these results concerning our demands and motivation declared at the beginning of this research. Moreover we will discuss how the applications can be extended in the future.

Chapter 2

Fundamentals

In this Chapter we will present the fundamental knowledge, that is necessary to understand and develop this thesis. The first Section is about the Android Operating System, explaining the key elements needed for development. The second Section will define the term *The Android Wear Network*. It aims at giving an idea about how the structure, our mobile and Wear devices create, looks like. Furthermore it will explain which functionality the network provides using the Google Play Services. Section 2.3 will guide you through the installation that is necessary for developing. The last Section aims at giving you an idea about what Opportunistic Networks are.

2.1 The Android Operating System

This Section deals with the structure and features of the Android operating system. Furthermore we will describe the basic application components of Android. In the last part of this Section we will explain the permission system in Android and point out the differences between a thread and a process.

2.1.1 Linux Kernel

The Android operating system is designed for mobile devices by Google. Android is open source software based on the Linux Kernel resulting in a highly adaptable and portable oper-

ating system [M.Z17]. The Linux kernel contains the following important features.

1. **Process Management**
   The Linux Kernel is responsible for starting, executing and stopping the program. It is also responsible for the memory management of each process.
2. **Driver Model**
   The Linux Kernel provides an user friendly environment for developing drivers. This fact enables the users to run their software on the Android Operating System.
3. **File System Management**
   The Kernel fully manages the file system, which results in managing all data storage services done on the Android device.
4. **Network Stack**
   Linux Kernel coordinates all communicating with the network.
5. **Security**
   The Linux Kernel is responsible for the security of the system and each application. Each application runs as a Linux Process under permissions set by the system.

2.1.2 Application Components

There are fundamental components [Goo17c] for building an Android application. The once we used to develop this thesis are listed and explained in the following

1. **Activities**
   Activities are made to interact with the user. They usually represent a single screen with an user interface, that allows to display or request information from the user [Goo17b].
2. **Services**
   Android Service are background process without an user interface. They are designed to perform long-running operations or work for a remote process [Goo17s].
3. **Broadcast Receivers**
   Broadcast Receivers allow our application to listen for messages and events that are transmitted through the system. With defining a filter the user is able to listen for special events only.

4. **Intents**
   Intents are messaging objects that can be used to request actions from other app components. An other purpose of Intents is to start new Activities or Services. Broadcast Receiver are able to listen for Intents [Goo17j].

2.1.3 Permissions

Each application in Android runs as a Linux process in a sandbox environment isolated from other applications and system services. If the application needs access to certain system data or features, the developer needs to define permissions that allow to make use of them [Goo17n]. Certain permissions conclude the presence of particular hardware and features. If it uses a permission to access the microphone for recording voice, the running device should have one to guarantee that the application will run properly.
Permissions are classified into the groups Dangerous and Normal. Since Android 6.0 Dangerous Permissions have to be granted by the user during run time of the application [Goo17p]. Even if they are denied, the application can run with limited capabilities. Table 2.1 lists all permissions used by our application. To get a quick overview of the Android Platforms Versions Table 2.2 will help you [Goo17g].

Table 2.1: Permissions

Permission	Purpose	Classification
ACCESS_WIFI_STATE	Access the Wifi-Connectivity	Normal
CHANGE_WIFI_STATE	Ahange the Wifi-Connectivity	Normal
ACCESS_NETWORK_STATE	Access Networks in Common	Normal
ACCESS_COARSE_LOCATION	Access approximate Location	Dangerous
ACCESS_FINE_LOCATION	Access precise Location	Dangerous
ACCESS_RECORD_AUDIO	Access to Microphone to record Audio	Dangerous
ACCESS_BLUETOOTH_ADMIN	Discover and connect to Bluetooth Devices	Normal

2.1.4 Process and Thread

When we start an application on our device, the Linux Kernel starts a new process for it with a single thread of execution [Goo17o]. This thread is called the *Main-Thread*. You will also find the term *UI-Thread*, because all user interface operations are done in this initially created thread. By default each component of our application will be using the main thread

Table 2.2: Android Platform Versions

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.8 %
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.8 %
4.1 - 4.3	Jelly Bean	16-18	8.8 %
4.4	KitKat	19	18.1%
5.0 - 5.1	Lollipop	21-22	30.8 %
6.0	Marshmallow	23	31.2 %
7.0 - 7.1	Nougat	24-25	9.5 %

to execute its function. Due to the resource limitations of a thread in Android, developers are responsible to avoid blocking the main thread, by outsourcing CPU and memory intensive computations. Ignoring this will result in negative user experience like high reaction times of an application, or even worse an ANR-Dialog, while interacting with it.
Generally outsourcing methods can be divided into the two groups worker threads and worker processes. A worker thread is a new thread created within the current process. A popular example of background worker threads, Android offers, are *Async-Tasks*. The option of a worker process creates a new process, associated to the application. Managing to create a new process for a component of your application is done in the manifest file of the responsible application.

2.2 The Android Wear Network

The Android Wear Network describes the structure your handset and Wear devices forge. Each of these devices is a node and the communication link between these nodes are both WiFi and Bluetooth completing the network. In this Section we will show the elements needed for each node, to build our Wear network. Chapter 4.2 will provide more details and will also give code examples accomplishing the access and data exchange within the network.

2.2.1 Google Play Services

The Google Play Service is a background service made up of a client library and a collection of different APIs [Goo17m]. These can be used by application developers. Popular features of the client library are for example *Google Maps* and *Google+*. The Google Play Service is provided through the Google Play Store, thus updates do not depend on carrier or OEM system image updates. This allows Google Inc. to distribute updates fast to their users, helping developers to use the latest libraries and APIs for development.

One of the APIs brought by the Google Play Service is the *WearableAPI*. Each application using this API will enter the Wear Network, therefore this step is also called *Accessing the Wearable Data Layer*. The Play Service contains a class called *GoogleApiClient*, which is the main entry point for any of the Google Play Service APIs. The GoogleApiClient provides a builder, which creates an instance of the client with the desired API. Once the instance is successfully created within the application, the device will be connected to the existing Wear network.

2.2.2 WearableAPI

The WearableAPI is a collection of APIs, that can be used after we have created an instance of GoogleApiClient. Table 2.3 provides the names of the newly gained APIs and a short description of their action performed. By using these APIs in our application, we determine which role our device will take in the network.
The following Figure 2.1 shows a network of two wearable devices and a phone. Lets suppose that each of these devices uses the *DataAPI*. If we save a new contact on our address book and this type of information is approved to be shared, it will automatically appear on both of the users Wear devices. So being part of the network and using the right API will ensure getting the information that is synchronized.

2.2.3 Google Cloud-Node

As it can be seen in the above example, there is a node called *Cloud Node*. This node is a service hosted by Google to allow the users to use multiple Wear devices simultaneously. It

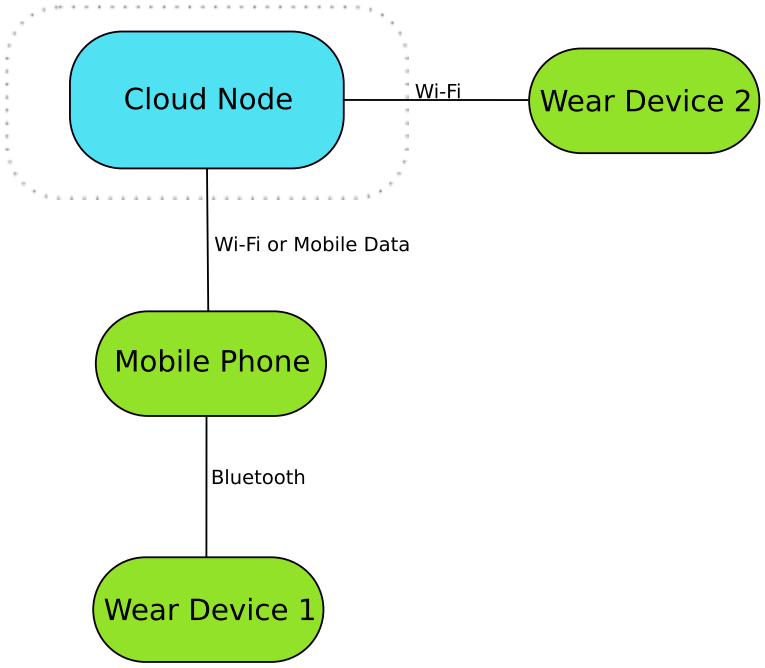

Figure 2.1: WearNetwork-Example

is created in regard to the application used and the phone bounded to our Wear devices, since wearable devices can be associated with a maximum of one device at a time. With entering the Wear Network, each device will establish a connection to this virtual node within our network. The precondition to use the benefits of the hosted Cloud Node is access to the internet. The cloud keeps track of each event within our network, like synchronization or disconnect events and will inform Wear devices not directly connected to the phone. It will also give report to the phone about any events happening on our Wear devices not connected via Bluetooth directly. This virtual node increases the size of our Wear Network from a small local one to a potential global one.

Table 2.3: Wearable-API-Collection

API-Type	Function
DataApi	API for nodes to read or write DataItems and Assets
MessageApi	API for nodes to send messages to other nodes
ChannelApi	API for nodes to send large data items
NodeApi	API to learn about local or connected nodes.
CapabilityApi	API to learn about capabilities provided by nodes on the Wear Network

Résumé Summing up we can say that the Wear Network is a network containing a phone and multiple Wear devices. Each network is characterized by the phone and the application accessing the data layer. The Wear device has to be paired to the phone once, to be associated with the specific network. By implementing the WearableAPI and the different APIs it contains, we can access the network and determine the task each device will fulfill in the future.

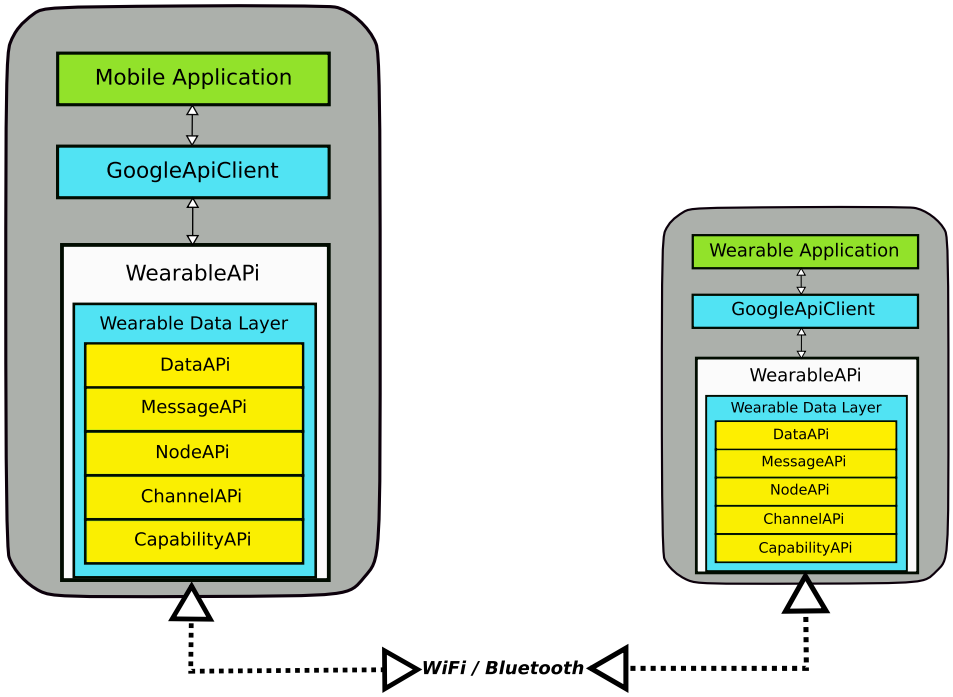

Figure 2.2: Overview WearableAPI

2.3 Installation

2.3.1 Manual

To access the functionality our Wear devices are providing, we need to pair our device to a phone. The first step to start data exchange between our phone and Wear devices, and more important, make use of the Wear functions, is to install the companion app called *Android Wear* on your phone. From this application we can request a Bluetooth paring between our devices. This will initiate a process of synchronization between the devices, transmitting

your address book to the wearable for example. After finishing the process we are good to go.

2.3.2 Companion Application

Attempts of avoiding the companion application were not successful. The WearableAPI is provided by the Google Play Service and at first it seemed to be unnecessary, because there is no need for a new application initiating the pairing process. This suspicion was supported by the fact, that we were able to pair and synchronize our devices without the companion app. But when it came to run our application on our phone, we could not connect to the WearableAPI. This behavior is caused by the GoogleApiClient. While creating an instance it checks whether the companion app is installed, resulting in failing if not. The reason the WearableAPI is providing our infrastructure for further communication forces us to install Android Wear on our phone.

2.4 Opportunistic Networks

Basic networks as we know them are mostly wired structures where a node is able to communicate with every other node connected to this network, no matter how big the distance between them is. The topology of this network is mostly static and hierarchical.

In contrast to this idea, there is a structure called Opportunistic Networks (OppNets). An Opportunistic Network is an ad-hoc network of wireless connected nodes. These nodes share data on opportunity based on a Peer-to-Peer (P2P) communication link. Since the communication range is limited to the amount of other nodes around you and the fact that any node can connect to a local one directly, the topology of the network is highly flexible. This also results in a possible lack of communication routes between nodes participating in this network, thus OppNets are characterized as delay tolerant.

Chapter 3

Demands and Design

In the first Section 3.1 of this Chapter we want to reveal the reasons and goals that led to developing this thesis. These objectives explained will shape the design of the application that will be discussed in Section 3.2. Since the result are two applications, one for the mobile device and the other for the Wear device, Section 3.2 is going to be divided into exactly these two components.

3.1 Demands

The background that motivated the development of this thesis is an application called Opptain designed by Andre Ippisch. This application builds an Opportunistic Network based on the possibilities Android offers. Installing this application will allow the device to enter this network and to start sharing data with other network participants without the need of an internet connection. Since wearable devices are designed to perform as an extension of the mobile phone in regard to improve functionality and the user experience, we questioned ourselves how wearable devices are able to bring these benefits to Opptain.

But before any improvements can be made, we needed to explore possibilities and limits actual Wear devices face. The target then was to collect as much information from the Wear device as possible and redirect it to the corresponding phone. The information collected mainly focused on network and sensor ones, but we also included control elements like events on the Wear touchscreen. The result should be two applications, one running on the phone

the other on the Wear device. The hierarchy between the two devices is split into master and slave, where the phone takes the rank of the master and each Wear device bound to this phone is assigned as a slave. The phone application should only request services on the Wear device. It should also work on devices running on a lower Android OS API and only need a few to no permissions. The Wear application on the other hand is responsible, for preparing and providing requested services and information to the mobile phone. This means that most part of the functionality is located at the Wear application.

Finally the mobile device should be holding a wide spectrum of information with the Wear device as the origin. It is then capable of analyzing and using gained data for increasing functionality and user experience.

3.2 Design

This Section will demonstrate a design that will fulfill the demands set by the Section above by listing structures that will be used for developing the both applications. Furthermore we will specify all services provided by the Wear application that can be requested through the mobile device and give examples on how this additional data can influence other applications like Opptain positively.

3.2.1 Wearable Application

The main task of this application is to collect information specified by the request of the mobile phone. This information collection needs to be transported reliably, thus we will use the DataAPI. To listen for incoming request events we will implement the MessageAPI. The idea of the Wear application is to provide a large amount of information. The type of information is listed in the following and can be divided into three main groups called *Network*, *Sensor* and *Control Element*.

1. **Network**
    a) Return nearby WiFi Networks
    b) Connect to the WiFi Network desired by the phone

c) Disconnect from the current WiFi Network

d) Open a Hotspot

e) Return current Network Status

f) Return available Bluetooth Hubs

g) Return bounded Bluetooth Hubs

h) Record Audio Sample and return it.

2. **Sensor**
   Each sensor shown in Table 4.1 can be requested by the phone. Additionally to these sensors the phone can also request the GPS sensor of the wearable for updates of the current location, if the device contains the necessary hardware.

3. **Control Element**
   The phone can activate a control element listener on the wearable, that will give report to the phone if there is any interaction with the touchscreen of the Wear device.

To demonstrate a possible positive effect of the main group Network, we will discuss an issue found in Opptain. Current mobile devices only have one WiFi module. Each device which is part of the network Opptain creates can either be a client searching for hotspots or be the hotspot itself to accept incoming client connections. Since there is only one WiFi module, the device can only be in one status at the same time. A possible scenario is, that our mobile device is currently transmitting a larger amount of data while being in the hotspot mode. In this time it is not possible for the phone to scan for other hotspots itself. This could led to miss a potential network member providing urgently needed data. But in case of having a wearable device, the task to scan can be forwarded while being in the hotspot status. The Wear device could inform the phone about this event, making it possible to reschedule the current transmitting and switch to client mode again. Since we also provide the function to establish a complete WiFi connection, our wearable could fetch the data itself and transmit it to the phone afterwards.

The possibilities brought by sensor and control element information is quiet similar. There might be an application, that asks the user to reboot the system for instance. This message could be forwarded to the wearable and answered there. Interacting with the touchscreen would imply a yes, shaking it would imply later. There are many examples like this, but they all affect the user experience in a good way, since we do not have to pull out the phone and unlock it for a simple input like in the above example needed.

You might have recognized that we put the recording of audio samples into the group *Network*. To avoid misunderstandings we will explain this a little more detailed. Voice or sound is a collection of different frequencies, that encode the message that needs to be transmitted. Since modern devices have speakers and microphones we can also provide this type of communication, but this time sound is the medium that transport the data between the clients in the network. You can find a detailed example about ultrasonic beacons in Section 4.4.7, treating this type of alternative communication.

3.2.2 Mobile Application

This application will be the control center of the whole project. Since it has to request services on the wearable application, it will have a user interface, unlike the Wear application. The UI will have two task, the first one is to provide interaction points where a user can decide what kind of request should be send. The second task is to display the results of the computed data, received from the Wear device. The request will be forwarded with the MessageAPI. Incoming data that answer trasmitted requests will be received with the help of the DataAPI.

Chapter 4

Implementation

In this Chapter we want to present the implementation of the Framework. In the first Section, we will name the environment, in which we implemented the application. In Section 4.2 we will take a closer look at the Wearable-API of Google, that is used to establish a communication channel between the handset device and the wearable device. Section 4.3 will list the components of which our project is made of to provide an overview of its structure. Sections 4.4, 4.5 and 4.6 will explain each of these components in detail.

4.1 Development

The development of the framework was accomplished with Android Studio, the official Integrated Development Environment (IDE) created by Google Inc. It is based on JetBrains' IntelliJ IDEA software and is designed for Android. Android Studio was always used in the most current version during the implementation, the last used one was 2.3.3.

4.2 Google Wearable-API

As explained in Subsection 2.2.1 the WearableAPI is part of Google Play services and provides a communication channel for different applications. Table 2.3 recaps the components

provided by the Wearable-API. For now we shall focus on the most used ones in our applications. After getting a deeper insight of these APIs, we will discuss what a *DataItem* is and how we can connect to the data layer to start sharing information between our devices.

4.2.1 DataAPI

The DataAPI is designed to synchronize user data with the nodes in the network [Goo17t]. Therefore it uses a construct called *DataItem*. Implementing the interface *DataListener* of the DataAPI, will allow us to listen for incoming events and more important, work with the data received, by overwriting the *onDataChanged* method. If we wish to send DataItems, we have to use the *putDataItem* method. The beneath code snippet will give you a short example on how to implement the DataApi properly. In this example, the network shares data regarding to different types of sensors. The sensor data is stored in float arrays, additionally there is a time stamp added, to determine how old the received data set is.

Listing 4.1: Implementation of the DataAPI

```
public class MyDataApiClass implements DataApi.DataListener{

  // Receiving DataItems
  @Override
   public void onDataChanged(DataEventBuffer dataEventBuffer) {
      for(DataEvent event : dataEventBuffer){
       DataItem dataItem = event.getDataItem();
       // Unpacking a received DataItem
       DataMap dataMap =
       DataMapItem.fromDataItem(dataItem).getDataMap();
       float[] values = dataMap.getFloatArray("VALUES_KEY");
       long timestamp = dataMap.getLong("TIMESTAMP");
      }
   }

   // Create and send a DataItem into the network
   public void createDataItem(float[] values, long timestamp){
     PutDataMapRequest data = PutDataMapRequest.create("PATH");
     data.getDataMap().putFloatArray("VALUES_KEY", values);
```

```
        data.getDataMap().putLong("TIMESTAMP", timestamp);
        PutDataRequest package = data.asPutDataRequest();
        Wearable.DataApi.putDataItem(googleApiClient, package);
    }
}
```

Setting DataItems while not connected to any nodes, will result in storing these packages. Stored items will be synchronized to other nodes in the network automatically, if the node goes online again.

4.2.2 DataItem

As mention above, DataItems are synchronized across all devices in the network. It is the base object of data storage in the Anroid Wear Network. They contain small blobs of data and associated *Assets*.
Each DataItem is identified by its Uniform Resource Identifier (URI), which contains the creator of the item and a path, which is defined be the developer. The URI follows this format: *wear://<nodeid></path>*.
The path must be unique in your application and start with a forward slash. The payload within this item is limited to 100KB. If we need to send data bigger than the given limit, we have to create an Asset attaching it to the payload of the package.
The standard procedure to create and send a DataItem is shown in the code snippet above. But instead of operating with DataItems itself, we use *PutDataMapRequests*. It is not necessary to work with this structure, but there are two reasons that strongly recommend using it. The first reason is that a PutDataMapRequest can be easily converted to a *PutDataRequest*, which was designed for the DataAPI and thus is supported by it. The second and most important reason is, that the PutDataMapRequest provides a collection of different put methods, like the *putFloatArray()* method in our example. These put methods allow us to manipulate data with Key-Value pairs easily. Furthermore using this will make retrieving information using the key and the correct get method easy and fast. The DataItem class does not support or contain any equal put/get methods, as provided by the PutDataMapRequest class. Using DataItems will force you to work with raw byte steams of your data you want to share in the network.

The following steps are needed to sync data in the Wear-Network:

1. Create a PutDataMapRequest instance. The parameter of this instance will be the path matching the criteria explained in this Subsection
2. Call the *PutDataMapRequest.getDataMap()* method to obtain a data map that you can set values on
3. Use the provided put methods fitting your type of information you want to store. The first parameter is going to be the key, the second the actual values
4. This step is optional but should be in your mind while developing. The method *setUrgent()* will give this package a higher priority ensuring the fastest delivery possible. Otherwise it can take up to 30 minutes to start the synchronization process
5. The package is now finished. This step will convert it to the PutDataRequest structure, which is needed for further transmitting. This is achieved by using the *PutDataMapRequest.asPutDataRequest()* method
6. The last step will start the synchronization process. The *DataApi.putDataItem()* will take care of this

As mentioned above, the DataAPI supports the PutDataRequest structure. The *putDataItem* method will take care of converting our PutDataRequest into a DataItem and send it to all nodes connected to the Android Wear Network.

4.2.3 Asset

The Asset class is a structure working with the DataItem class that provides a reliable method to synchronize data across the Wear network that exceeds the payload limit of 100KB. Since using the asset structure is not fully intuitive, we will describe the steps of creating, sending and receiving one in this Subsection in detail.
For creating an Asset, the data that needs to be shared in the network, must be provided as a byte array. If this is the case, we can simply create a new Asset with this byte array as the only parameter.
The next step is to send this asset. Therefore we need a data item, where the asset is attached on by using the provided method *putAsset()*. If this is done successfully, we can continue working with this data item as usual.

The last step to close the cycle, is to receive this asset. This is a little more difficult, since the payload of the asset remains at the data layer of the device that created it. This is very important, because as we have learned so far, if a part of your application implements the DataAPI, it will get any data item synchronized, even if it will not do any computation with it. This avoids the problem, that a very large asset is received by a device, to drop it immediately after transmitting finished. Furthermore we have more options to control receiving events of large data units. An example of this would be, that our device is normally interested in this asset, but since the battery level is low we reschedule the event. The question now to answer is, how can we access the data stored on the sending device?
If we get a data item known by its path to contain an Asset, we can get a reference to the providing data layer by using the method *getAsset()*. This reference is combined with the method *getFdForAsset()* provided by the DataAPI to open a file input stream to the desired data. The following example will demonstrate all the steps made shortly. It is about synchronizing audio samples in the Wear network.

Listing 4.2: Working with the Asset Structure

```
public class AssetTransmitter{

  public void sendDataItem(byte[] audioSample, long time){
    Asset asset = Asset.createFromBytes(audioSample);
    PutDataMapRequest data = PutDataMapRequest.create("PATHAUDIO");
    data.getDataMap().putAsset("KEY_AUDIO", asset);
    data.getDataMap().putLong("KEY_TIMESTAMP", time));
    PutDataRequest putDataRequest = data.asPutDataRequest();
    transmitter.sendData(putDataRequest);
  }
}

public class AssetReceiver implements DataApi.DataListener{

  @Override
  public void onDataChanged(DataEventBuffer dataEventBuffer) {
    for(DataEvent event : dataEventBuffer){
      DataItem dataItem = event.getDataItem();
      DataMap dataMap =
      DataMapItem.fromDataItem(dataItem).getDataMap();
```

```
        if(dataItem.getUri().getPath().equals("PATH_AUDIO")){
          Asset audioReferece = dataMap.getAsset("KEY_AUDIO");
          downloadAudioSample(audioReference);
        }
      }
    }

    public void downloadAudioSample(final Asset audioReference){
      new AsyncTask<Void,Void,Void>(){
        @Override
        proteced Void doInBackground(Void... params){
          InputStream assetInput =
          Wearable.DataApi.getFdForAsset(apiClient, audioReference)
          .await()
          .getInputStream();
          if(assetInput != null){
            byte[] audioSample = getByteArray(assertInput);
          }
        }
      }.execute();
    }
}
```

4.2.4 The *onDataChanged* - Method

During the development of this thesis, we ran into some issues with regard to the onDataChanged method. This Subsection will discuss what should be considered if the DataAPI listener is used.

As one can assume the method gets triggered only if the data really changed. Once we get a data item which is synced in our network, its will be compared with the actual data item located at the given path. If there is no change, this event will be simply dropped. We ran into this issue, while developing a control element listener. When the user interacted with the touch screen of the Wear device, we created a simple data item that consisted of only an integer value of 1 which should represent a touch event. As you can imagine, we only

received the very first of these items. All following touch events remained unrecognized. Implementing a counter that rose from 1 to 10 with each touch event still did not fix that problem, that means the data has to change significantly on the binary representation.
There are two ways to get rid of this problem:

1. If you want to send status updates only, like in our case that there has been a touch event you will suffer from the lack of information to trigger the listener. In this case it could be useful to implement the *MessageAPI* explained in the next Subsection. This API will not compare any incoming data, so dropping will not be a problem any more.

2. We can simply put a time stamp into our data item. A small period between two status events is enough for the time stamp to change its binary representation that much that it not dropped during the comparison. This is more a workaround and not a real fix for this issue, because there is still the possibility left that the period between two status updates might be to small or even zero, to change the binary representation clearly. This will result in dropping associated data items finally ending with a lack of information.

4.2.5 MessageAPI

The MessageAPI is used for remote procedure calls (RPC), or one-way request/response [Goo17q]. Each message consists of path, action and the *Node-ID* of the device, which should receive the current message created. This time a target is needed, because this API is not designed for a synchronization of data across the whole network. The path has the same format like the ones described for data items. The action parameter is normally a simple text that identifies and calls procedures or events on the receiving device.
The following code example shows how to set up a structure that makes it possible to listen and send messages in the Wear network.

Listing 4.3: Implementation of the MessageAPI

```
public class MessageReceiver implements MessageApi.MessageListener{

  @Override
  // For receiving incoming messages
  public void onMessageReceived(MessageEvent messageEvent) {
    String path = messageEvent.getPath();
```

```
      String action = new String(messageEvent.getData());
    }
}

public class MessageTransmitter extends Thread{

  public void sendMessage(String path, String action){
    NodeApi.GetConnectedNodesResult nodes =
    Wearable.NodeApi.getConnectedNodes(googleApiClient).await();
    for(Node node : nodes.getNodes()){
      MessageApi.SendMessageResult result =
      Wearable.MessageApi.sendMessage(googleApiClient,
      node.getId(),
      path,
      action.getBytes())
      .await();
      if(result.getStatus().isSuccess()){
        // Message was queued successfully
      }else{
        // Something went wrong
      }
    }
  }
}
```

Lets take a look at the *sendMessage()* method first. As described above, each message requires, apart from path and action, the desired node which shall receive the message. Therefore we use the *NodeAPI*, to get the corresponding NodeID. The NodeAPI was designed to request information about all connected nodes in our current network, providing a method called *getConnectedNodes()*. This request returns all connected devices, excluding the one starting the query. The example above send its message to all connected devices, and does not filter the result set containing these nodes anymore. One more thing to point out is, that the action needs to be a byte array. This function runs in an own thread, since the *await()* calls would completely block the main thread.
Receiving a message is straightforward. With the methods *getPath()* and *getData()* we can fetch the path and action of the incoming message event. The only thing we should mind

here is that the action is represented as a byte array and need to be converted to the desired data type before processing.
As maybe recognized already, the action does not have to be of the type string or any other simple data type, since when sending a message they actual payload is represented by a byte array. The action is also limited to a size of 100KB, but it does not provide any comfort functions like the *PutDataMapRequest()* of the DataAPI, so working with raw byte streams is necessary. An other reason is that there is no storage function and no guarantee that the message was delivered to the target. So instead of putting complex data structures into an unreliable transporting system, it is recommended to use the DataAPI with a unique path, which only gets scanned by the devices it belongs to.

4.2.6 Connect to the Android Wearable Data Layer API

Now that we know how to use the APIs presented, the last step to successfully transfer information is to build the infrastructure transporting our packages. Therefore the WearableAPI comes in handy. The Data Layer API is part of the Google Play Services, thus Google provides a builder class, that makes it easy to establish a connection to the Wearable Data Layer. This builder class is called GoogleApiClient [Goo17i]. Once connected, the API will take care of data transfer and Bluetooth negotiation between devices in the network. The following listing [Goo17a] will demonstrate, how to connect to the data layer.

Listing 4.4: Connecting to the Wearable Data Layer

```
GoogleApiClient googleApiClient = new GoogleApiClient.Builder(this)
      .addConnectionCallbacks(new ConnectionCallbacks() {
            @Override
            public void onConnected(Bundle connectionHint) {
              // Now you can use the Data Layer API
                Log.d(TAG, "onConnected: " + connectionHint);
            }
            @Override
            public void onConnectionSuspended(int cause) {
              // Something went wrong, check your Logcat output
                Log.d(TAG, "onConnectionSuspended: " + cause);
            }
      })
```

```
.addOnFailedListener(new OnConnectionFailedListener() {
      @Override
      public void onConnectionFailed(ConnectionResult res){
         Log.d(TAG, "onConnectionFailed: " + res);
      }
   })
.addApi(Wearable.API)
.build();
.connect();
```

Résumé At this point we gained the knowledge about the different APIs provided and which purpose they meet best. We have learned about the data structures particular APIs use and how each of these structures is build up. Furthermore we can create, send and build interfaces to receive them. Combined with the ability to connect to the Android Wearable Data Layer we complete the communication channel, that allows us the send and receive data from our wearable devices to our handset devices -and the other way around.

4.3 Structure

This project is made up of these three following components:

- **Library**
  This module contains all provided functionality that can be requested by the mobile application.
- **Wear Application**
  The Wear application listens for incoming service request by the corresponding mobile phone and is responsible to start the demanded service.
- **Mobile Application**
  The mobile application provides a graphical user interface (GUI), that allows the user to remote control the Wear devices connected and display the results of all desired services requested.

4.4 Library

The Library contains the classes providing the services which can be requested by the mobile device. In this Section we will take a detailed looked at each class explaining the way it works, the information we send and problems we faced during development.

4.4.1 DataTransmitter

The *DataTransmitter* class allows us to send a PutDataRequest to the Wearable Data Layer, which will be first converted to a DataItem and then synchronized with all other clients in the wearable network. It also provides a callback function, that reports the delivery status of the DataItem send.
Therefore an instance of the DataTransmitter needs a PutDataRequest containing all information. This PutDataRequest is provided by the classes who intend to synchronize computed data with the network.
Also an established connection to the data layer is needed. How we connect to the Android Wearable Data Layer API is explained in Section 4.4. The main Activity of each application takes care of connecting to the data layer and provides a reference to this communication channel, that is finally used by the DataTransmitter instances.

4.4.2 SensorDataCollectorService

The *SensorDataCollectorService* enables us to listen for sensor events. Our service creates an instance of the *SensorManager*, which is connected to the sensor system service of Android [Goo17r]. We can determine the sensor we want to keep an eye on, by registering them to the SensorManager. To receive updates on desired sensors, two requirements must be met. First our sensor must be available on the device. This is simply done by requesting the desired sensor, via *getDefaultSensor(sensorID)*. If an instance is returned, we know that our device is capable of giving us updates regarding to the requested sensor. Second the sensor has to be approved by the hand held device. When receiving the message to activate the service, the message action also contains a string of sensor ids the user wants to track. This string is build in the following way: *[SensorNo1], [SensorNo2], ... ,[SensorNoX].*

Table 4.1: Supported Sensors

Supported Sensor Type	Sensor.ID
Accelerometer	TYPE_ACCELEROMETER
Temperature	TYPE_AMBIENT_TEMPERATURE
Gravity	TYPE_GRAVITY
Gyroscope	TYPE_GYROSCOPE
Light	TYPE_LIGHT
Acceleration	TYPE_LINEAR_ACCELERATION
Magnetic Field	TYPE_MAGNETIC_FIELD
Pressure	TYPE_PRESSURE
Proximity	TYPE_PROXIMITY
Humidity	TYPE_RELATIVE_HUMIDITY
Rotation	TYPE_ROTATION_VECTOR
Game Roatation	TYPE_GAME_ROTATION_VECTOR
Step Counter	TYPE_STEP_COUNTER
Heart Rate	TYPE_HEART_RATE

When registering a sensor to the SensorManager, this string is checked for occurrence. If both conditions are met, it will be activated for future updates. This procedure is done for each sensor provided in this thesis. Table 4.1 gives an overview of all supported ones. Since our service implements the *SensorEventListener* interface, the *onSensorChanged* method gets called, if one of our tracked sensors signals a change. This method then creates a *PutDataRequest*, storing a float array containing the values received by the corresponding sensor, a time stamp and an integer value called accuracy in it. This accuracy value reports, how reliable the data received is. The finished PutDataRequest gets forwarded to the DataTransmitter.
The SensorDataCollectorService runs in an own process. Since there are devices with a huge amount of sensors that could possibly be requested at the same time, lots of traffic is caused. In regard to future devices, it is likely that the number of integrated sensors will increase. Thus running this service in an own process is a future-proof option.

Table 4.2: Accuracy Status

Accuracy	Status
3	High accuracy
2	Medium accuracy
1	Low accuracy
0	Unreliable accuracy
-1	No contact to sensor

4.4.3 BluetoothScanner

The *BluetoothScanner* is responsible for returning bounded and available Bluetooth hubs surrounding the Wear device. The difference between bounded and available is, that a bounded device was already connected to our Wear device before and thus the devices contain security information to initiate a new connection without sharing keys again. The BluetoothScanner creates an instance of the *BluetoothAdapter*, that is connected to all system services provided by Android regarding Bluetooth connectivity [Goo17e]. The process to inform the handset device of bounded Bluetooth hubs near is divided into two steps.

First there are references to the bounded Bluetooth devices stored, that can be requested with the *getBondDevices()* method provided by the BluetoothAdapter instance. This will return a set of the type *BluetoothDevice*. While iterating through this set, we retrieve the name and the MAC address of the device storing these values in a hash map temporally. This method provides a possibility to work with the gained data before sending them to the handset device. The next step is about to prepare the PutDataRequest containing all information needed for the handset device. Therefore the hash map will be transferred into an array list, where the first entry is the name of the device and following entry is the MAC address according to this name. Additionally to this information, we also put an integer into this data item. This integer indicates if devices were found or not. 1 signals that at least one device was found and 0 indicates that no devices were found. We also put a time stamp into this request, to have the information about the exact time the storage for bound devices was read. The finished PutDataRequest get forwarded to the DataTransmitter.

If we want to send data according to devices available around our wearable, there is one additional step to make. Since this time we do not have stored information about the desired hubs, we need to request a scan with our BluetoothAdapter instance. The method targeting the system service to start a scan is called *startDiscovery()*. This method only returns true, if the call was successful. To keep track of what is happening during this scan, we have to set up a BroadcastReceiver, listening to the following actions:

1. **BluetoothDevice.ACTION_DISCOVERY_STARTED**
   This indicates that the system service has successfully started, it is useful to listen for this event regarding debug purposes and for a consistent user feedback.
2. **BluetoothDevice.ACTION_FOUND**

This event signals that a new device was found. The Intent send contains all information about the new device found. We can access them by using *getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)* on the Intent received, which returns a *BluetoothDevice* object. By using the methods *getName()* and *getAddress()* we can fill our hash table properly.

3. **BluetoothDevice.ACTION_DISCOVERY_FINISHED**
   This Intent shows that the scan process finished. We use this event to close our BroadcastReceiver and start the process to send the collected data to the phone.

The BluetoothScanner class sends two different packages into the network. One containing the information about bounded devices the other package about available ones.

4.4.4 WiFiScanner

The task of the *WiFiScanner* is to give a report about all available wireless networks around the wearable device if demanded. Therefore we create an instance of the *WifiManager*, that allows us to access the Android system services provided, concerning the W-LAN connectivity status of our wearable device [Goo17u].

The WifiManager instance provides a list, that contains all elements found during the last scan. These results are saved as *ScanResult* objects and contain fields witch inform us about the SSID for instance. We can get a copy of this list by using the *getScanResults()* method. The SSID (Service Set Identifier) represents the name, on which the related network is displayed during Wi-Fi scans. ScanResult objects provide more than the SSID only, a few examples of additional information is the BSSID or capabilities of the network. In this thesis we are interested in giving the phone the information about all surrounding SSIDs and related BSSIDs. Therefore we create two arrays. The first one will store all SSIDs, the second will save all BSSIDs. The position of the SSID array will match with the BSSID one, this means that for example the first entry of both arrays belong to the same network. The PutDataRequest, which will be synchronized with the network will contain both of these arrays. Additionally we put an integer and a time stamp into this data item. The integer value indicates if any WiFi networks were found during the last scan. 1 signals that at least one network was found and 0 zero indicates that no network was found.

But there is still the possibility, that the list with scan results is empty. Therefore the NetworkScanner provides the option to initiate a scan, via the *startScan()* method given by the WifiManager instance. Since this method uses a system service only, it will return true if the demand was successfully made. Luckily the system service responsible for the scan will broadcast a *SCAN_RESULTS_AVAILABLE_ACTION* through the system, telling that a WiFi scan was finished. To recognize this event, the NetworkScanner contains a BroadcastReceiver listening for this type of Intent only. If we receive this Intent, we can proceed with the *getScanResults()* method as described before.
One problem is, that the results of a scan can vary within seconds, caused by a location change of the user or simply a shut down of a network. This is the reason why the implementation of this thesis always runs a scan first, before sending the gained data to the demanding phone, no matter how old the last scan result was.

While developing this class we faced a strange difficulty. Initiating a scan for available networks did not work on devices running on API 23 and higher, as long as a permission for location determination was missing [Goo17v]. Some developers reported, that GPS had to be also active, to receive updates. This issue is know to Google and has been classified as a bug, that should be removed in future builds. Since we do not have any other test devices, we can not tell if there is any change regarding this issue.

4.4.5 WiFiConnectionManager

The *WiFiConnectionManager* was designed to manage all connectivity issues of the wearable device. It allows to connect to a local WiFi network, disconnect from currently connected ones and provides a function that returns the connection status of the Wear device.
In order to establish a connection to a local network, we need to define a configuration and activate it. Android provides a class called *WifiConfiguration*, which will be filled with the necessary information in the following.

The first step is to set the desired SSID and pass phrase by accessing a provided fields *SSID* and *preSharedKey* of our WifiConfiguration. It is important that both constants are within quotation marks, otherwise the configuration can not be build up correctly. The next step is to define which protocols, group ciphers and pairwise ciphers should be supported. In this thesis we chose the protocols to be RSN and WPA. Supported ciphers are TKIP and CCMP.

At this points we finished the configuration, yet we need to insert and activate it somehow. This is done by the *WifiManager* class provided by Android. As we have learned in Section 4.4.4 an instance of this class will allow us to access W-LAN connectivity status of our wearable device, but it also able to manipulate this status directly.
To insert this configuration to the internal configuration storage of our system, we use the *addNetwork(wifiConfiguration)* method with our configuration as the only parameter. This method returns an integer value, that represents the identifier internally given to this configuration. Thus it is essential to save this ID for the purpose of activating it. The last step is to use the *enableNetwork(wifiConfigurationID, true)* method. The first parameter represents the internally decided ID, the second one indicates that this configuration should be used for further processing.

If we want to disconnect from the current network, we need to delete the related configuration. During the development of this thesis we tried to use the provided WifiManager method *disconnect()* and to manually disable the configuration with *enableNetwork(wifiConfigurationID, false)* without reaching the attended goal of a complete disconnect, only deleting the configuration did. Using the two mentioned attempts resulted in a short disconnect from the network followed by an automatically initiated reconnect event.
To successfully delete a configuration, we need to make use of a WifiManager instance again. The WifiManager class has a function called *getConfiguredNetworks()* returning a list of all configuration. While iterating within this list, we can determine the SSID and the ID that was given to the configuration, by entering the fields *SSID* and *networkId*. If the SSID matches with the one we want to disconnect, we use the method *removeNetwork(networkID)*.

The last function provided is to check the current connection status of our wearable device. Again we will use a WifiManager instance for this purpose. This time we will use the *getConnectionInfo()* method, which returns a *WifiInfo* instance representing the current WiFi connection status. The WifiInfo structure contains several methods to retrieve information about the current connection, the one that we are interested in are the SSID and the IP address of the wearable device. This information can be fetched with the help of the *getSSID()* and *getIpAddress()* methods. But before we create a PutDataRequest with all the relevant information, one more check is done. For this step we use the *ConnectivityManager*. A reference to this class grants us to enter all network connections provided on the Wear device. This system service is not limited the WiFi interface, like the WifiManager. Since our device is connected to one network at the same time only, the request *getNetworkInfo(TYPE_WIFI)* will give us further information about the current WiFi connection, wrapped into a class

called *NetworkInfo*. This structure enables us to check whether network connectivity exists and it is possible to establish connections and pass data [Goo17f]. This check is done with the *isConnected()* method. The WifiManager is not capable of determining, whether the connection is able to pass data or not.

If the *isConnected()* call returns true, we create a PutDataRequest containing the SSID, IP address and a time stamp of the current network status. In case of *isConnected()* returns false, the SSID and the IP address are filled with the constants -1, to indicate that there is no active connection or the current connection is not able to pass data.

4.4.6 WiFiHotspotManager

During the development of this theses, we tried to design a class that is able to turn our wearable device into a hotspot. Unfortunately this approach failed, due to the fact that the we could not get the permission *WRITE_SETTINGS*, that is necessary. This is caused by the absence of the Permission Management Screen, that is not available under Android Wear.
The permission *WRITE_SETTINGS* is categorized as a dangerous permission and for that needs to be requested and authorized by the user during the run time of the application. Normally this is done with the *requestPermission()* call, but in case of getting this permission this approach will not work. The following steps are necessary to call the Permission Management Screen and enable it manually.

1. **Declaration in the Manifest**
   This step is always needed, but listed for the purpose of completeness
2. **Check Application Permission Status**
   There is a function called *canWrite()* provided by the Android system since API 23. This call returns true if the permission is granted, otherwise it will return false. In case of getting the return value false, the next step is needed.
3. **Open Permission Management Screen**
   To open the permission management screen for your application, we have to build a special Intent first. The first part of building is to give the Intent the right action. This is done by the method *setAction(Settings.ACTION_MANAGE_WRITE_SETTINGS*). This signals that we want the *WRITE_SETTINGS*. The second and last part is to announce which application needs this permission. This is achieved by the Intent method *set-*

Data("package:" + getPackageName())). The Intent is now correctly build. To send it to the system, we simple start it with the call *startActivity(intent)* with our Intent as the only parameter.

Following these steps will open the Permission Monitor, where the user has to enable this permission by checking a switch in the UI for example. But during the development with Android Wear we got an error reporting about, that the system service who handles this Intent is not found. Further research revealed that there are several tickets with the same problem found on the Google Issue Tracker [Goo17l]. Google is aware of this problem, but does not provide a fix for it.

4.4.7 AudioRecorderManager

Before we start going into detail how the implementation works, it is possibly interesting how we ended up in implementing this feature into this thesis, since audio is whether a sensor nor a network interface.

While developing we came across an article of the technical university of Brunswick, Germany. This research [AQWR] is about threats brought by ultrasonic side channels on mobile devices. The basic idea is that applications on your device sometimes request to use the integrated microphone, without obviously seeing any function related to it. This part of the application that uses the microphone starts to run, when ultrasonic beacons are detected. Ultrasonic beacons are audio fragments with a frequency range between 18 and 20 kHz. The special thing about the chosen range is, that it is not recognized by the average human being. But what are parts of the application, that start working without the user noticing, doing? Lets demonstrate this with a little example.
User A installs an application of the Label B, because user A grants access to a discount code for products of label B. While installing, the application wants the permission to record audio. User A grants this permission, because he is only interested into getting the discount code. The next day, A walks into a store of Label B. There are lots of speakers in the main entrance playing music, but they embed ultrasonic beacons in each soundtrack too. The application of A triggers with entering the store, sending the exact beacon frequency to a server of B. B is now able to determine the exact location of user A, because each store uses a unique frequency. Which sound like utopia is already part of our everyday life, one popular

application using this method is called *ShopKick*.

There is way more information these applications can extract by using ultrasonic beacons, but in the following we are not concerned about the security risk. We have learned that audio can be used as a medium to transmit information between devices. Considering that almost each device is capable of recording as well as playing audio, we can build an alternative network that works parallel to the common known like Bluetooth or WiFi.

Android provides the two classes *MediaRecorder* and *AudioRecorder*, that fit the task of recording audio. The MediaRecorder is an easy to use class, that only needs a path where to store the audio sample, but it has a great disadvantage regarding to our goals. It compresses the audio sample internally, to consume less free space on the device. The problem of compressing is, that frequencies which are not heard by humans are kind of useless information that can be dropped. The compressing progress will therefore cut the bits representing this information. Since this thesis is build to set a solid and wide base for further investigation and developing, we can not afford to lose information regarding high frequencies. Thus we use the AudioRecorder class.

The *AudioRecorderManager* was designed to start recording on demand of the corresponding mobile device and to send this audio sample to the requesting device after it tells to stop recording. As discussed above, the AudioRecorder class is used for this [Goo17d].

If the application gets the request to start recording, a new worker thread is created. The first step is to create a *FileOutputStream*, where the recorded data can be stored. The data is stored within the application, resulting in no need of further permission to write to the external storage and that only our application can access the audio sample. If this stream is set up successfully, we need to create a buffer, in regard to the sample rate, channel configuration and the encoding type. Our implementation uses a sample rate of 44100Hz, mono as the channel configuration and the encoding type PCM 16. An array with correct size is necessary, because the AudioRecorder dumps the recorded values into a buffer first. Therefore it provides a method called *getMinBufferSize()*, using the values mentioned above, helping us to conclude the smallest size of the buffer needed to continue. Since our encoding is done in 16 bit, we need to choose short as the data type of our array. Using the data type byte would cause invalid information, because it is capable if storing eight bit only. Combining the type and the size, we are ready to fully initialize the buffer. Once this is done, we create an instance of the AudioRecorder and call the method *read()*. It will return how many

short values have been read during the last cycle. This value represents the upper limit when iterating over the buffer, while writing each value into the file output stream created at the very beginning. But before we start explaining what happens if current record is stopped, we would like to point out that this approach allows us to access the raw data, produced by the microphone, during run time of recording.

If the request is to stop recording, the worker thread will close the file output steam, signal the AudioRecorder instance to stop recording and finally initiate a process that will create a DataItem containing our newly recorded audio sample, before the thread closes itself. Closing the thread is simply done by setting the boolean value keeping the thread alive to false. The next task is to create an Asset of the audio sample stored in the internal storage of the application. As we have described in Subsection 4.2, we need a binary array representing our sample to successfully create an Asset. Therefore we have to open a *FileInputStream* with the previously defined path defining the location of our audio file. This input stream is forwarded to the method *getByteArrayFromAudioFile()* returning the desired byte array representation. Once this is done we are ready to attach the asset to the data item. Additionally we store a time stamp and forward the finished item to the DataTransmitter.

4.4.8 LocationScanner

The *LocationScanner* was designed to determine the current position of the Wear device. Android brings a system service that cares about all functionality concerning the location management of the device. To access this, we create an instance of the *LocationManager* class [Goo17k].
This instance provides a field, that can be accessed with the *getLastKnownLocation()* method, that returns a *Location* object that stores information about the last fixed position. This method needs a parameter that specifies the provider used to fix this position. There are two providers available, the first one is the own GPS module built in the device, the second one is called the network provider. If this provider was used, the location was determined based on availability of cell towers and WiFi access points around the device.
Once the Location object is retrieved, we can access the longitude and latitude of the devices last position. To make the information of the data item more interesting we use a class called *GeoCoder*. This class maps the double values of longitude and latitude to the corresponding address [Goo17h]. Using this class, requires an internet connection. Additionally to this

information, we put the time of the location approximation into the data item and how many satellites have been used for fixing the current position.
If the last known position is too old maybe, or simply not present cause there has not been a location scan yet, we can request one with help of the *LocationManager* instance. Therefore we use the function called *requestLocationUpdates()*. Similar to the *WifiScanner* this method triggers the scanning process only, thus we need to implement the interface *LocationListener* to get updated, if the new location was determined successfully. Since the process of location fixing can take some time, the LocationScanner class implements the *GpsStatus.Listener*. This interface gives report, whenever the number of active satellites used changes during the computation. This information is separately transmitted to phone and has the effect of a progress bar. A growth of active satellites often indicates, that the new position will be accessible soon. Once the new position is fixed, the method *onLocationChanged()* will be called with a new Location object. From now on we can continue as described earlier.

4.4.9 ControlelementListener

The idea behind the *ControlelementListener* class is, to inform the user of the mobile application about touch events on the screen of the Wear device. There is a class called *View.OnTouchListener* that is capable of achieving this goal. Once the request of the mobile application is received, this listener is activated and waiting for a touch event. To activate the listener, we need to use the *setOnTouchListener()* method, that is provided by the *View* instance of our launching Activity of the Wear application. If the user interacts with the screen, it creates a data item storing the integer value 1 and a time stamp before forwarding this package to the DataTransmitter class. Once this event is triggered, the touch listener is deactivated. Reactivation is only possible, if the mobile application user sends a new request.

4.5 Wearable Application

The wearable application is responsible for entering the Wear network. It will provide a reference to the communication channel that connects the device to the network, as described in Subsection 4.4. Furthermore it is the connection between the mobile application and the library. Each request will be send to Wear application, depending on the message received,

the class satisfying the demand will be started from the library. In Section 4.3 we explained how to listen for incoming request.
The Wear application is equipped with a BroadcastReceiver that is listening to changes in the devices WiFi status. This receiver works analog to the status update function given by the WifiConenctionManager, that is described in the last paragraph of Section 4.4.5. The only difference is, that the NetworkInfo instance is included within the Intent of type *WifiManager.NETWORK_STATE_CHANGED_ACTION* that is send across the device. This additional BroadcastReceiver will guarantee the most current WiFi status, avoiding to use manual refreshes all the time.

4.6 Mobile Application

The mobile application is, like the Wear Application, responsible for entering and providing a reference to the Wear network. Its duty is divided into sending requests and displaying received data recording to the request. How to send messages targeting the Wear devices is described in Subsection 4.3. To display received data, we need to listen for them first, this is described in Section 4.1. This process of requesting and receiving data is initiated through Buttons deployed on the UI of the application. Clicking the Button *Location-Scan* will start the LocationScanner class resulting in a data item containing our current position synchronized across the network. This information will be received and displayed in the corresponding *TextField* of the UI.

Chapter 5

Evaluation

In this Chapter we want to present the tests designed for the applications developed in this thesis and evaluate the results based on the collected data. Section 5.1 will introduce the devices used. The following Section 5.2 will explore the limitations and possibilities of the infrastructure connecting our devices and explain our test environment in detail. Sections 5.3 and 5.4 will illustrate in what way the applications will require the devices' hardware regarding the CPU, memory and battery.

5.1 Test Devices

While choosing our test devices, we need to keep our demands recording the Wear and mobile application in mind. Since the mobile application should contain as little functionality as possible, it will ensure that it will work properly on many Android OS APIs. The Wear device on the other hand must contain as much functionality as possible to fulfill our goal of building a wide base of information. Table 5.1 will list the devices we finally decided on.

Table 5.1: Devices used for Testing

Name	Model	Type	Android-Version
Samsung Galaxy S6	SM-G920F	Smartphone	6.0.1
Huawei Y3	Y360-U61	Smartphone	4.4.2
Sony SmartWatch 3	SWR50	Smartwatch	6.0.1

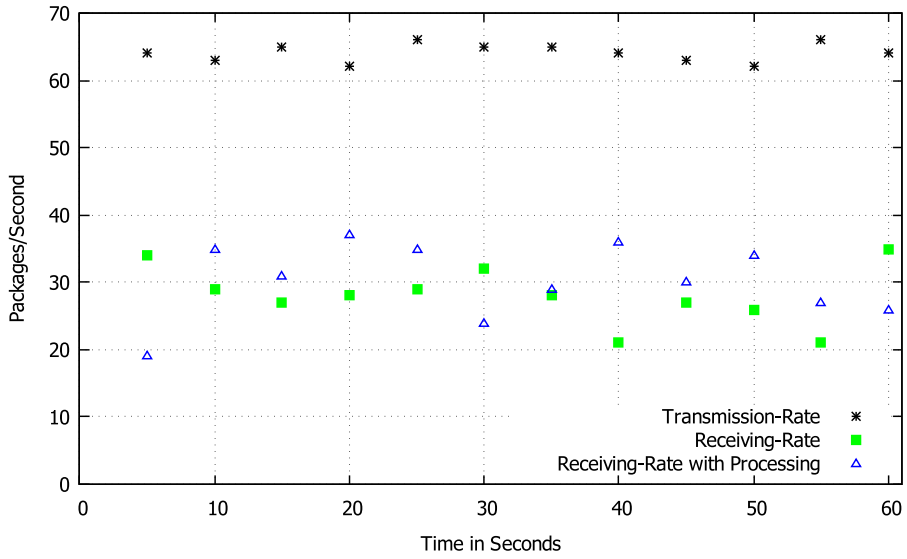

Figure 5.1: Synchronize 28 Byte Items

5.2 Network Performance

To test the performance of our network, we first need to create an event of high frequency that occurs in a regular time interval. To simulate an event like this, we make use of the sensors provided by our device described in Section 4.4.2.

Since each sensor event initiates a transmission of a data item, the idea was to raise the number of active sensors slowly to determine how much packages could be send each second. Surprisingly the very first sensor of type *Accelerometer* activated, exceeded the limitation that seems to be at approximately 30 Packages per second.
To eliminate the suspicion that the mobile device, especially the further processing of the received data could slow down the receiving rate, we implemented a receiver type that dropped all items right after successfully getting them. The results from this test, illustrated in Figure 5.1, showed nearly no differences in average. In fact the test run with processing was slightly faster. This leads to the conclusion that the network is being the limiting factor.

Recapping the results from this test run, we had an average of 28 Packages per second with a package size of 28 Bytes. This results in a bandwidth of 0,77 KB per second. Since we observed that we can not increase the number of transmission per second, we need to increase the data item size monitoring the bandwidth again. The next experiment enlarges the data item size from 28 Bytes to 10028 Bytes. Figure 5.2 display the results of this test run.

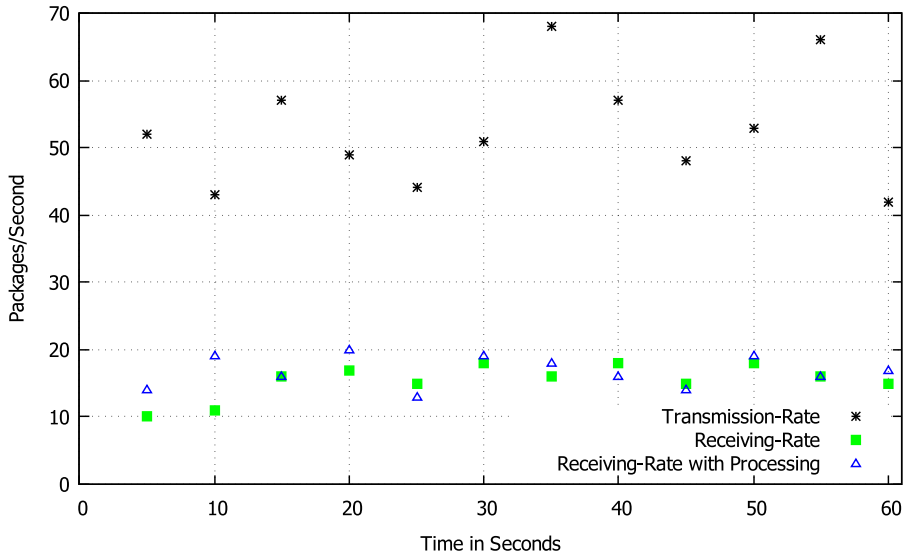

Figure 5.2: Synchronize 10028 Byte Items

Again we distinguished between processing and not processing the received items. This time we had an average of 16 Packages per second, which returns a bandwidth of 156.69 KB per second. In some test runs we achieved a bandwidth up to 220 KB per second. Increasing the size of the data items, resulted in a more than 200 times larger bandwidth.

5.3 CPU and Memory Usage

To monitor how much of the devices' hardware is used to run the applications, we will synchronize sensor data between them similar to the first test of Section 5.2. This time the test will last for ten minutes, while synchronizing we will protocol the CPU and memory usage on each device. Figure 5.3 shows the test run for the Huawei Y3 while Figure 5.4 represent the results for the Samsung Galaxy S6. The Y3 has an average CPU load of about 32%, the S6 an average load of about 17%. The memory allocated by the applications is approximately 10 MB for the Y3 and 25 MB for the S6. These differences are caused, because the hardware built in the S6 is superior to the hardware of the Y3.

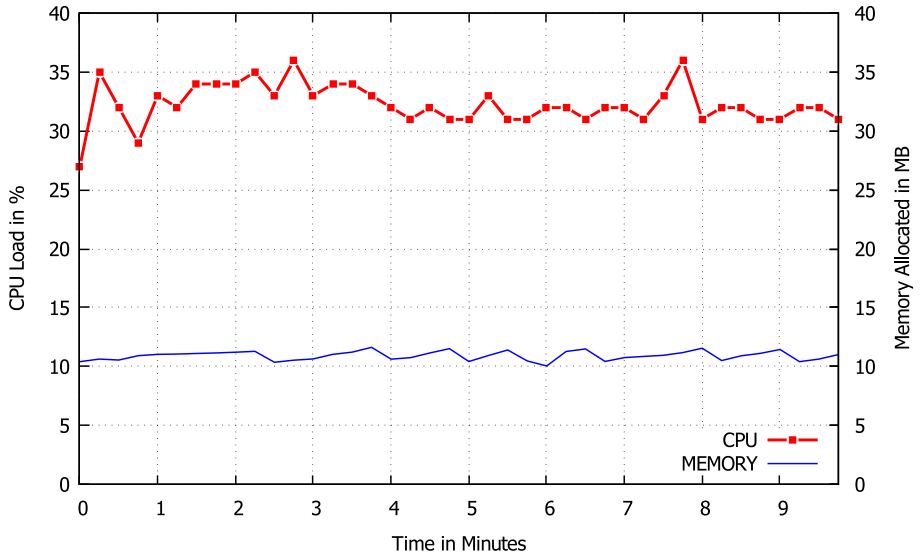

Figure 5.3: CPU and Memory Usage of the Huawei Y3

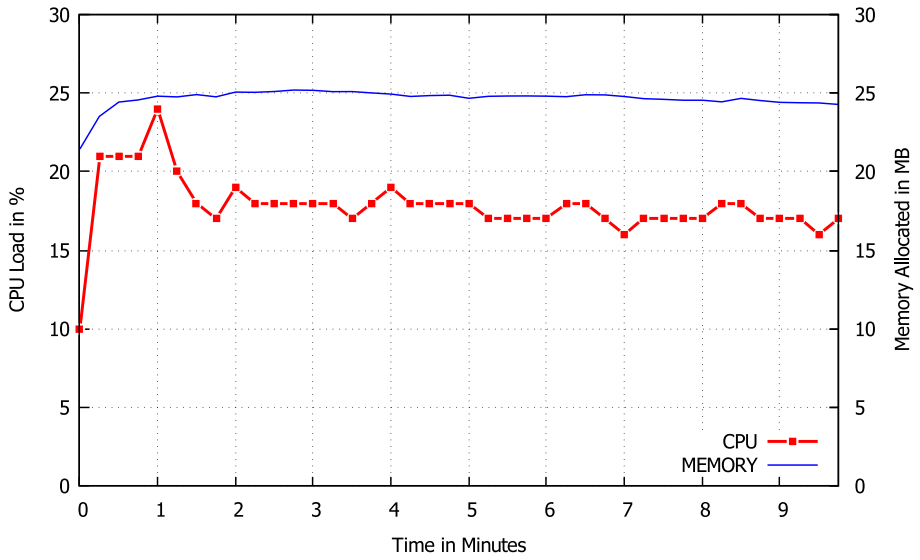

Figure 5.4: CPU and Memory Usage of the Samsung S6

More interesting is the resource usage of the wearable, demonstrated in Figure 5.5. While the CPU usage is generally around 40% during the whole test, the memory usage increases continuously. This behavior reveals a very important fact that should be kept in mind when developing for Android Wear. As we have learned in Chapter 4.2 the DataAPI represents a reliable method to transport data. Since our transmission rate is way higher then the capacity of the network to transport these packages (See Figure5.1), the Wear device is forced to store the remaining synchronization events.

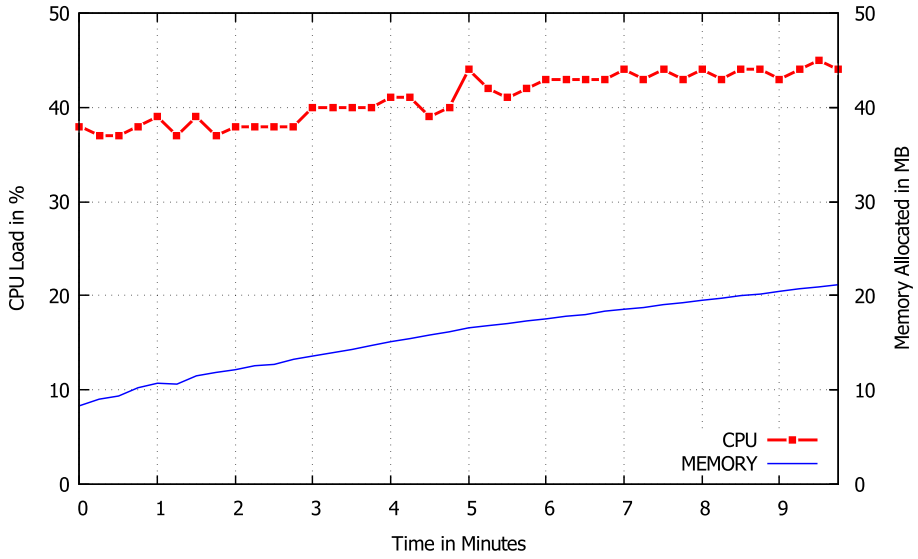

Figure 5.5: CPU and Memory Usage of the Sony SmartWatch3

Another negative aspect resulting from this can be, that we are receiving outdated sensor data. Regarding in how this data is used for further processing it could be essential, that the sensor values are nearly live.

To demonstrate how devastating it is to not keep a balance in transmission rate and package size we ran the same test again with increasing the payload by 10KB. The result can be observed in Figure 5.6. This time we were only able to keep the test alive for about two minutes, because it crashed due to the lack of available free space.

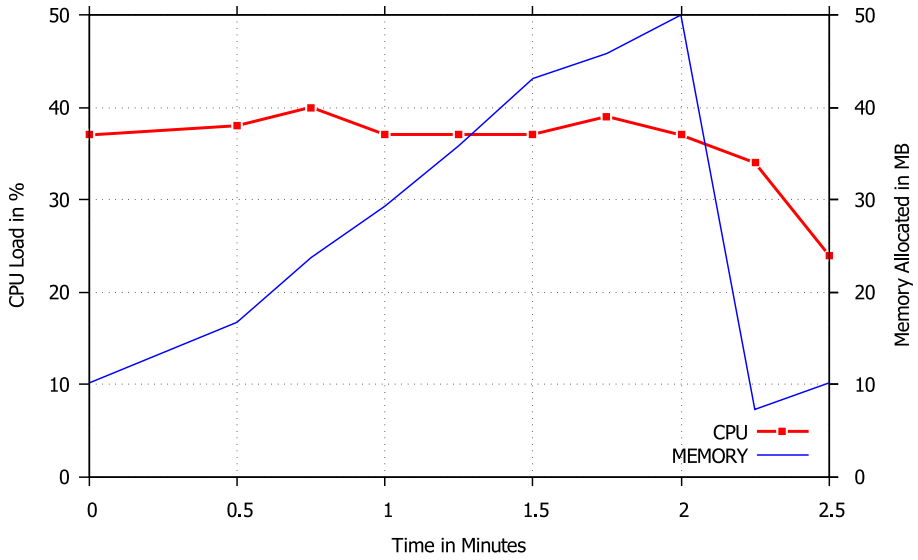

Figure 5.6: CPU and Memory Usage of the Smartwatch with enlarged packages

5.4 Battery Life

In this Section we will reveal how the hardware usage, caused by our application, influences the battery of the running device. Therefore we additionally recorded the battery status while logging the CPU and memory usage as described in the Section above. The results can be observed in Figure 5.7. Since both applications are designed to run as the foreground Activity, we need to consider the battery consumption of the devices' screen. Thus we ran the applications doing nothing than keeping their screens unlocked for ten minutes visualized in Figure 5.8.

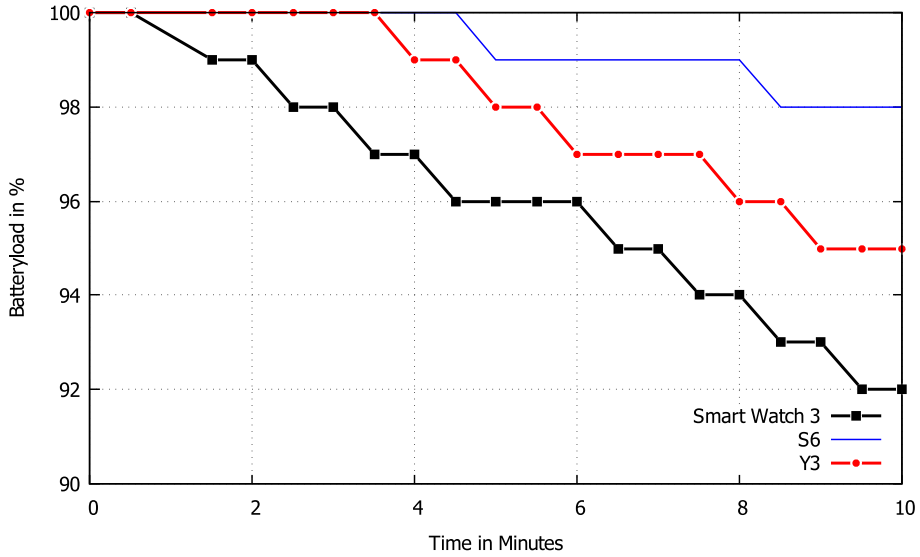

Figure 5.7: Battery usage while synchronizing

During the test run with the extended data items in the previous Section we noticed, that the CPU load of the SmartWatch 3 was not influenced, despite to the huge relative growth of the package payload. This led to hypothesis that the main battery consumer is not the process of computing and storing data into packages but the Bluetooth negotiation between the devices itself. Therefore we create a new test scenario, where the item size is increased slightly to 100 Byte, to guarantee the same duration of testing without any aborts due to missing free space. Since the results regarding the battery status were exactly the same at the end of the test, despite increasing the package size and the bandwidth by the factor of three, the hypothesis made is strengthened.

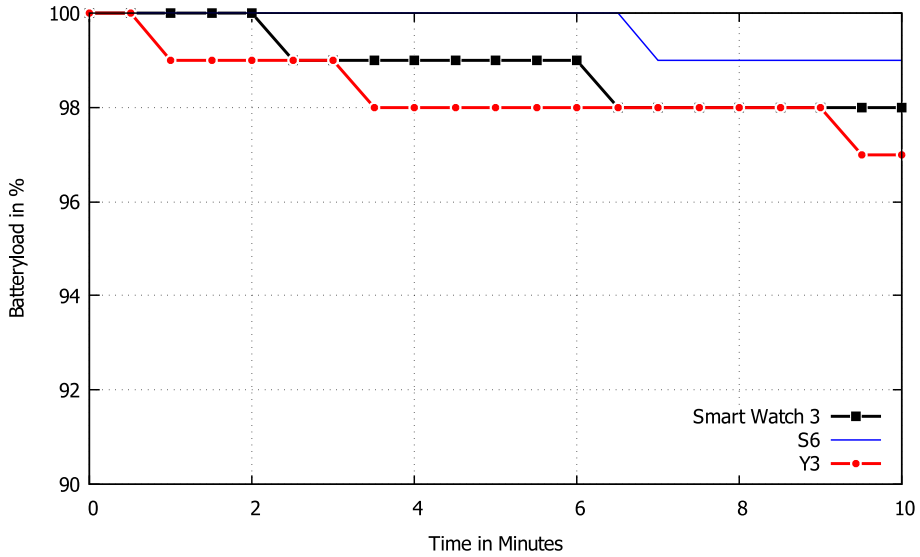

Figure 5.8: Battery usage while not synchronizing

Reviewing this test, we could have stopped transmitting after only four minutes, since we have already forwarded the same amount of data as the previous one during the whole test duration displayed in Figure 5.7. This would result in a consumption of only 3% of the devices battery instead of the prior usage of 8%.

Chapter Conclusion Recapitulating it can be said, that each connection between the devices should be used as efficient as possible, since negotiation between the devices is the main energy consumer. With bundling information into one package we ensure to use the capacity provided by the network as good as possible resulting in the need of less connections to transport data at the same time. Thus bundling packages does not only increase the bandwidth, it also saves battery. Before we move to the last Section it should be mentioned that the test cases created are made up of the most hardware intensive functions provided by the application, therefore the results demonstrate an upper bound regarding the hardware consumption. Table 5.2 lists all services that are provided by the Wear application connected to their package consumption. Stream in this context means, that once the request is done updates need to be disabled through the GUI again.

Table 5.2: DataItems needed to answer a Request

Service	**DataItems**	**Stream**
Location	1	No
Satellites	1	No
Control Elements	1	No
WiFi Networks	1	No
WiFi Status	1	No
Audio	1	No
Bluetooth	2	No
Sensors	0-60/s	Yes
Accelerometer	*∼60/s*	*Yes*

Chapter 6

Conclusion and Future Work

The first Section 6.1 of this Chapter will present a conclusion of the achievements made while developing this thesis, in regard to the demands set at the beginning. Section 6.2 will build on this conclusion illustrating possible future options that can be made to extend and improve the applications.

6.1 Conclusion

While developing this thesis, we created two applications. One application is running on the mobile device while the other one is running on the Wear device. The communication between the devices was defined in the way, that the mobile application could request a service on the Wear application. The Wear application on the other hand was responsible for answering this request. Due to the different types of services created for the Wear application we achieved the goal of providing a solid and wide base of information stored on the mobile device where the origin of the data is the corresponding Wear device. A wide base of additional information is urgently needed for decision making, especially if we want to influence the functionality and user experience of mobile applications as good as possible. In this thesis we went one step further and did not limit ourselves with receiving information only, we provided a function to actively manipulate the wireless status of our Wear device, by telling it to connect to a mobile application desired network. This allows us to control the kind of future received data in a certain way.

The other objective of this thesis was explore the possibilities and limitations nowadays Wear devices face. The only negative experience made while developing was the lack of a system service, that prevented us from getting the *Write_Permission* on the Wear device. This missing permission is needed for opening a hotspot on Android, thus we had to stop developing on this feature. Luckily this was the only negative experience made. In fact we were surprised about the reliability and working speed of these devices. In most cases developing for Android Wear was exactly like developing for an Android mobile application.

Summing up we can say that the wearable technology is currently in the early stages of mobile communication, nevertheless they provide a huge amount of opportunities that can be accessed by developers of the Android community already today. The release of *Android Wear 2.0* in June 2017 reveals the trend of future Wear devices. These devices will be equipped with faster hardware, an increasing number of interfaces and the ability to operate more and more independent from a mobile device. We think that wearable devices are going to be an important part of the mobile communication future. This is the reason why working and integrating them into current projects is an interesting and future proof decision.

6.2 Future Work

The are options to extend and improve the current implementation of the thesis. In the following we will demonstrate some approaches.

One opportunity is to integrate this project into Opptain, to test whether it is possible to influence the connectivity of a mobile phone with the additional information brought by the Wear device. The issue demonstrated in Subsection 3.2.1 could be the first major task to improve.

Another possibility is to resolve the problems caused by the imbalance between transmission and receiving rate discovered in Chapter 5. We could implement observers monitoring the current network load, helping us to dynamically determine the best transmission rate possible.

Currently the Wear application is operating only, if it is the foreground Activity. Implementing it as a background Activity, would allow the user to continue interacting with his wearable

as usual. This will result in fetching data, without interrupting any current process working on the Wear device.

Since our test device did not receive an update on Android Wear 2.0, it would be interesting to patch this project to the latest version. Maybe this update provides improvements concerning our services. Moreover it would be lovely if the missing system service that prevented us to develop a hotspot function is available now.

The GUI of the mobile application is quite minimal, as well as the GUI of the Wear application. In fact the GUI of the Wear application provides nearly no functionality. To achieve a better user experience we could work on expanding and prettifying the overlay of each application's surface.

Bibliography

[AQWR] ARP, Daniel; QUIRING, Erwin; WRESSNEGGER, Christian; RIECK, Konrad: Privacy Threats through Ultrasonic Side Channels on Mobile Devices.

[Goo17a] GOOGLE INC.: *Accessing the Wearable Data Layer | Android Developers*. https://developer.android.com/training/wearables/data-layer/accessing.html, June 2017. Last checked: June 27th 2017

[Goo17b] GOOGLE INC.: *Activities | Android Developers*. https://developer.android.com/guide/components/activities/intro-activities.html#tcoa, June 2017. Last checked: June 27th 2017

[Goo17c] GOOGLE INC.: *Application Fundamentals | Android Developers*. https://developer.android.com/guide/components/fundamentals.html, June 2017. Last checked: June 27th 2017

[Goo17d] GOOGLE INC.: *AudioRecorder | Android Developers*. https://developer.android.com/reference/android/media/AudioRecord.html, June 2017. Last checked: June 27th 2017

[Goo17e] GOOGLE INC.: *Bluetooth | Android Developers*. https://developer.android.com/guide/topics/connectivity/bluetooth.html, June 2017. Last checked: June 27th 2017

[Goo17f] GOOGLE INC.: *ConnectivityManager | Android Developers*. https://developer.android.com/reference/android/net/ConnectivityManager.html, June 2017. Last checked: June 27th 2017

[Goo17g] GOOGLE INC.: *Dashboards | Android Developers*. https://developer.android.com/about/dashboards/index.html, June 2017. Last checked: June 27th 2017

[Goo17h] GOOGLE INC.: *Geocoder | Android Developers*. https://developer.android.com/reference/android/location/Geocoder.html, June 2017. Last checked: June 27th 2017

[Goo17i] GOOGLE INC.: *GoogleApiClient | Android Developers*. https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient, June 2017. Last checked: June 27th 2017

[Goo17j] GOOGLE INC.: *Intents and Intentfilters | Android Developers*. https://developer.android.com/guide/components/intents-filters.html, June 2017. Last checked: June 27th 2017

[Goo17k] GOOGLE INC.: *Location and Maps | Android Developers*. https://developer.android.com/guide/topics/location/index.html, June 2017. Last checked: June 27th 2017

[Goo17l] GOOGLE INC.: *No System UI for granting Write Permission | Issue Tracker*. https://issuetracker.google.com/issues/37079055, June 2017. Last checked: June 27th 2017

[Goo17m] GOOGLE INC.: *Overview of Google Play Service | Android Developers*. https://developers.google.com/android/guides/overview, June 2017. Last checked: June 27th 2017

[Goo17n] GOOGLE INC.: *Permissions | Android Developers*. https://developer.android.com/guide/topics/permissions/index.html, June 2017. Last checked: June 27th 2017

[Goo17o] GOOGLE INC.: *Process and Threads | Android Developers*. https://developer.android.com/guide/components/processes-and-threads.html, June 2017. Last checked: June 27th 2017

[Goo17p] GOOGLE INC.: *Requesting Permissions at Run Time | Android Developers*. https://developer.android.com/training/permissions/requesting.html, June 2017. Last checked: June 27th 2017

[Goo17q] GOOGLE INC.: *Sending and Receiving Messages | Android Developers*. https://developer.android.com/training/wearables/data-layer/messages.html, June 2017. Last checked: June 27th 2017

[Goo17r] GOOGLE INC.: *Sensors Overview | Android Developers*. https://developer.android.com/guide/topics/sensors/sensors_overview.html, June 2017. Last checked: June 27th 2017

[Goo17s] GOOGLE INC.: *Service | Android Developers*. https://developer.android.com/reference/android/app/Service.html, June 2017. Last checked: June 27th 2017

[Goo17t] GOOGLE INC.: *Syncing Data Items | Android Developers*. https://developer.android.com/training/wearables/data-layer/data-items.html, June 2017. Last checked: June 27th 2017

[Goo17u] GOOGLE INC.: *WifiManager | Android Developers*. https://developer.android.com/reference/android/net/wifi/WifiManager.html, June 2017. Last checked: June 27th 2017

[Goo17v] GOOGLE INC.: *WiFiManager needs GPS for Scan | Issue Tracker*. https://issuetracker.google.com/issues/37060483, June 2017. Last checked: June 27th 2017

[Ipp15] IPPISCH, Andre: *A fully distributed Multilayer Framework for Opportunistic Networks as an Android Application*, Department of Computer Science, Heinrich Heine University Düsseldorf, Diplomarbeit, März 2015

[M.Z17] M.ZINOUNE: *Why is Android built on Linux Kernel*. http://www.unixmen.com/why-is-android-built-on-linux-kernel/, June 2017. Last checked: June 27th 2017

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 30.June 2017 Bashkim Berzati

Diese CD enthält:

- eine *pdf*-Version der vorliegenden Bachelorarbeit
- die LaTeX- und Grafik-Quelldateien der vorliegenden Bachelorarbeit samt aller verwendeten Skripte
- die Quelldateien der im Rahmen der Bachelorarbeit erstellten Android Software
- den zur Auswertung verwendeten Datensatz
- die Websites der verwendeten Internetquellen