Location and Movement-Awareness in Android-based Opportunistic Networks

Bachelor Thesis

by

Justin Marks

born in

Neuss

submitted to

Technology of Social Networks Lab

Jun.-Prof. Dr.-Ing. Kalman Graffi

Heinrich-Heine-Universität Düsseldorf

Juli 2017

Supervisor:

Andre Ippisch, M. Sc.

Abstract

The Smartphones these days are capable of increasing our standard of living in so many ways. We use them as an alarm clock, camera or navigation system and new technologies like Augmented Reality are showing, that they are capable of doing so much more. Furthermore existing systems could be optimized or expanded, with the data provided from Smartphones.

In this thesis we want to make use of these capabilities and extend Smartphone-based Opportunistic Networks to be aware of Location and Movement of the device. We lay the foundation for further development. We present an extension for an Android-based Opportunistic Network that is using the Location and Movement -awareness for GPS-based routing.

We developed an extension that is able to recognize and classify the user's movement, to determine the user's position without the usage of GPS data and to create a data collection with information about the location the user has been visited.

We evaluated the accuracy of the provided data by this extension and measured the additional power usage.

We came to the conclusion the Android-based Opportunistic Network greatly benefit from the developed extension. In some sections further development is required.

Acknowledgments

First of all I would like to thank my girlfriend for supporting me during this thesis and for helping to set up the various tests.

Then I want to say Thank You to my father for providing me with unfailing support and continuous encouragement throughout my years of study.

At least I would like thank to my thesis advisor Andre Ippisch, M. Sc. He is a great guy and his door was always open whenever I ran intro trouble.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

In this section we want to describe the motivation behind this thesis and want to look at related work.

The Smartphones these days are capable of increasing our standard of living in so many ways. We use them as an alarm clock, camera or navigation system and new technologies like Augmented Reality are showing, that they are capable of doing so much more. Furthermore existing systems could be optimized or expanded, with the data provided from Smartphones.

In this thesis we want to make use of these capabilities and extend Smartphone-based Opportunistic Networks to be aware of Location and Movement of the device. We lay the foundation for further development. We present an extension for an Android-based Opportunistic Network that is using the Location and Movement -awareness for GPS-based routing.

1.2 Related Work

S. Lu, Y. Liu and M. Kumar published in year 2012 an article with the title *LOOP: A location based routing scheme for opportunistic networks*. They presented a routing scheme, that

predict the node's future movement, based on analysed data they received from the previous behaviour of the node itself.

1.3 Outline

In this chapter we describe our motivation behind this work and also looked at related work.

In Chapter 2 we will introduce the basic fundamentals that are used within this thesis.

In Chapter 3 we will describe the demands for our system and design and each aspect of it.

In Chapter 4 we will show how we implemented the system.

In Chapter 5 we will present test results and an evaluation of the system.

In the Chapter 6 we will review the conclusions of this work and will point out aspects that could be optimized in future work.

Chapter 2

Fundamentals

In this section we want to introduce the basic fundamentals that are required for the thesis.

2.1 Android Smartphone

2.1.1 Operating System

The Android Operating System (Android OS) is an open source Linux-based software stack. The different stack layer are: Linux Kernel, Hardware Abstraction Layer, Native C/C++ Libraries, Android Runtime, Java API Framework and Applications. The Java API Framework contains a lot of High-Level methods and is used to develop Android applications.

2.1.2 Applications

Android Applications are mostly capsuled and running independently on their own. Therefore every Application has its own thread and virtual machine. An Application is made of two kinds of components: Activities and Services. In order to access system or user information an Application requires granted permission (eg. Permission to read the file system).

Manifest

Every Android Application has a structured XML-File that declares each component used by the Application. It also contains information about each permission that is requested by the Application.

Activity

An Activity is a graphic user interface and allows the user to interact with the Application. Normally an Application is made of more than one Activity. Each Activity has its own purpose. For example a calender application has an activity to show the calender and another activity to make it possible to add information.

Service

Services perform operations in the background. They do not have a graphic user interface.

Intent

An Intent is a call to start an Activity or a Service. There are two kinds of Intents. **Explicit intents** start an Activity by calling its class name and they are used to start Activities that belong to the application. **Implicit intents** start an Activity by asking for a specific action. For example, if you want to take a picture, you can use an implicit intent to request an activity that is capable of taking pictures. This activity does not has to be provided by the application you are currently using. Furthermore it is possible to add data to the intent. These data could be used by the Activity.

2.1.3 Built-in Sensors

Smartphones have a variety of built-in sensors. These sensors are devided into *Motion Sensors*, *Environmental Sensors* and *Position Sensors* [Gooa].

2.2 Global Positioning System

The **Global Positioning System** (GPS) is a satellite based navigation system that uses radio frequencies of 24 orbiting satellites to find your exact location. It was originally developed by the U.S. Department of Defence for the military to know the location and movement of planes, ships and soldiers. The GPS works in any weather conditions, anywhere in the world, 24 hours a day. A GPS is used to tell how far a device has travelled, the device's current direction, the speed and ETA. Satellite navigation systems all work in the same way. There are three parts: the network of satellites, a control station somewhere on Earth that manages the satellites, and the receiving device you carry with you. Each satellite is constantly beaming out a radio-wave signal toward Earth. The receiver listens out for these signals and if it can pick up signals from three or four different satellites, it can figure out your precise location.

2.2.1 Geographic Coordinate System

The **Geographic Coordinate System** is used to describe a specific location on earth. The system overlays the earth with a grid of horizontal and vertical lines. The *Equator* and the *Prime Meridian* are marking the central point. In this point the values of *Latitude* and *Longitude* are both *0*. Moving north or south from this point would change the *Latitude* value and moving east or west would change the *Longitude* value.

Chapter 3

Demands and Design

In this chapter we will build a solution with the basic fundamentals of this thesis, as presented in the previous chapter. The demands of the thesis are declared in the first Section 3.1. In the second Section 3.2 we will develop a concept which will implement the extension described in Section 1.1.

3.1 Demands

The idea behind this thesis to take benefit of the movement awareness that goes with Mobile Opportunistic Networks (Mobile OppNets) to expand the functionality of Opptain. Therefore is it very important to meet all demands required by Opptain, especially to work without the Internet and transfer the data fast and securely [Ipp15].

In order to increase the spread of Opptain the extension should minimal increase the Android Application Package (APK) size of Opptain to ensure a fast pass on from one device to another device.

Furthermore, this extension should use the power of the device as less as possible. A major challenge for location aware mobile applications.

3.2 Design

In this section we want to design a concept for the subsequent implementation of the in Section 1.1 described extension of the Android-based Mobile Opportunistic Network Opptain with the requirements describe in Section 3.1. The extension covers a range of functionalities, therefore the concept is separated into each of them.

At first we design the concept of a possible GPS-Based Routing 3.2.1, which is the main functionality of this extension and is related to all the other functionalities. In the second Section 3.2.2 we design the concept of **User Interaction** and its purpose regarding GPS-Based Routing. In the last Section 3.2.3 we design the background process which collects user related information that are required for the routing.

Location

A location could be described in different forms. People usually expressing a location by naming an address, which holds variants of information like a country, postal code, street name, house number and to be specific they use all of this information together. This form is not suitable for the design of the extension, because it requires a lot of information and it is not comparable.

The first thing that makes this form not suitable is the amount of information. Every location has nearly completely different values. To be specific an address needs the name of the country, the street, the postal code and the housing number. All that information need to be saved and exchanged in each Bundle which would increase the number of meta data. Furthermore these information are mostly in text form, which increase the number of information that are need to be handled. For example "Germany", "Deutschland", "De" are the mostly common ways to address Germany and they mean all the same. A person could know this or could easily look this information up, but our extension should not carry all kinds of information, nor could search for this information on the internet regarding to the demands describe in Section 3.1.

All that information lead to the second, much bigger problem of this form. It is not comparable, because their meaning has no system. Street name are mostly random words and have

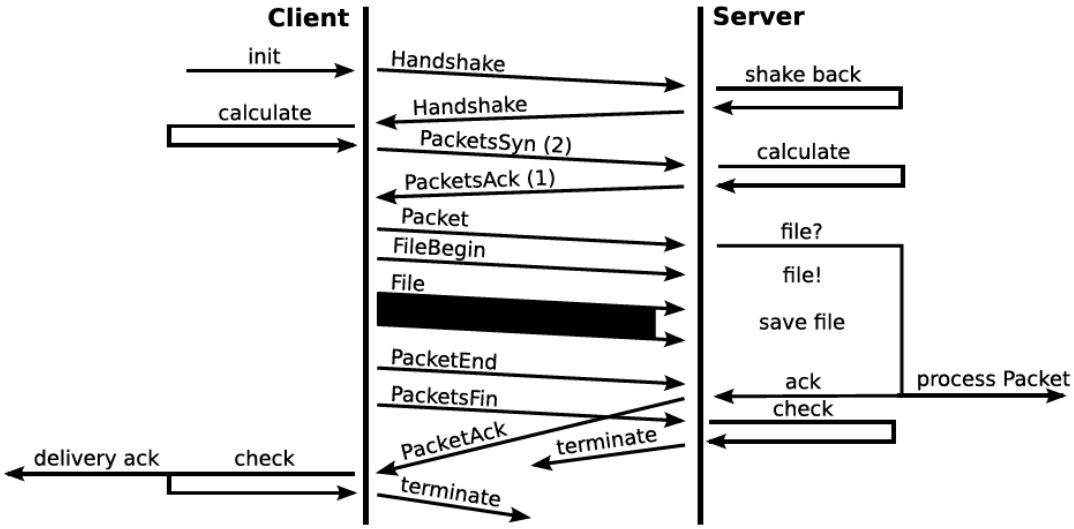


Figure 3.1: Exchange of data between smartphones according to the given protocol in Opptain. Source: [Ipp15]

no physical relation, such as coordinates. Therefore it is not possible to calculate or compare the distance between addresses.

A way more effective form to describe a location within our extension is to use coordinates 2.2.1, because information exchange and storage is only increased by two values (Latitude, Longitude) and it is metric-based, which enables comparing and calculating. In this thesis a location ist described by coordinates.

3.2.1 GPS-based Routing

Routing Protocols (RP) are a necessary part of every network. They define how nodes communicate among each other. The network properties like the number of nodes, their reliability or their residence time, have impact to the network. Optimal stability and reliability of a network requires the correct choice of Routing Protocols. Furthermore their number increase with new technology or scientific findings. In Android-based OppNets we have the possibility to use information about the movement and location of the nodes to develop or expand Routing Protocols.

The Figure 3.1 shows the communication between two devices. After the handshake, the

Client offers data that could be send to the *Server*. In the figure, this step is named **PacketsSyn (2)**. In detail, the *Client* has a bundle or more, that he wants to offer to the *Server*. Therefore it sends *Meta Information* of each bundle to the *Server*. The *Server* analyse the passed information and determines, which bundle he would like to receive. After that, the *Clients* start sending the accepted files.

Various factors are helping the *Server* to make a decision. Either the bundle is accepted or declined. Our GPS-based routing shall expand these factors with the precondition, that the meta data contains GPS-based data of the destination. The *Server* needs an algorithm to calculate the probability of reaching the destination. This value should help the *Server* to make a decision. For this thesis a basic approach of this algorithm should be implemented.

GPS position are very accurate and the probability that a user with a bundle reaches this destination should be very low. To raise the probability and to reduce the priority of an accurate GPS position, another value should be send along the GPS data in the *Meta Information* of each bundle. This value should be a tolerance limit. The higher this value is, the lower should be the accurate of the GPS position.

3.2.2 User Interaction

GPS-based Routing requires additional information, like the GPS data of the destination. These information come from the sender and needs to be added manually through a graphical interface. This action represents the User Interaction of this extension. This graphical user interface (GUI) should be simple to use and should perform a single action, to select a location 3.2. This Activity should be called *LocationChooserActivity* (LCA). The LCA should be highly versatile in use. Especially Third Party Applications should be able to use the GUI to let their user add a destination. Furthermore it could also be used to set up some location-based settings.

The tolerance limit mentioned in the previous section should also be set in the LCA as a radius around the location. It therefore follows that the user should be able to set three values (Latitude, Longitude and Radius). To apply these values as easy as possible a map should be displayed where the user is able to interact with like zooming in and out or to navigate. The GUI should be designed similar to common Map-Applications like Google Maps.

An important aspect that is required and described in section Demands 3.1 is the capability to be used offline. Therefore the map should be accessible without internet.

3.2.3 User Data

An accuracy collection of data are needed for the **GPS-based Routing**. Therefore we need to design a background service that collects geographic related user data. These data should be periodically read and saved. As in Section **Location** 3.2 mentioned, the most reliable form of describing geographic data are **Geographic Position Systems**. Therefore, and because Smartphones are able to determine GPS position via the embedded GPS sensor, we are going to collect GPS data.

Requesting the current GPS position is very consuming, therefore the service should only request location updates if the user is not at the same position like he was in the period before. The service has to classify the user's movement into three possible states.

- While being in the *Sleeping* state, the service has not detected any movement and therefore it does not request location updates. This state is the power saving state.
- *Walkling* is classified as the second state, where the service only requests location updates if the user has walked away from the last known position.
- The last state is the power consuming *Driving* state. Being in that state forces the service to request location updates in every period, because the user is moving very fast.

Chapter 4

Implementation

In this chapter implement the extension, we designed in the previous chapter.

4.1 Development

We are using Android Studio, the official integrated development environment (IDE) for developing the extension. The IDE is free to use and provided by Google. Our testing devices are Sony Xperia Z5 and Huwei Y3.

4.2 User Interaction

In this section we will implement the *LocationChooserActivity*, as possibility to allow the user to select a geographic location. Therefore the Activity should display an interactive map.

4.2.1 Requirements

First we need an API to build the map application. Android comes with Google Maps API, which does not allow to copy, pre-fetch or cache any content [Goo17a]. Therefore the Google

Listing 4.1: Converting OSM data into Mapsforge map

```
$ ./osmosis  --rx file=../duesseldorf.osm
         --mapfile-writer file=../duesseldorf.map
```

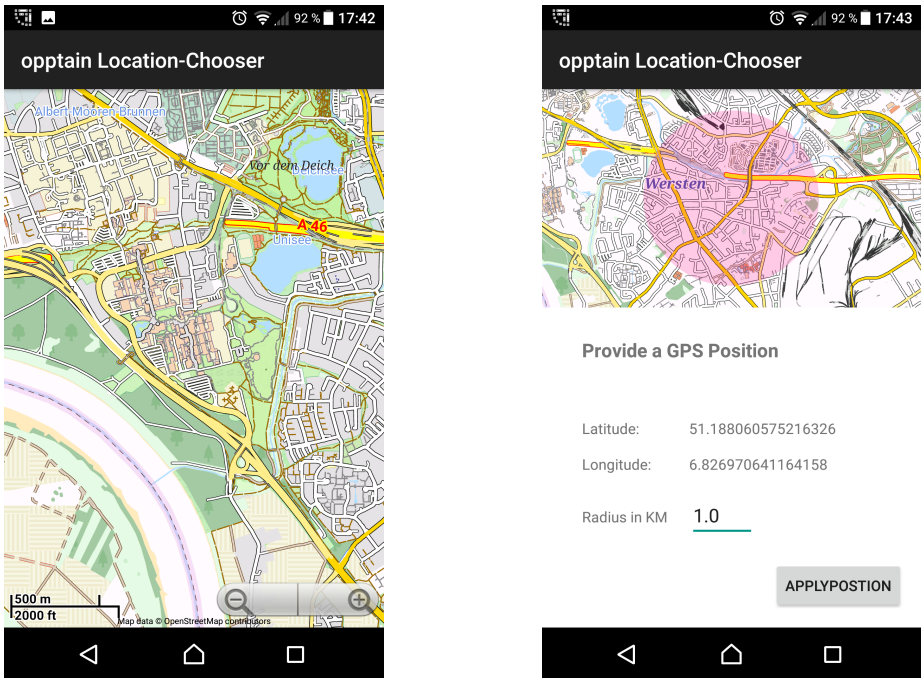
Maps API is not permitted in term of the demands of this work. A free to use alternative is *Mapsforge* [Map17b] which provides a free, open-source, offline vector map library for Android and Java-based applications. Furthermore it is capable of creating offline maps as well as modify map in size or style. In order to create our own map with the Mapsforge API we need card material. The most known provider of free geographic data is the community project *OpenStreetMap* [Ope17b], that collects geographic based information. An OpenStreetMap file (*.osm) is licence-free and could be modified. The project offers pre formatted files for nearly every country in the world as well as the opportunity to download a custom area [Ope17c].

OSM files may vary in two aspects. Firstly they vary in their content and design, because OSM provides different card material. For example they have card materials especially for cycling maps or traffic maps. Secondly they vary in coverage, which determines the size of the map. The file itself is a structured XML file that contains geographic elements and their relation between each other [Ope17a] and cannot be used by *Mapsforge*.

Osmosis [Osm17] is a free command line Java Application for processing OSM data. In combination with the *Mapsforge* it is capable of compiling an osm file into an usable map file for *Mapsforge*.

The output of this compilation is a map in the Mapsforge Binary Map File Format [Map17a]. The header of this file includes various information about the map properties like the spatial bounds or maximum and minimum Zoom Level. The Zoom Level describes the level of accuracy of the location information. A map with a very high Zoom Level contains accurate information about a location, but consequently the size of the map file is also high. The reason for that is that map files contain layers for each zoom level. Each layer contains a set of tiles, little pieces of the map. The higher the Zoom Level, the higher the number of tiles for each layer.

For for the purpose of this thesis we download and compiled a simple map of Düsseldorf that is delivered with Opptain (See Listing 4.1).



(a) Default View

(b) View after selecting a location

Figure 4.1: *LocationChooserActivity* (LCA)

4.2.2 Activity

In this section we will describe the class *LocationChooseActivity*. To be an Android Activity this class extends the class *AppCompatActivity*. Consequently, the class has the life circle of an *Activity*. Firstly the class checks if the Activity was started with additional data 2.1.2. Additional data could be Latitude, Longitude and Radius value. If these data exist, they will be the center of the map (Listing 4.2). The idea behind this feature is to simplify the usage. The user could adjust his location without searching for the old Location. (See code extract)

Secondly the permission (`READ_EXTERNAL_STORAGE`) to read the map from the file system is checked. If the permission is denied, the Activity does not display the in Figure 4.1 shown layout, instead an information that the Activity needs the permission to work as well as a button to request the permission, is displayed.

As soon as the user long-tap a position on the map a **Bottom Sheet** slides up and covers the lower half of the display (See Figure 4.1 b)). The **Bottom Sheet** displays the Latitude

Listing 4.2: Initialize Mapsforge MapView

```
oldLat = getIntent().getDoubleExtra(OLD_LAT_NAME, -1);
oldLong = getIntent().getDoubleExtra(OLD_LONG_NAME, -1);
oldRadius = getIntent().getFloatExtra(OLD_RADIUS_NAME, 0);

if (!(oldLat == -1 || oldLong == -1)) {
 oldPosition = new LatLong(oldLat, oldLong);
 this.oldRadius = oldRadius;
 radius = oldRadius;
 hasOldData = true;
}
...
if (hasOldData) {
 mapView.setCenter(oldPosition);
 markedPostion = oldPosition;
 drawMark();
} else {
 mapView.setCenter(MAP_CENTER_POSITION);
}
```

and Longitude of the tapped position. Furthermore the user has the possibility to adjust the radius by clicking on the number next to the field *Radius in KM*. The upper half of the screen displays the map with view on the tapped location. The radius around this location is coloured red. Therefore the user is able to review his location including the radius. Adjusting the radius cause the red circle to change his radius.

The *LocationChooseActivity* performs its task if the user clicks the *Apply Position Button*4.3. This action creates a new **Intent** and pass all the information from the marked position (*Latitude, Longitude and Radius*) into the **Intent**.

If the **Parent Intent**, the Intent which started the *LocationChooseActivity* used the method `void startActivityForResult (Intent intent, int requestCode)` it will receive a result via the callback method. As soon as callback method is called and the **Integer** `requestCode` matches in both methods, the parent is aware of the end of the *LocationChooseActivity*. If the `resultCode` also matches `Activity.RESULT_OK`, the Activity can handle the received data.

A very important part of the design of the *LocationChooseactivity* was the possibility to be

Listing 4.3: Apply a marked position

```
private void applyPostion() {
 if (isMarked) {
  Intent data = new Intent();
  data.putExtra(LAT_NAME, markedPostion.getLatitude());
  data.putExtra(LONG_NAME, markedPostion.getLongitude());
  data.putExtra(RADIUS_NAME, radius);
  if (oldPosition != null) {
   data.putExtra(OLD_LAT_NAME, oldPosition.getLatitude());
   data.putExtra(OLD_LONG_NAME, oldPosition.getLongitude());
   data.putExtra(OLD_RADIUS_NAME, oldRadius);
  }
  if (getParent() == null) {
   setResult(Activity.RESULT_OK, data);
  } else {
   getParent().setResult(Activity.RESULT_OK, data);
  }
  finish();
 } else {
  Toast.makeText(this, "Mark a position by pressing on a
     location",
   Toast.LENGTH_LONG
  ).show();
 }
}
```

called by Third Party Application. To do so we set up an **Intent Filter** called `de.opptain.waitress.intent.action.CHOOSE_LOCATION`

4.3 Collect User Data

In this section we describe the implementation of the background services which is collecting user data. The service has to maintain the ambitious demands described in Section 3.2.3.

4.3.1 MovementTracker

The class *MovementTrackingService* (MTS) is working in the background and expands the class *Service*. Like an Activity, an **Intent** is able to start and also to stop the service. The task of the service is to collect GPS-based data and to save them into a database. Therefore it creates an instance of the class *GpsPositionManager*. After starting, the *MovementTrackingService* automatically searchs for an initial GPS position.

Getting GPS data

The *GpsPositionManager* handles the GPS data request. There are two ways to listen for GPS data. The first on is the *GpsLocationListener* that uses the Android framework location API which is a native way to receive location based data. All Android Smartphones with GPS sensors are capable of using it. The second one is the *GoogleGpsListener* and it uses the LocationServices API provided by the Google API. In order to use it, the device needs the latest version of **Google Play Service** installed as well as a minimum Android version (2.2). According to Google the number of devices with version 4.x or higher is around 98.40% [Goo17c]. If the latest version of Google Play Service is installed, the *GpsPositionManager* will use the *GoogleGpsListener* or else it will the *GpsLocationListener*. Google's Location Service API is considered the better choice in accuracy or power consuming. [Goo17b].

Both listener are identical structured according methods or field names. As soon as they are started, they are listening for location updates. After an amount of tries (`MAX_TRIES`) or af-

ter receiving an update with an accuracy greater then a pre defined value `MIN_ACCURANCY`, the listener stops listening.

The result, a `Location` object is passed through *MovementTrackingService*'s callback method `void receiveNewGpsLocation(Location location)`.

After *MovementTrackingService* received this first position, the service loop starts. Each loop pass or **cycle** starts the *MovementListener* and has a runtime of one minute, which is set with `delayTimer` in the MTS.

Analyse the Movement

To be accuracy in tracking, the user location should be known in each **circle**. Listening for location updates via GPS sensors is very power consuming. Furthermore it is often not necessary to request location updates, like if the user is staying in a location that is already known. Therefore we implemented a system with different states. The three possibilities of movement are `SLEEPING_STATE`, `WALKING_STATE` or `DRIVING_STATE`. Each state has it own behaviour regarding to location updates. The different states are described in Section 3.2.3.

The class *MovementListener* analyses the user's movement. It is called in the beginning of each circle by the *MovementTrackerService*. It has a callback function to return the result when a specific movement is determined. Likely every other Listener it has methods to be started and stopped. It implements the interface *ConnectionCallback*, a service provided by Google's ActivityRecognitionApi. The API recognizes the user's movement activity by periodically reading short burts of sensor data. The used sensors are all low powered sensors.

Behaviour while being in the Sleeping state

While being in the `SLEEPING_STATE` the *MovementTrackingService* does not request location updates, but it starts the *SignificantMotionDetector*. This class is designed to detect suddenly significant motion changes, like start walk. The class uses the motion sensor `TYPE_SIGNIFICANT_MOTION`.

The sensor TYPE_SIGNIFICANT_MOTION does the work of analysing the values of the sensor TYPE_LINEAR_ACCELERATION, which reads accelerations in $\frac{x}{m^2}$ on each axis. It works as a wake up sensor, which triggers an event after detecting a significant motion. Furthermore it has a very high accuracy of determining whenever the user made a significant motion. For example, moving the device very fast forward and back does not trigger the sensor. TYPE_SIGNIFICANT_MOTION also is one of the least power consuming sensors and therefore it is suitable for the power saving SLEEPING_STATE.

After the significant motion event was triggered, the current state changes to WALKING_STATE.

Behaviour while being in the Driving state

If the *MovementListener* recognizes a driving movement, the state is set to DRIVING_STATE. On each **circle**, while being in this state, the *MovementTrackingService* starts the *GpsPositionManager* to request the current GPS location, because it means that the user is moving fast and to be accuracy a new location is requested in each loop pass. That is why it is the most power consuming state compared to the other states.

Behaviour while being in the Walking state

In the walking state, the MTS has to decide whenever the user is walking away, for example walking down the street or if the user is walking around, for example walking around in a room. In the second case, no GPS location update is necessary, because technically the person is on the same position. To measure if a person is still in the same place we implemented the *PdrPositionManager*. In the second case, where the user walked away, the *GpsPositionManager* starts to request a new position.

Implementation of *PdrPositionManager*

The class *PdrPositionManager* calculates the user's position without using the power consuming GPS sensor. The *MovementTrackingService* is calculating the distance between this

Listing 4.4: Recognise new step

```
private void newStep(float stepSize) {
 lastLocation = locationCalculator.calculateNextLocation(
  lastLocation,
  stepSize,
  directionListener.getDirection()
 );
 parent.receiveNewCalcLocation(lastLocation);
}
```

position and the last known GPS position. This distance is used to decide whenever a person is walking away or is walking around the last known position, within the period time of one **circle**. Each calculated location is passed to the *MovementTrackingService*. In order to calculate a new position we need:

1. **Step Recognition**, which recognizes whenever a step has been taken.

2. **Starting Point** is the location where a user was before he took a step.

3. **Direction Recognition** is needed to determine in which direction the step is taken.

4. **Step Length** is setting the distance a person moved after taking one step.

5. **Location Calculator**, calculates the new location with the parameter from above.

Regarding to *James Oat Judge* [JDO96] an average step length of a young person is about 0.74m and of an old person about 0.65m. We used the arithmetical average of 0.7m for step length. This value is saved in field `STEP_LENGTH`.

The *PdrPositionManager* reads sensor data from `TYPE_LINEAR_ACCELERATION`, to detect whenever a new step was taken. Therefore it implements the `SensorEventListener`, an interface for receiving sensor data. If a value is higher than $\frac{1}{m^2}$, the *PdrPositionManager* recognizes a step and calls the function `void newStep(float stepSize)`4.4 with `STEP_LENGTH` as parameter. The function calculates the new location and passes it to the *MovementTrackingService*.

The class *DirectionListener* also implements the `SensorEventListener`, but it reads data from the sensor type `TYPE_ROTATION_VECTOR`, which delivers the orientation of the device. This information is used to determine in which direction the device is moving.

The class *LocationCalculator* is doing the mathematical part. The *Haversine Formula* is used to calculate geographic distance between two points on earth [Ltd]. If the distance is given, it can be transformed to calculate a new location.

$$\varphi_2 = \arcsin(\sin\varphi_1 \cdot \cos\varphi_1 + \cos\varphi_1 \cdot \sin\delta \cdot \cos\theta) \tag{4.1}$$

$$\lambda_2 = \lambda_1 + \arctan_2(\sin\theta \cdot \sin\delta \cdot \cos\varphi_1, \cos\delta - \sin\varphi_1 \cdot \sin\varphi_1) \tag{4.2}$$

Where φ is latitude, λ is longitude, θ is the bearing (clockwise from north), δ is the angular distance d/R; d being the distance travelled, R the earth's radius [Ltd].

The *LocationCalculator* implemented the *Haversine Formula* 4.5 and returns a new *Location*.

4.4 Database

We start this section with a short introduction of databases. After that we will introduce the structure of the implemented database and implement a class called *LocationDBHelper*, which helps to interact with the database. In the end of this section we will implement the class *PersonalLocationAanalyzer* that analyses the collected data.

A **Database** is a structured collection of persistence data and is used to save information outside of the runtime of an application. **Database Management System** (DBMS) is a computer software and the interface between the application and the database. **Statements** are commands to interact with databases. They are used to add, delete, filter or read data from the database.

In this section we will implement a **DBMS** called *SQLite* and use it to save the collected Location-based data. *SQLite* is a lightweight DBMS and is able to interact with databases without using a server software. It is also recommended by Android to be used for Android

Listing 4.5: Calculating a new location

```
private static final int EARTH_RADIUS = 6371000;

public Location calculateNextLocation (
   Location currentLocation, float stepSize, float direction
) {

 Location nextLocation = new Location(currentLocation);
 float distance = stepSize / EARTH_RADIUS;
 double oldLat = currentLocation.getLatitude();
 double oldLng = currentLocation.getLongitude();
 double newLat = Math.asin(
   Math.sin(Math.toRadians(oldLat)) * Math.cos(distance) +
   Math.cos(Math.toRadians(oldLat)) * Math.sin(distance) *
      Math.cos(direction)
 );
 double newLon = Math.toRadians(oldLng) +
   Math.atan2 (
     Math.sin(direction) * Math.sin(distance) *
      Math.cos(Math.toRadians(oldLat)),
     Math.cos(distance) - Math.sin(Math.toRadians(oldLat)) *
        Math.sin(newLat)
   );

 nextLocation.setLatitude(Math.toDegrees(newLat));
 nextLocation.setLongitude(Math.toDegrees(newLon));
 return nextLocation;
 }
```

Applications, because Android provides full support for SQLite databases [Goob]. These databases are not accessible by Android's file system, only by the application.

4.4.1 Structure and Content

In this section we will show the structure of the database and which data are saved. First of all, each database entry represents only one location. Therefore we save the *Latitude* value and the *Longitude* value of this location. Together they form a location. The value *Counter* represents the time the user stayed in this location. Therefore it is *1* when the entry is created and will be incremented each time the user is staying at the location.

SQLite statements are not capable of using cosine or sinus functions, but the latitude and longitude values of these functions are required for several calculations, like calculating the distance (See Haversine Formula 4.1). Therefore these values are pre calculated and saved into the database.

The location database contains following columns:

- **id**, an unique number to clearly identify each database entry
- **lat**, the *Latitude* value of the location
- **long**, the *Longitude* value of the location
- **accuracy**, shows how precise the location determination was
- **timestamp**, the *Time* of the location determination
- **counter**, *Duration*, in circles, the user stayed at the location
- **cos_lat_rad**, $\cos(Latitude \cdot \pi \div 180)$, calculated cosine value of *Latitude*
- **sin_lat_rad**, $\sin(Latitude \cdot \pi \div 180)$, calculated sinus value of *Latitude*
- **cos_long_rad**, $\cos(Longitude \cdot \pi \div 180)$, calculated cosine value of *Longitude*

Listing 4.6: Creating the Location Database

```
private static final String SQL_CREATE_ENTRIES =
 "CREATE TABLE " + LocationEntry.TABLE_NAME + " (" +
  LocationEntry._ID + " INTEGER PRIMARY KEY," +
  LocationEntry.COLUMN_NAME_LAT + " FLOAT," +
  LocationEntry.COLUMN_NAME_LONG + " FLOAT," +
  LocationEntry.COLUMN_NAME_ACCURACY + " FLOAT," +
  LocationEntry.COLUMN_NAME_TIMESTAMP + " TIMESTAMP," +
  LocationEntry.COLUMN_NAME_COUNTER + " INTEGER," +
  LocationEntry.COLUMN_NAME_COS_LAT_RAD + " FLOAT," +
  LocationEntry.COLUMN_NAME_SIN_LAT_RAD + " FLOAT," +
  LocationEntry.COLUMN_NAME_COS_LONG_RAD + " FLOAT," +
  LocationEntry.COLUMN_NAME_SIN_LONG_RAD + " FLOAT)";

@Override
public void onCreate(SQLiteDatabase db) {
 db.execSQL(SQL_CREATE_ENTRIES);
}
```

Listing 4.7: SQLite examle to increase counter value

```
UPDATE 'counter' SET 'counter' = 'counter' + 1 WHERE _id = 5
```

- **sin_long_rad**, $\sin(Longitude \cdot \pi \div 180)$, calculated sinus value of *Longitude*

The accuracy of the location and the time, when the location was determined, are both values that are also saved into the database. These values are not used in the implementation of this thesis, but these are values that will be used in future work.

4.4.2 LocationDBHelper

The class *LocationDBHelper* extends the android class *SQLiteOpenHelper*. It is used by the *MovementTrackingServie* to write data into the database. It overrides the inherited `void onCreate()` method, which is shown in Listing 4.6. This method is only called, if the application initialized *LocationDBHelper* and no database for the application was found. The class *LocationEntry* contains fields for the column names.

The class provides two methods that are used by the *MovementTrackingService*. The first method creates a new database entry. It requires a *Location* as parameter and returns the *id* of the new created database entry, which is saved to the field `lastDatabaseRow`.

```
long addNewEntry(Location location)
```

The second method is used to increment the counter value of the last database entry and is called, whenever the user stayed in range of the last known location.

```
void incrementLastEntry(long lastDatabaseRow)
```

The value of `lastDatabaseRow` is passed as a parameter. The Listing 4.7 shows an example of the *SQLite* statement, which increments the counter value of the database entry with the *id* of *5*.

Furthermore the class contains a method to read data from the database. This method is used by the *PersonalLocationAnalyzer* and returns a **HashMap** of *Location* objects. The coordinates of these locations lay within a radius of a location. The radius and the location are passed as parameter.

```
HashMap<Location, Integer> getNearLocations(Location location,
double distance)
```

The following equation shows the used mathematical function to calculate the distance between two locations $Location_1$ and $Location_2$. The function is a transformed version of the Haversine Formula 4.1.

$$\cos Distance = \sin Lat_1 \cdot \sin Lat_2 + \cos Lat_1 \cdot \cos Lat_2 \cdot \sin Lat_1 \cdot \sin Lat_2 + \cos Lat_1 \cdot \cos Lat_2 \tag{4.3}$$

If we multiply the result by 6380, the earth radius in kilometres, we get the distance between the locations in kilometres.

$$Distance\ in\ km = \arccos(\cos Distance) \cdot 6380 \tag{4.4}$$

We added a condition to determine if $Location_2$ is in the *Radius* of $Location_1$:

$$\cos(Radius \div 6380) > \cos(Distance) => Location_2\ is\ in\ Radius\ of\ Location_1 \tag{4.5}$$

The method uses Condition 4.5 to get all the location we are searching for. In the next section we implemented a class to analyse the locations.

Listing 4.8: Analyse location

```
private static final double WEIGHT_NODES = 0.3;
private static final double WEIGHT_COUNTER = 0.7;
private static final int MIN_REQUIREMENT = 1;

public static boolean analyze(Location location, double
    radius) {
 LocationDBHelper dbHelper = new LocationDBHelper(mContext);
 HashMap<Location, Integer> locationIntegerHashMap = dbHelper
     .getNearLocations(location, radius);

 if (locationIntegerHashMap.isEmpty()) {
  return false;
 }
 int counter = 0;

 for (Location loc : locationIntegerHashMap.keySet()) {
  counter += locationIntegerHashMap.get(loc);
 }
 return WEIGHT_NODES * locationIntegerHashMap.size() +
   WEIGHT_COUNTER * counter
  > MIN_REQUIREMENT;
}
```

4.4.3 PersonalLocationAnalyzer

The class *PersonalLocationAnalyzer* has the task to analyse a set of locations and convert the result into a *value*. This value represents the possibility that the user is entering the radius of a location. The method in Listing 4.8 takes the part of analysing. It takes two parameters. The first one is a *location* and the second one is a *radius*. In the beginning, the method requests all locations that are within the *radius* of *location*.

This implementation is a basic analysing of the locations within the database. We used a possible approach of analysing the data. The approach used the number of locations as well as the sum of all counter values. Both numbers are multiplied with a possible weight, which could represent the significance of each number, but if one database entry is within the *radius*, the *PersonalLocationAnalyzer* already returns true.

In the future, the algorithm should be enhanced to use more collected data. For example it

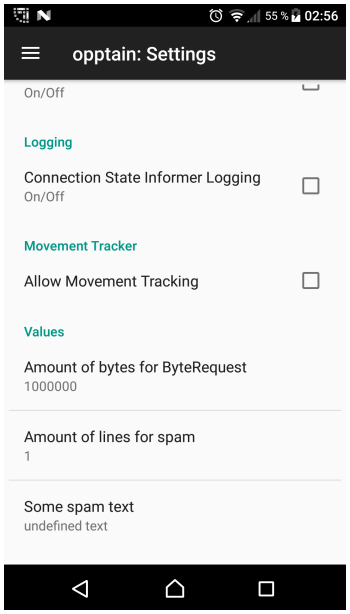


Figure 4.2: Option to active/deactivate the *MovementTrackingService*

should also include information like the *Timestamp*, to include the possibility to calculate when the user could reach the destination.

4.5 Opptain Integration

In this section we will integrate all the implementations, we worked on in the previous sections, into the main application *Opptain*.

4.5.1 User Settings

The user should be able to active/deactivate the *MovementTrackingService*. Therefore we expanded the settings of *Opptain* and add a new check box. Figure 4.2 shows the new added option.

If the check box is selected, the *MovementTrackingService* starts and stops if the check box is deselected. Figure 4.9 shows the implementation of the *OnChangeListener* that was added to the check box. This listener is triggered whenever the user clicks on the option.

Listing 4.9: Implementation of new settings

```
private class
 ButtonAllowMovementTrackerConfigurationOnPreference-
  ChangeListener
 implements Preference.OnPreferenceChangeListener
{
 @Override
 public boolean onPreferenceChange(Preference preference,
    Object newValue) {
  boolean value = (Boolean) newValue;
  if (value) {
   getActivity().startService(new Intent(getActivity(),
      MovementTrackingService.class));
  } else {
   getActivity().stopService(new Intent(getActivity(),
      MovementTrackingService.class));
  }
  return true;
 }
}
```

An **Intent** is used to stop and start the service. If the *MovementTrackingService* is enabled, it automatically starts with *Opptain*.

4.5.2 Network Bundle

The following implementation was made in the *Third Party Application* **Ping**. This application is able to send a ping to a device within the *Opptain Network*. We expanded this application to add a destination and a radius into *Optain*'s network bundle. Listing 4.10 shows how the data are added into the *AdditionalInformation* object. This object is part of the **Meta Data** described in Section 3.2.1.

After the user clicks the *Send Button* in **Ping**, a dialogue opens, where the user can decide, if he wants to add a destination. Clicking *Yes* starts the *LocationChooserActivity*. The values of *lat*, *lng* and *rad* were set by the user via the *LocationChooserActivity*. This action is only performed by the creator of the *Network Bundle*.

Listing 4.10: Add additional information

```
AdditionalInformation additionalInformation = new
    AdditionalInformation();
if (lat != null && lng != null && rad != null) {
  additionalInformation.put("LOCATION_ROUTING_LAT", lat);
  additionalInformation.put("LOCATION_ROUTING_LNG", lng);
  additionalInformation.put("LOCATION_ROUTING_RAD", rad);
}
```

4.5.3 Device Communication

In this section we will integrate the *PersonalLocationAnalyzer* into the *Device Communication*, described in Section 3.2.1. Listing 4.11 shows how the *Latitude*, *Longitude* and the *Radius* are extracted from the *Additional Information*.

These values are passed into the *PersonalLocationAnalyzer* which returns a boolean value. If this value is false or the *Additional Information* object does not contain GPS-based data, the *Device Communication* ignores the *GPS-based Routing* extension.

Listing 4.11: Extract GPS data from Additional Information

```
boolean gpsWanted = false;

//check if additional information contains GPS-based data
if ((peerCopyOfOpptainBundle.getAdditionalInformation()
 .get("LOCATION_ROUTING_LAT")!=null)
 && (peerCopyOfOpptainBundle.getAdditionalInformation()
 .get("LOCATION_ROUTING_LNG")!=null))
{
 Location destination = new Location("");
 destination.setLatitude((double)peerCopyOfOpptainBundle
  .getAdditionalInformation()
  .get("LOCATION_ROUTING_LAT"));
 destination.setLongitude((double)peerCopyOfOpptainBundle
  .getAdditionalInformation()
  .get("LOCATION_ROUTING_LNG"));
 Double radius = (Double)peerCopyOfOpptainBundle
  .getAdditionalInformation()
  .get("LOCATION_ROUTING_RAD");
 gpsWanted = PersonalLocationAnalyzer
  .analyze(destination, radius);
}
```

Chapter 5

Evaluation

In this chapter we want to present the results of testing the developed extension. In the first part we will evaluate the *MovementTrackingService* which has the task to collect the **User Data**. After that we will evaluate the implementation of the GPS-based Routing. In the end we will analyse the power consuming.

5.1 User Data

In this section we want to evaluate the tracking of User Data. We begin with analysing the different parts of the *MovementTrackingService* and end with the evaluating the collected data.

5.1.1 Movement Recognition

First we want to show the results of evaluating the movement recognition implemented in section 4.3.1. This service provides information about the user's current movement. Therefore we prepared a test. The test should show how accuracy the movement recognition works.

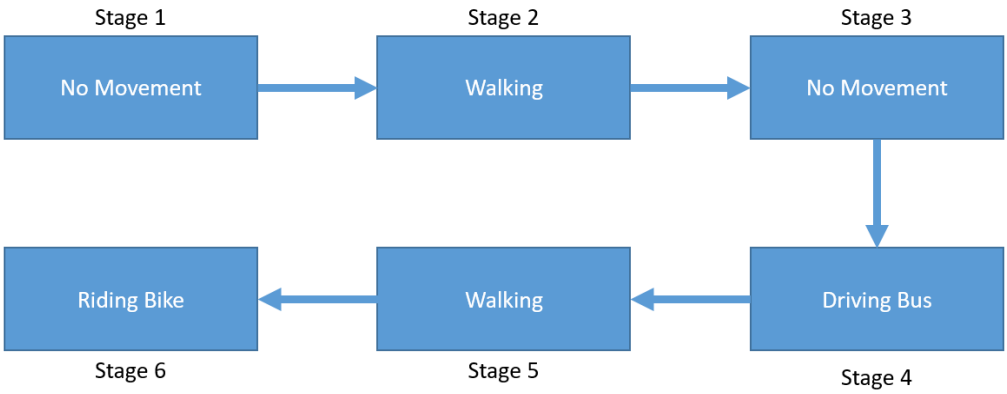


Figure 5.1: Movement Recognition Test Setup

Test Setup

In order to test the movement recognition we modified the extension to log the calculated state together with a timestamp. Following movements were tested:

- standing still, should be recognized as `SLEEPING_STATE`
- walking, should be recognized as `WALKING_STATE`
- driving by bus, should be recognized as `DRIVING_STATE`
- driving by bicycle, should also be recognized as `DRIVING_STATE`

We planned a scenario to cover all the the movements in one run. So we were able to analyse the recognition of a change in the movement, too. In this test were two people involved. The first person was the tester. He used to carry the test device. To be realistic the person used the device for various actions, like listening music, texting or carrying the device in the pocket. The other person documented each part of the test and prepared the different stages.

In the first stage we tested the `SLEEPING_STATE`. The person was sitting with the device in his hand for 20 cycles. Each cycle takes one minute. The user stood up and started walking. There the second stage started. The tester walked down a street without stopping to test the recognition of `WALKING_STATE`. The possibility to recognise that the person is walking around a location, without leaving the location will be tested in the next section and was not

Table 5.1: *Movement Recognition* test result

Movement	Time in Minutes	correct	incorrect	Success Rate in %
no movement	28	26	2	92.85
walking	24	20	3	83.33
driving by bus	23	19	4	82.60
driving by bicycle	20	20	0	100

part of this test.

The tester carried the device in his hand as well as in his pocket. He also varied in his walking speed. After walking for 20 cycles he arrived at a bus station and waited there for another 8 cycles before he got in the bus to test the recognition of `DRIVING_STATE`. While sitting in the bus the tester conventional used the device. This stage took 23 cycles.

In the last stage we also tested the `DRIVING_STATE`, but we used a bicycle instead of the bus. Therefore the person who documented the test prepared a bicycle at the bus station. The tester walked 4 cycles between the two stages.

While riding the bike, the device was carried in the pocket of the tester. After 20 cycles of the stage and the test ended. All in all the tester was standing still for 28 minutes, walking for 24 minutes, driving with the bus for 23 minutes and riding the bike for another 20 minutes. In total he changed his movement six times and the test took 95 minutes.

Test Results

In this section we will present the result of the previous test and evaluate it. The Table 5.1 shows the result. The different stages are generalized into the different movements and not shown in the table. Furthermore the table shows each movement in minutes, parted into correct identified and incorrect identified movements. In the following we will analyse the result of each movement. The tester was not moving in two stages. In the first stage he was sitting 20 minutes. Each minute was correctly recognised as `SLEEPING_STATE`. The second time, when the device was not moving was as the tester was waiting for the bus in stage 3. In the first cycle, while waiting, the *MovementTrackingService* wrongly identified the movement as `WALKING_STATE`, which was the first mistake caused by the class

MovementListener. While waiting for the bus, the tester was using the device and holding it without walking around, but for one circle the *MovementTrackingService* was falsely set into `WALKING_STATE`.

The second row of the table shows the result for *Walking*. The state immediately changed from `SLEEPING_STATE` to `WALKING_STATE` after the tester started walking in stage two. While walking in stage two the *MovementListener* correctly identified the state as `WALKING_STATE`, except for one minute. In one cycle the state was set to `DRIVING_STATE`. At this point the tester was walking while holding the device in his hand. The other two incorrectly identified movements where the state was not `WALKING_STATE`, were in stage 5. After the tester went off the bus, the state was wrongly set to `DRIVING_STATE` for two more cycles. Another important notice we can take from result, is that the *PdrPositionManager* works very accurate for walking straight. In every circle, where the tester was walking, the *PdrPositionManager* determined a distance greater than the radius of 10. The average distance the tester walked within one minute was around 80 meter.

The next row shows the results while driving bus. In total the tester was sitting for 23 minutes in the bus. After getting in the bus in stage 4, the *MovementListener* correctly and imminently identified the movement as *Driving*. In four cycles the movement were falsely identified as `WALKING_STATE`. All of them were moments, when the bus was speeding up, after waiting at a traffic light or a bus station for a moment. Every time the bus was waiting longer than a minute.

The last row shows the result of the *MovementListener* while riding with the bike, which were perfectly recognized as *Driving*. The tester was riding bike for 20 minutes and all of them were correctly identified, although the tester had to wait at a traffic light for nearly 45 seconds.

All in all the calculations of *MovementListener* are very accuracy. Nearly 89.70% of all a result were correct. The rate of 92.85% as result for identification the `SLEEPING_STATE` is a great success, because this state is the power saving state. It does not request GPS position and for that should not be wrongly identified.

(a) test setup

Test	trails	successful	%
walking within the circle	25	15	60
walking out and in	25	18	72
walking out the circle	25	22	88

(b) test result

Figure 5.2: *Pedestrian Dead Rocking* accuracy test

5.1.2 Pedestrian Dead Rocking

In the last section we evaluated the **Movement Recognition**, but ignored the possibility that the user could walk within a radius of the last known position. This scenario could happen, when the user is walking around in a room, without leaving it. We implemented a class called *PdrPositionManager* to calculate the position of the user without using GPS data. We needed this function to determine whenever the user is outside a specific radius in the end of a cycle. The center point of this radius was the last known GPS position. In order to test the exactness of the calculated location we planned a test.

Test Setup

For testing the *PdrPositionManager* we modified the implementation to notify the user in the end of a cycle. This notification had two possible outputs. If the calculated position is within the radius it shows *IN*, or else it shows *OUT*. The test location was a big empty place. We marked a position and used chalk to draw a circle with a radius of 10 meter around this position. Figure (a)5.2 shows the setup. The drawn circle was the line between being inside and outside the radius. Furthermore we created an Android Activity with only a button to start the *MovementTrackingService*, that was also modified to only get the current GPS position and after that it started the *PdrPositionManager* for one circle. Therefore we

could test the *PdrPositionManager* only by pressing the button. We tested three scenarios. In the first scenario the tester walked within the circle. In the second one the tester walked in and outside the circle, but the ended within. In the last scenario the tester was outside the circle in the end of one minute. All scenarios were tested 25 times. In the beginning of each test the tester stand in the center of the radius and pressed the button. After receiving the current GPS position, which was used as a start point to calculate the first step, the *PdrPositionManager* started and the tester walked exactly thirty steps. Furthermore the tester walked each time a different way. We used thirty steps for this test, because the tester used to count his steps while walking in his flat. In total walking time, the tester used to make an average number of 30 steps within a minute. We did not use a new GPS position to check if the calculated position is accurate, because both GPS positions, the initial position as well as the end position, would have needed to be very high accuracy value to be reliable as comparison value for the calculated position.

Test Result

Table 5.2(b) shows the test result. The first thing we noticed is that the *PdrPositionManager* often calculates more steps than we took. The average number of steps that we recognised were around 33 steps. This problem is caused by the class *PdrPositionManager* itself, because it determines, based on the acceleration sensor, whenever a step was taken. Adjusting the minimum value of acceleration that is necessary to trigger the step event, could solve this problem.

In the first scenario the tester walked within the radius. In 60% of the tests, the *PdrPositionManager* determined the right result. The result strongly depends on the route the tester had walked. A route, where the tester often walked through the center of circle, was often determined with the right result. As a result of this test you can say that the *PdrPositionManager* works more precisely when the direction of a few steps point towards the start point. Thus, walking at the edge of the circle mostly ended with the wrong result.

The second scenario, where the user is walking inside and outside, but ended in the circle, the success rate was 72%. This result supports the thesis, that walking towards the circle increases the chances of determining the right result. Another thing we noticed was, that whenever the result, of a route (*a*), was right and we nearly walked the same route (*b*), but further away from the outermost point of route (*a*), route (*b*) also had the right result. That

could indicate that the *PdrPositionManager* is able to notice very well, whenever the user is turning around and is walking back.

The last scenario covers the case, where the user is walking inside and outside, but is not walking back into the circle. In this scenario we received the best results. 88% of the decisions were correct. In 3 tests the *PdrPositionManager* calculated a wrong result and all of them had one thing in common. The last step, in each of the three wrong identified routes, was very close, around 1 meter, to the edge of the circle. Therefore the result is more accurate if the user is walking far. This recognition covers with the results we made in the previous chapter, where walking straight was every time determined right by the *PdrPositionManager*.

The average success rate is around 75.33%. In order to increase this rate, two parts of the *PdrPositionManager* needs to be adjusted. First we should fix the problem, that more steps are recognised than taken. From that bug a greater distance is recognised by the device than the user walked, which results in a greater calculated distance. Often the extra distance made the *PdrPositionManager* decide that the user is outside the radius. The second factor that could increase the success rate, is a custom step length. The fixed value for the step length is set to 70cm. The tester had a step length around 76cm. The total deviation is increased by the discrepancy on each step.

Conclusion

The *MovementTrackingService* works very accurate. It is able to determine whenever a person is staying at a location, regardless of the person's current movement. It therefore follows that the number of circles, when the *MovementTrackingService* requested a GPS-based location update, enormous decreased. A day has 86,400 minutes. Without the implementation of *PdrPositionManager* and *MovementListener*, the *MovementTrackingService* would start to request the GPS data every minute. After analysing the results of the next section 5.2, the number of GPS request strongly depends by the user's movement. On average days the total number of location updates were decreased to 150 requests. That means the *MovementTrackingService* decreased the number of GPS request by 99.99%.

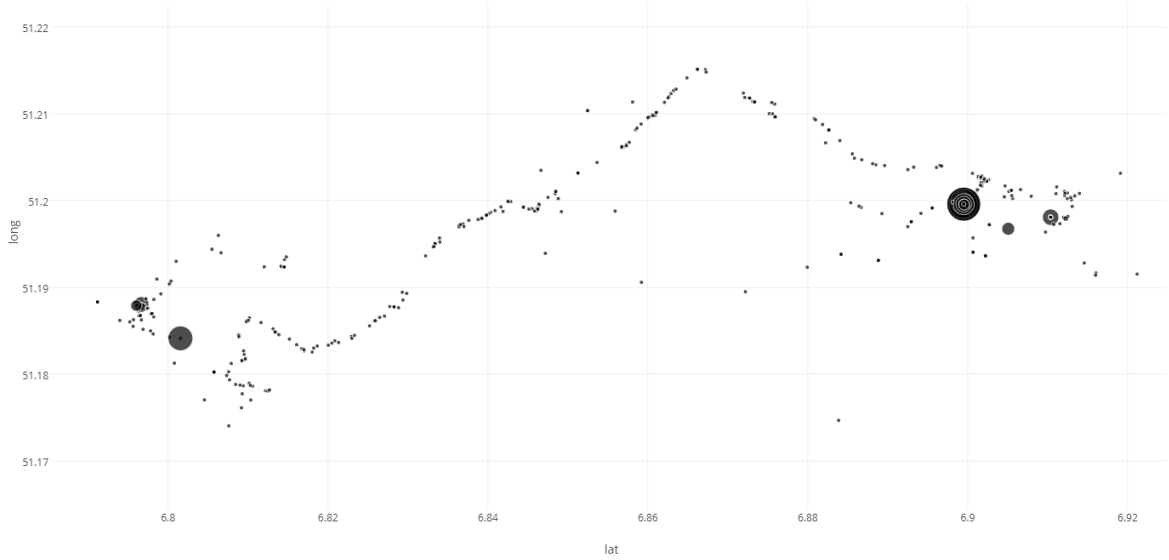


Figure 5.3: Collected data after three days

5.2 Evaluating the Data Collection

In this section we are going to evaluate the collected data from the *MovementTrackingService*. In the previous section we already evaluated the main components of it. In order to analyse the data, we exported the database from the test device, on which the *MovementTrackingService* was running for three days. Within these days the tester was in the university, at home or driving between those two spots by bus.

In this period of time, the service wrote *443* database entries. For this evaluation we only look at the columns **Latitude**, **Longitude** and **Counter**. The first two values are making the coordinates of the location and the third value represents how long, in minutes, the user stayed in this location. Figure 5.3 shows a plot of the mentioned data. The x axis represents the **Latitude** and the y axis represents the **Longitude**. Therefore a location is displayed as a point and the size of this point represents its duration time. The plot shows two area with a high density of points that are above-average size. In Figure 5.4 these areas are marked as *Area A* and *Area C*. We will start by evaluating this two areas. *Area A Area A* contains a few bigger points. At those locations the tester stayed for a longer period of time. Furthermore there are a lot of little points. That means that the user is also moving around in this area. In reality, this area represents the university where the tester is studied. We imported some of the points into Google Maps and the tester was able to verify the marked locations

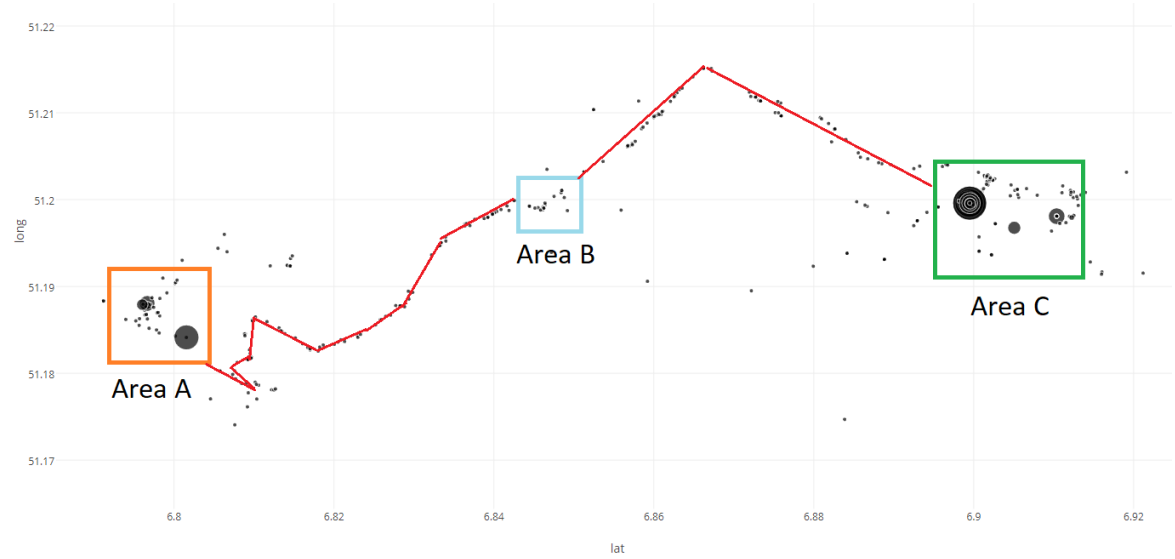


Figure 5.4: Added additional information to Figure 5.3

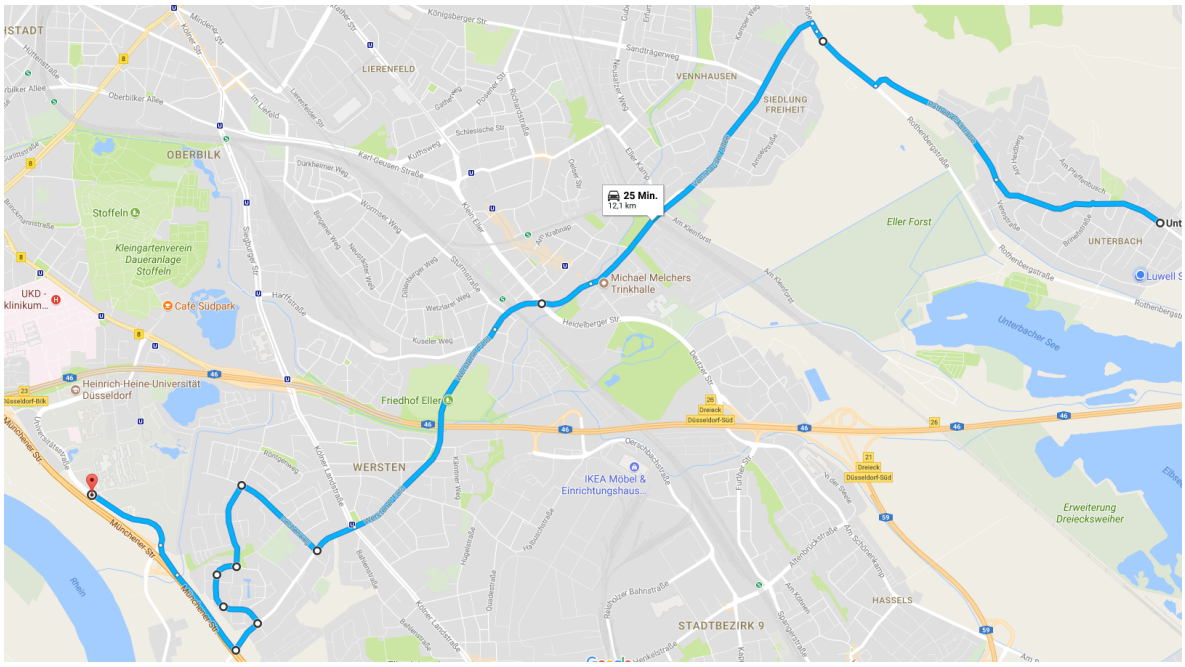


Figure 5.5: Bus Route (Source Google Maps)

Area C contains a lot of big points. Most of them are overlapping each other. This indicates that the tester stayed there for a long time and that he was there very often. In reality, the tester is living in this area and he could also verify the accurate of this data.

It seems that the other dots are trying to draw a line between those two areas by overlapping each other. We highlighted this possible line with the red line in Figure 5.4. A high density of points means that the tester was often around this points. The tester could verify that he often moved between *Area A* and *Area C*, because this was the route of his bus. In the three days, when the *MovementTrackerService* was enabled, he took the bus 7 times. Figure 5.5 shows the bus route. The red line from Figure 5.4 does look very similar, compared to the bus route in Figure 5.5.

In *Area B* are the points a little bigger than the points on the red line. The tester explained, that in this area is a bus station, where the bus driver is taking a little pause. Sometimes up to five minutes, which cause the points to grow bigger.

In conclusion it can be said that the *MovementTrackerService* is very accurate tracking the users movement and that the number of GPS position requests is linked with the user's movement. The high density of points around the red line and the background information clearly shows that fast movements require a lot of location updates.

To adjust the duration of a cycle while being in the driving state could decrease the number of position request. This could be part of future work.

5.2.1 Battery Usage

We designed the *MovementTrackerService* to be as power saving as possible. The results of reducing the number of GPS requests was calculated in the conclusion of Section 5.1.2. With the introduction of the different states the number of request could be decreased nearly 99.99%, which is very high.

In this section we want to evaluate the actual battery usage. In the three days, when the service was enable, the *GpsPositionManager* requested *443* locations. Nearly *150* requests per day. The device that we used was the **Sony Xperia Z5 (E6653)**. this device is capable of measuring the battery usage for each application. Our extension was requesting GPS position

for nearly 2.5 hours per day and nearly used 220 mAh per day. The total battery capacity of the device is about 2900 mAh. That means that the extension is using 7.6% when the battery is fully charged.

In conclusion we can say that the *MovementTrackerService* is not using too much power and while evaluating the implementation we found several possibilities to decrease the battery usage even more.

Chapter 6

Conclusion and Future Work

In the last chapter of this thesis we will summarize the conclusion in Section 6.1 and we will look at the possible future work in Section 6.2

6.1 Conclusion

The evaluation showed that *MovementTrackingService* as well as its components *MovementListener* and *PdrPositionManager* are providing accurate data. Therefore we were able to reduce the GPS location updated to a minimum.

An important aspect that was not mentioned earlier is the preserving of privacy. The approach of the GPS-based routing shows that no private data are exchanged between devices. Each device holds the collected data and does not share with any system. Androids file system is also not able to extract information from the database or the database itself.

6.2 Future Work

As we mentioned in the motivation, the thesis was to lay a foundation. The extension does not include a proper algorithm for the **GPS-based Routing**. Future work could include a

self-learning algorithm that is trying to predict, whenever accepting a *Bundle* would help the *Bundle* get closer to its destination.

The *MovementTrackingService* could also be complemented by a learning algorithm. After determining daily routines, the service could stop tracking duplicated data and would only recognises changes in this routine. This could drastically decrease the daily GPS requests for users that are moving fast for a long period of time. Like driving the same long track each day to work.

Another part that could be adjusted in the future is the *PdrPositionManager* by adding filters like the **Kalman-Filter** to reduce wrong calculated locations.

Bibliography

[Gooa] GOOGLE: *Sensors Overview*. https://developer.android.com/guide/topics/sensors/sensors_overview.html. [Online; accessed 26-June-2017]

[Goob] GOOGLE: *Storage Options - Using Databases*. https://developer.android.com/guide/topics/data/data-storage.html#db. [Online; accessed 26-June-2017]

[Goo17a] GOOGLE: *Google Maps APIs Term of Service*. https://developers.google.com/maps/terms. Version: 2017. [Online; accessed 21-June-2017]

[Goo17b] GOOGLE: *Making Your App Location-Aware*. https://developer.android.com/training/location/index.html. Version: 2017. [Online; accessed 21-June-2017]

[Goo17c] GOOGLE: *Platform Versions*. https://developer.android.com/about/dashboards/index.html. Version: 2017. [Online; accessed 21-June-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

[JDO96] JUDGEROY, James O.; DAVIS, B. III; OUNPUU, Sylvia: Step Length Reductions in Advanced Age: The Role of Ankle and Hip Kinetics. In: *The Journals of Gerontology: Series A* 51A (1996), Nr. 6, S. M303.

[Ltd] LTD, Movable T.: *Calculate distance, bearing and more between Lati-*

tude/Longitude points. http://www.movable-type.co.uk/scripts/latlong.html. [Online; accessed 23-June-2017]

[Map17a] MAPSFORGE: *Specification: Mapsforge Binary Map File Format*. https://github.com/mapsforge/mapsforge/blob/master/docs/Specification-Binary-Map-File.md. Version: 2017. [Online; accessed 21-June-2017]

[Map17b] MAPSFORGE: *Vector map library written in Java - running on Android and Desktop*. http://mapsforge.org/. Version: 2017. [Online; accessed 21-June-2017]

[Ope17a] OPENSTREETMAP: *Basic components of OpenStreetMap's conceptual data model of the physical world*. http://wiki.openstreetmap.org/wiki/Elements. Version: 2017. [Online; accessed 21-June-2017]

[Ope17b] OPENSTREETMAP: *OpenStreetMap is a map of the world*. https://www.openstreetmap.org/about. Version: 2017. [Online; accessed 21-June-2017]

[Ope17c] OPENSTREETMAP: *Start Mapping*. https://www.openstreetmap.org/export. Version: 2017. [Online; accessed 21-June-2017]

[Osm17] OSMOSIS: *Osmosis is a command line Java application for processing OSM data*. http://wiki.openstreetmap.org/wiki/Osmosis. Version: 2017. [Online; accessed 21-June-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. Juli 2017 Justin Marks

Diese CD enthält:

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