OppMap: Opportunistic Map and Location Sharing

Semester Thesis

Reto Da Forno and Benjamin Dissler

February, 2013

Advisors: **Sacha Trifunovic**
Supervisor: **Prof. Dr. Bernhard Plattner**

Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

Smartphones are heavily used to localize its users on a map and to guide them to a destination. The most prominent application that offers this service is Google Maps. However, these services usually access map data on demand, thus requiring Internet access. Unfortunately, when a user is abroad and the map and localization service gains in importance, Internet access usually results in expensive roaming fees.

Our android application "OppMap" allows users to share map material over an oppurtunistic link for free. Under ideal conditions it's sufficient to pass by another OppMap user that has the surrounding maps stored on the smartphone. The maps around his current location will be automatically exchanged in the background within less than a minute without being noticed by the users. OppMap improves the availability of maps wherever no free Internet access is available.

Contents

Chapter 1

Introduction

1.1 Motivation

Today most map applications on smartphones need an Internet connection to get the map material. If a smartphone user is abroad, charges for data usage for the cellular network are usually expensive and open wireless access points with free Internet are rare. But in an airport, rail station or city, there will be a lot of other smartphone users and since map applications are popular, some of them already might have downloaded or cached the local map material. The idea of the android application “OppMap” is to allow smartphone users to share already downloaded map material without an Internet connection.

1.2 Concept

To connect two smartphones OppMap uses WLAN-Opp to create an opportunistic link. With the ability of WLAN-Opp, mobile phones can exchange data directly or via an access point. It takes advantage of the assumption that many public access points only require login data for Internet access, but allow clients to connect and communicate with each other.

Every user of OppMap declares a home location and stores the map tiles around that location permanently. Whenever an Internet connection is available, all map material required to display is downloaded through that link and stored in the cache.

Whenever two users are connected over an opportunistic link, they send map requests to each other (if they do not already have the map tiles around the current position) and the missing map tiles will be exchanged automatically in the background. Just the map material around the current location will be shared, but no other maps which might be present in the cache.

Chapter 2

Related Work

2.1 WLAN-Opp

Our app is based on WLAN-Opp, a platform that enables opportunistic networks based on the AP mode of the 802.11 standard [1]. As other ad hoc communication protocols such as Wifi Ad-Hoc, Bluetooth, or Wifi Direct are unavailable or unsuited to enable opportunistic networking, WLAN-Opp is a welcome addition to the Android platform, enabling energy efficient peer-to-peer communication without the need of a network infrastructure. The platform was implemented on Android by Meliopoulos and Sheikh [2]. WLAN-Opp achieves opportunistic connectivity without Wifi Ad-Hoc or the Bluetooth protocol by using either an open public access point or the tethering ability of the smartphone. In the latter, one phone takes the role of the access point and becomes a WiFi hotspot, enabling other devices to connect. The WLAN-Opp platform exports an easy API to empower other applications with opportunistic communication.

2.2 OSMDroid - Map API

The first idea was to build an Android application which runs in the background, searches the SD card for map tiles (caches of different apps including Google Maps) and shares them via an opportunistic network. However, the Google Maps cache is saved in a proprietary file format. We couldn't find any hint on how Google saves map tiles in the cache of their own application. This would have been very useful as everybody is using the Google Maps app today. We did not want to reverse engineer the binary cache files, it would take up too much time.

An other approach to distribute map material is to use vector graphics. Vector graphics are better for zooming without loading new data every time, as it would be the case with map tiles. And it needs less disk space for the same map area. The newest Google Maps API uses such vector graphics, but we didn't find any API which uses those and allows us to cache it.

For this reason we decided to build our own application for viewing maps with an own cache. Google provides an easy to use API[3] for their maps, but the problem stays the same: There are no built-in functions to control the cache. That's why we decided on using OSMDroid[4] instead which is open source, supports different map sources and allows full access to the cache.

The OSMDroid API provides tools for Android to interact primarily with OpenStreetMap[5] as map tiles source. There are also other tile sources to choose from, such as OpenCycleMap.org[6] or different OpenStreetMap based sources. And we were able to specify custom sources as well. This is necessary to load official Google Maps tiles. The core of the API is the OpenStreetMapView class, which basically replaces Googles MapView. The source is open and licensed under GNU Lesser GPL[7]. So there would be the possibility to make profound changes in the APIs source.

What the OpenStreetMapView basically does, is to download map tiles from the specified source and to put them together to a bigger, screen filling graphic. Map tiles are pieces of the (world) map, they are square and most sources provide 256x256 pixel tiles. To get a specific map tile you need three coordinates X, Y and Z, where (X, Y) are the 2D map coordinates and Z the zoom level. The bigger Z the more detailed is the map.

Chapter 3

Design

3.1 Feature specification

We decided to concentrate on basic functionality in terms of map control and easy, preferably silent map sharing. The features we specified include:

- A map will be shown to the user.
- Basic controls must be available: displaying the current position and orientation as well as moving and zooming (pinch) the map with touch gestures.
- On the first application start-up, the user should be able to specify his or her home town. All tiles within a certain radius from the center of that town will be downloaded and stored permanently.
- Map tiles shall be automatically transferred in the background. No user interaction needed. Therefore, it must be running all the time without draining the battery too fast.
- The GUI should be kept simple and clean: no fancy title bar, menu or tabs, just an action bar on the bottom of the screen with all basic controls. If possible, use just one view (no nested windows).
- Find a way to get the current position faster. Waiting for a GPS fix might need some time, especially in a town where most of the field of view is occluded by buildings.

3.2 Application Design

Our application consists of four major parts, as seen in Fig. 3.1:

- The **OppMap application** itself, which represents the GUI and uses the OSMDroid API to show the maps. It allows the user to specify his home location and downloads the corresponding map material. All downloaded and shown map material is stored on the device, i.e. in the cache.
- The **OppMap service**, which runs in the background. The service requests missing map material on newly found neighbours (as TCP client)

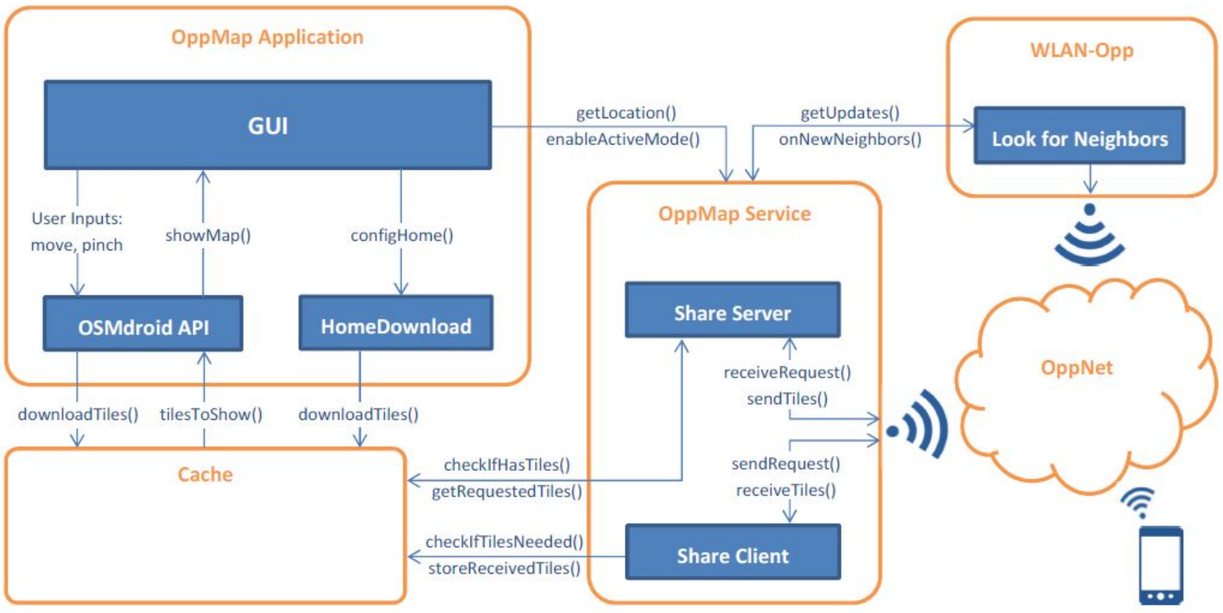


Figure 3.1: OppMap Diagramm

and answers to map requests (as TCP server). It also exchanges location information, on user request.

- The **WLAN-Opp** platform, which searches for new neighbours over opportunistic links, and reports them to the OppMap service.
- And the **Cache**, where all the map material is stored. There the OppMap service looks for maps to share, checks if maps are missing and stores received data from neighbours. Also, the OppMap application searches the cache for maps to display before it loads them over the Internet, if available.

3.3 Location

As mentioned in the features list, another goal was to get the current position within a reasonable amount of time. On the first smartphones with GPS support it usually took up several minutes to get a GPS fix, especially when surrounded by buildings. So the idea was to share not only maps, but also the location to support the GPS module with aiding data. Unfortunately, injecting assistance data proved to be tougher than we initially thought. For security reasons, injecting arbitrary aiding data into the GPS library cannot be done without rooting the device. An application can only force the GPS library to download new aiding data from a list of predefined servers.

We tried to inject GPS data, e.g. the last known location, hoping to speed up the time needed for a GPS fix. As shown in de evaluation (Section 5.3), this unfortunately did not help to accelerate the fix time.

Luckily, GPS modules in modern smartphones are pretty fast. Several cold start GPS fix tests showed that modern chips barely need more than a minute for the first fix. This is astonishing when compared to older devices. Under this assumption one could question the advantage of location sharing if it just takes a minute to get a GPS fix. However, this amount of time applies for ideal situations, but whenever you are walking around in a city, GPS reception is much worse. Due to heavy occlusion of the field of view it might easily take several minutes to get a first fix and of course most GPS modules won't even deliver a location fix indoors. A last but probably not very common case where location sharing is essential, is for devices which do not have a GPS module. When considering these points, sharing the location makes perfectly sense despite the fast GPS fix under ideal condition.

Chapter 4

Implementation

4.1 OppMap Application

In this section we describe the user interface, explain the automated map sharing capability of the service, including the map tile prioritization and discuss two basic aspects of an Android app: Multi threading and the API level.

4.1.1 User interface (GUI)

When the application is started, the user will see the map with a blue blinking cursor at the last known location. If the smartphone has a magnetometer, the cursor direction will show the current orientation of the device. There is an action bar on the bottom of the screen with four buttons: home, active mode, location and settings, as seen in Fig. 4.1. The first time the home button is pressed, the user will be asked if he/she wants to configure the home town. With a comfortable dialogue the home town can be searched by keywords or the map can simply be centred around the desired location. Afterwards, all tiles within a certain radius will be downloaded and stored permanently. This procedure of course requires Internet access.

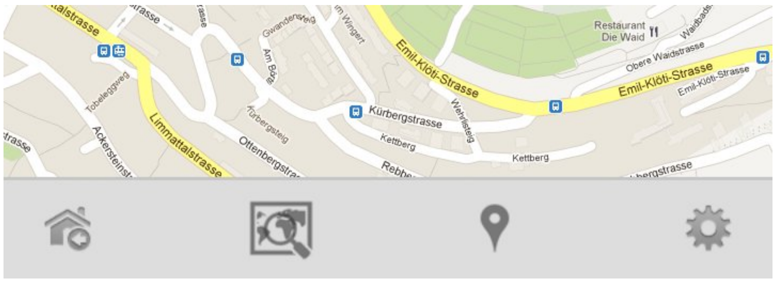


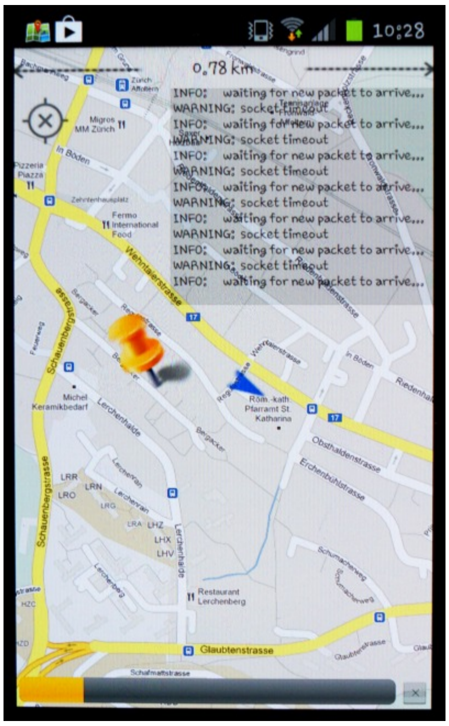
Figure 4.1: OppMap ActionBar

Once the configuration process is done, pressing the home button will just center the map around the home town (marked by an orange pin). The button 'location' causes the application to look for a recent location by sending requests to neighbours and by requesting a GPS and network location update. Once a new location is available, the map will be centred around it. The button ‘active mode’ is used to actively search for new neighbours, i.e. WLAN-Opp will spontaneously change the network settings, switch the access point or even create a WiFi hotspot. If OppMap is not in active mode, the network settings won’t be changed even if there’s currently no neighbour. In this way, the user will most likely not be disturbed and won’t even notice the map exchange. The button ‘settings’ opens the homonymous screen, where the basic configuration of the application can be changed.

On the top left corner, a GPS icon can be spotted. By pressing this icon, the user will be asked if he/she wants to turn the GPS on or (if it is already enabled) whether the GPS module should request continuous position updates. Note that continuous position updates drains the battery faster.

Last but not least, there’s a ruler at the top end of the screen which shows the distance on the map (in kilometres) corresponding to the screen width.

All in all, OppMap has a simple user interface while still providing access to all the necessary functionality.



```
INFO:    waiting for new packet to arrive...
WARNING: socket timeout
INFO:    waiting for new packet to arrive...
WARNING: socket timeout
INFO:    waiting for new packet to arrive...
WARNING: socket timeout
INFO:    waiting for new packet to arrive...
WARNING: socket timeout
INFO:    waiting for new packet to arrive...
WARNING: socket timeout
INFO:    waiting for new packet to arrive...
WARNING: socket timeout
INFO:    waiting for new packet to arrive...
```

Figure 4.2: Screenshot from OppMap

4.1.2 OppMap service - Automated sharing

When the application is started the first time, the OppMap service will be launched and keeps running in the background if the user decides so. This service starts WLAN-Opp and handles all map and location requests. Whenever new neighbours are available and map tiles are missing around the current location,

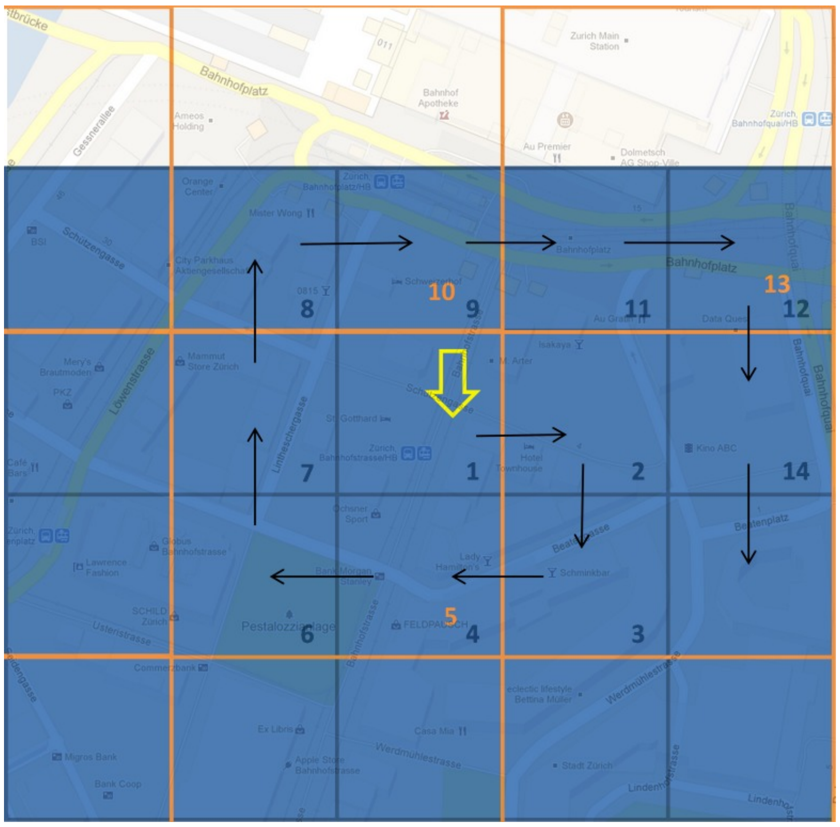


Figure 4.3: map tile prioritizing algorithm

a map request is sent to each neighbour. If the last known location is too old (e.g. 2 hours), OppMap looks for a recent location by using the GPS or network location provider and by sending location requests to the neighbours. As long as no recent location is available, no maps will be exchanged. If the user closes the app, he will be asked to let the OppMap service run in the background, this is necessary to continue looking for neighbours and local maps. If the service is stopped, no more maps will be exchanged.

4.1.3 Map tile prioritizing

While sharing maps, the tiles will be sent in some sort of order. This needs to be prioritized because the two map sharing neighbours could loose connection at any time and we want to get the more important tiles first. We assume the important map area is right where the user stands and the importance drops with increasing distance to the user. We implemented a sorted list of tiles to send. The first tile is the one of the users location on the highest zoom level. Then, the algorithm (as seen in Fig. 4.3) circulates around that position to populate the list. Every time a tile which is the lower right corner of its higher zoom level tile (parent) gets added to the list, this parent tile gets added, too.

4.1.4 Multi threading

Whenever longer tasks such as an http request have to be performed, they should be carried out in a separate thread. In fact, the Android documentation [8] points out the following two basic rules:

1. Do not block the UI thread.
2. Do not access the Android UI toolkit from outside the UI thread.

The former means that intensive work such as network or disk access must be performed in a separate thread to keep the application responsive and prevent the infamous "application not responding" dialogue. The second rule just says that no other thread than the main thread may manipulate the user interface. That's where the name UI thread comes from. In our application, many different tasks must be performed at the same time:

- listening to user inputs (GUI, main thread)
- listening to incoming map and location requests
- sending map and location request whenever needed
- transferring map data: either sending (server) or receiving (client) map tiles
- and some more (e.g. threads of the OSMDroid API)

Therefore, multi threading is essential to guarantee a smooth GUI without freezes.

4.1.5 API level

The first step in implementing an Android application is to choose a minimum and a target API level. The latter limits the functionalities which will be available for development and the former designates the minimum API level required on the target device for the application to run.[1] We decided to support Android 2.3 devices as there are still many devices out there with this OS. Despite choosing the minimum API level 9, we wanted to use some more recent functionalities (such as an action bar) for Android 4.0 and newer devices. To enable this, different code must be written for old and new devices. The appropriate code is then automatically chosen based on a simple conditional statement.

4.2 Development

In this section we write about the development environment we worked with, our debugging approach and which problems are still left to solve.

[1]see Appendix A for a table with all API levels

4.2.1 Environment

For the implementation we've been using Eclipse as integrated development environment and the Android SDK. To synchronize our code, a SVN repository was installed. The Eclipse plugin Subversive showed to be useful to directly synchronize with the repository from the IDE. As a reference, the Android SDK Documentation[8] is helpful, but not always complete as it often lacks sample code. A good web page for samples or solutions to programming problems is stackoverflow.com[9]. To keep track of all errors in the beta phase, we have created a simple bug tracker[10](Fig. 4.4).

4.2.2 Debugging

At first we developed all major features independently, without interacting with each other. For example we just showed a map or just sent a file over an opportunistic link. That prevented to much overhead in the beginning, because the errors depended on just one feature.

Testing the software was complicated, not at least because many errors were quite difficult to reproduce. This is common for apps which involve multiple threads and networking.

Some errors are even device and/or OS version dependant. Therefore, to test functionality, especially while sharing data, we had to check multiple different device and OS version constellations as server and client.

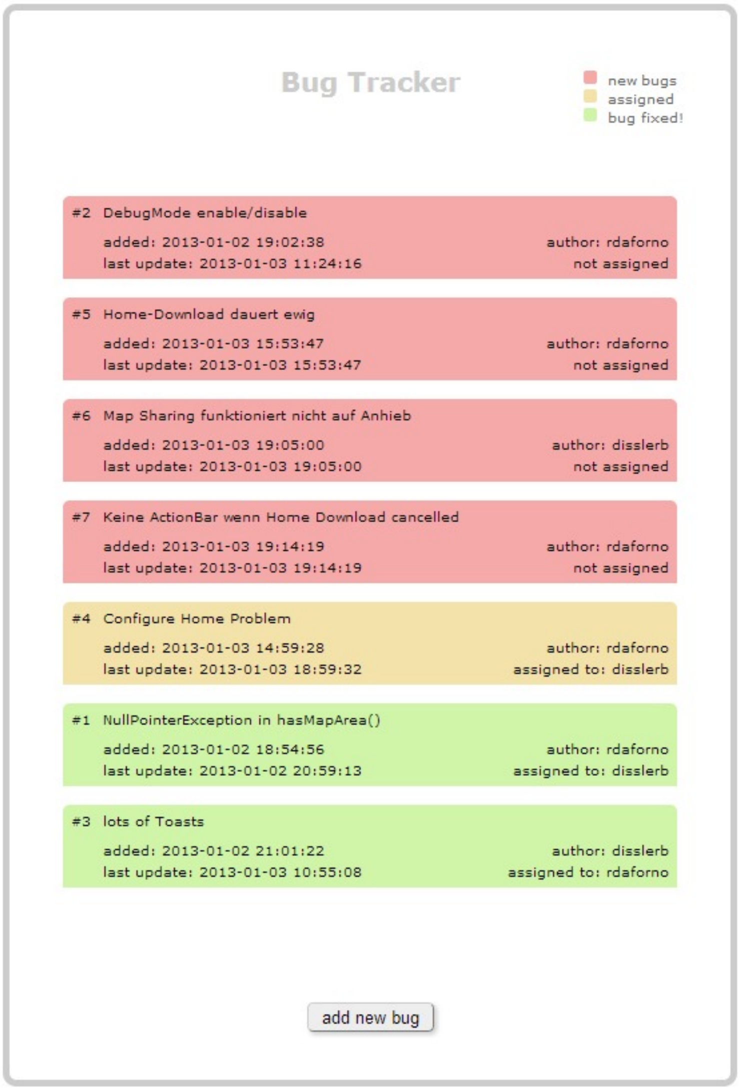
Another factor which aggravated debugging was the internal state of the application, i.e. the cache content, global variables and even the last known location. Many bugs simply did not show up in the usual case, but when the application had been running for a while.

4.2.3 Unsolved problems

As with every application, there are almost always unsolved problems. One major issue can occur if maps are shared between two smartphones and one of them runs on Android 2.3. When the screen is turned off on that device, the operating system seems to slow down the application to save battery power. This causes the map sharing process to take up to 20 times longer (1 tile per second instead of up to 20). As soon as the screen is turned on again, the sharing process accelerates to the usual speed. We were not able to determine the exact reason for this behaviour. However, it only occurs on older devices and seems to be an OS related problem.

4.3 Installing OppMap

The installation of OppMap on a smartphone is pretty straight forward. It has been tested on the Samsung Galaxy (S1, S2 and S3) as well as on an HTC Nexus One. However, it should run on any device with Android 2.3 or higher. Note that the security option ‘Unknown sources’ must be enabled to allow the installation of this non-Market app. OppMap needs WLAN-Opp to run. If it is not existing, OppMap will exit with an error message at startup. Theoretically it could run without the service, but then it would be just another map app.


Bug Tracker

🟥 new bugs
🟨 assigned
🟩 bug fixed!

#2 DebugMode enable/disable
added: 2013-01-02 19:02:38 author: rdaforno
last update: 2013-01-03 11:24:16 not assigned

#5 Home-Download dauert ewig
added: 2013-01-03 15:53:47 author: rdaforno
last update: 2013-01-03 15:53:47 not assigned

#6 Map Sharing funktioniert nicht auf Anhieb
added: 2013-01-03 19:05:00 author: disslerb
last update: 2013-01-03 19:05:00 not assigned

#7 Keine ActionBar wenn Home Download cancelled
added: 2013-01-03 19:14:19 author: rdaforno
last update: 2013-01-03 19:14:19 not assigned

#4 Configure Home Problem
added: 2013-01-03 14:59:28 author: rdaforno
last update: 2013-01-03 18:59:32 assigned to: disslerb

#1 NullPointerException in hasMapArea()
added: 2013-01-02 18:54:56 author: rdaforno
last update: 2013-01-02 20:59:13 assigned to: disslerb

#3 lots of Toasts
added: 2013-01-02 21:01:22 author: disslerb
last update: 2013-01-03 10:55:08 assigned to: rdaforno

Figure 4.4: Screenshot of the bug tracker

Chapter 5

Evaluation

5.1 Map sizes

The map gets saved in form of multiple tiles. Each map tile is a 256x256 pixel PNG file. This file format stores a lossless compressed picture, therefore its size is content dependent and not static. So, the disk space which is needed by the cached map tiles does not simply depend on the number of tiles, but also on what is in the picture of a tile and therefore on the location of the cached map area. As shown in table 5.1 a city with a lot of streets, parks, buildings and other object, all in different colors, needs up to 6 times the space of a simple landscape with a lot of green woods and meadows, some little streets and some blue currents. The map areas in table 5.1 contain multiple zoom levels. It starts at zoom level 18 with the most details. This level takes up the major part of the required disk space. Each lower zoom level has less detail and only a quarter of the number of tiles which the higher one has. The lowest downloaded level will have only between one and four tiles and therefore requires almost no disk space. We left out the zoom levels below 10 because they don't really supply more information for the map area of interest. An other thing you see in table 5.1 is, that the number of tiles depends on the location, too. This is the case since tiles on the same zoom level vary in their diameter in meters, depending on the latitude. I.e. a tile on the equator covers a much smaller area as one in Scandinavia.

It is reasonable to limit the radius for the home town map download to a diameter of 10 km, not because of the required disk space, but rather the download time. Even with a fast Internet connection it takes between 10 and

Diameter	Location	Size	#Tiles
10 km	San Francisco	87.9 MB	10.000
10 km	Napf	14.4 MB	13.000
20 km	San Jose	280.1 MB	37.000
20 km	Napf	59.1 MB	50.500
10 km	Luzern	29.3 MB	
10 km	Zrich	56.5 MB	

Table 5.1: Size and number of map tiles, depending on the location

Server	Client	speed[KB/s]	2MB transfer [s]
Galaxy S1	Galaxy S3	54 - 79	25 - 37
Galaxy S3	Galaxy S1	9 - 17	118 - 222
Galaxy S3	Galaxy S2	43 - 50	40 - 47
Galaxy S3	Nexus One	47 - 50	40 - 43

Table 5.2: Map sharing speed, depending on devices

20 minutes to download all the tiles up to zoom level 18. This is partially because of our sequential download approach. Sending more than one request to the map server at a time would significantly increase the download speed. The reason why we are not doing this is because of the amount data which is requested within a short time from the same IP. The Google Maps server sees this behaviour as bulk download and cuts all connections to this IP. Gladly, the home download is usually only done once on the first start-up of the application.

The amount of data which is shared over the opportunistic network is much smaller. Two devices will only share maps with a diameter up to 2 km when they become neighbours. That is approximately 2-5 MB.

5.2 Map sharing

In order to find out how long the map exchange takes we evaluated the opportunistic communication with the following test setup. One smartphone has a recent location (few minutes old) and all map tiles around that location. The device is connected to a wireless access point. The screen is turned off and the OppMap service is running, so WLAN-Opp will look for neighbours in the background. A second smartphone without any map tiles or location enters the room, connects to the same access point and stays within its range for at least 5 minutes. The experiment was repeated several times for each client-server combination as listed in table 5.2.

The time needed to copy all the tiles depends also on the server and client device (see table 5.2), and can range from 25s to 220s for 2 MB of data. We assume the crucial factor is the writing speed of the client device. In a real world setup this time might be slightly higher or the connection might be interrupted unexpectedly when one of the devices moves away (out of range).

The OppMap needs recent location information to share the correct material. To prevent sharing map material of an old location, which is no longer of interest, the OppMap will not share map material based on location information older than 60 minutes. If the location is only based on the network location, but not on a GPS signal, it can happen that no maps are shared, even with a valid location. It is a problem of how network location updates work: If the neighbour stays near the same access point for a long time, he won't get any location updates. Hence, the location is correct but its time stamp shows that it's too old to be used. We did not see any simple solution to this issue. In most situations this behaviour won't occur, so it is not such a serious problem as it might sound.

Samsung Galaxy S1	25 - 50s	less than 10 satellites found
Samsung Galaxy S2	35 - 70s	less than 10 satellites found
Samsung Galaxy S3	30 - 50s	more than 10 satellites found

Table 5.3: GPS fix duration, depending on devices

5.3 GPS fix

Sharing the location with our application enables immediate availability of the approximate coordinates. But how long does it take to get a GPS fix? We wanted to determine the average fix time for a cold start on modern smartphones. The test was carries out on a sunny day with an at least 150-degree obstacle free field of view (see table 5.3).

We were not able to find out why the Galaxy S1 was faster or as fast as the newer competitors. Also, the Galaxy S2 seems to be a bit slower than the S1 and S3. This might be related with the different chipsets these phones are using. The S2 uses CSRs SiRFstarIV[11] and the other two counterparts a chipset from Broadcom (4751 and 47511)[12]. However, assuming these values are somewhere in a plausible range, an average cold start time of roughly 45 to 60 seconds under ideal conditions seems to be a good guess. In contrary to the (almost) ideal conditions it is quite difficult to evaluate the GPS performance in a town. It really depends on where the phone is located. Moving a few meters may cause more satellites to become visible. The average time to the first fix may be double or three times as long. In the worst case, it is even not possible to get a fix (e.g. in a town with lots of skyscrapers).

A quick research in the web delivers the following fix times:[13]

- early devices (1980s): 12.5 Minutes
- today: 1 - 2 minutes
- warm start (updating the ephemerides of previously fixed satellites): 45s
- if more than 2-6h are passed, the information about the satellites orbit are outdated and a cold start must be performed
- hot start (position and exact gps time known and planet orbits recent): 15s

These values match quite well with the ones we obtained from our field test.

We also tried to override the genuine GPS location provider by a fake (test) provider with the same name, hoping to speed up the time needed for a GPS fix. With this test provider, we were able to fool other applications by specifying a wrong last known location. A test (see table 5.4) confirmed what can be read in forums on the Internet: The GPS library doesn't care what the last known location was. So this didn't help to accelerate the fix time.

As mentioned earlier, the time needed for a first GPS fix varies depending on many different factors[14]. To get a reasonably fast fix at least four satellites need to be in the line of sight. The terrain relief and other occluding objects like buildings or trees are the main interference factors. Whereas these interferers can be eliminated (e.g. by climbing a mountain), other factors as atmospheric disturbances in the ionosphere cannot be influenced. The weather may also

	A-GPS deleted?	average fix time
cold start (device rebooted)	yes	64s
no reboot	yes	47s, 46s
hot start	no	12s, 11s, 6s
mock 1km away	yes	42s
mock 100m away	no	15s, 12s
	yes	37s, 35s
mock 10m away	no	13s
	yes	52s
mock Washington D.C. coords	yes	37s
mock coords (0.1, 0.1)	no	12s
	yes	43s

Table 5.4: GPS fix times with different preconditions. Measured with our small test application for the fake location provider

have a slight impact on the signal quality, but this usually doesn't affect the fix time. This is due to the wavelength of the GPS signal, which is around 19cm. Objects between the sender and receiver which are much smaller than the wave length are more or less transparent.

Chapter 6

Conclusion

As part of our semester thesis we have developed an application for Android which enables map and location sharing via opportunistic networks. The application runs in the background and looks for neighbouring smartphones with the help of WLAN-Opp. As soon as new neighbours are available, map and location data is automatically exchanged. There's no need for an Internet connection. We were able to measure an average time of roughly 1 minutes for sharing the map tiles. This is a usable result as it seems feasible that for many real world scenarios (e.g. waiting at a bus stop, riding on a train or sitting in a coffee shop), two smartphones are within each others range for at least 1 minute.

We hope our semester project has generated some motivation to build more applications with WLAN-Opp in the future.

Appendix A

Android API levels

platform version	API level	version code
Android 4.2	17	JELLY_BEAN_MR1
Android 4.1, 4.1.1	16	JELLY_BEAN
Android 4.0.3, 4.0.4	15	ICE_CREAM_SANDWICH_MR1
Android 4.0, 4.0.1, 4.0.2	14	ICE_CREAM_SANDWICH
Android 3.2	13	HONEYCOMB_MR2
Android 3.1.x	12	HONEYCOMB_MR1
Android 3.0.x	11	HONEYCOMB
Android 2.3.4, 2.3.3	10	GINGERBREAD_MR1
Android 2.3.2, 2.3.1, 2.3	9	GINGERBREAD
Android 2.2.x	8	FROYO
Android 2.1.x	7	ECLAIR_MR1
Android 2.0.1	6	ECLAIR_0_1
Android 2.0	5	ECLAIR
Android 1.6	4	DONUT
Android 1.5	3	CUPCAKE
Android 1.1	2	BASE_1_1
Android 1.0	1	BASE

see Android SDK documentation for further information [8]

Bibliography

[1] Sacha Trifunovic, Bernhard Distl, Dominik Schatzmann, and Franck Legendre. Wifi-opp: Ad-hoc-less opportunistic networking. Technical report, Communication Systems Group, ETH Zurich, September 2011.

[2] Steven Meliopoulos and Suhel Sheikh. Wlan-opp: Wlan-based opportunistic networking implementation on android. Technical report, Computer Engineering and Networks Laboratory, ETH Zurich, March 2012.

[3] by Google. Google maps android api. `http://developer.android.com/google/play-services/maps.html`. Accessed: 2013-01-20.

[4] Nicolas Gramlich. osmdroid - openstreetmap-tools for android. `http://code.google.com/p/osmdroid/`. Accessed: 2013-01-20.

[5] OpenStreetMap Foundation. Openstreetmap - the free wiki world map. `http://www.openstreetmap.org/`. Accessed: 2013-01-20.

[6] Andy Allan and Dave Stubbs. Opencyclemap.org - the openstreetmap cycle map. `http://opencyclemap.org/`. Accessed: 2013-01-20.

[7] GNU Operating System. Gnu lesser general public license. `http://www.gnu.org/licenses/lgpl.html`, June 2007. Accessed: 2013-01-20.

[8] Google. Android sdk documentation. `http://developer.android.com`. Accessed: 2013-01-20.

[9] Stackoverflow.com. `http://stackoverflow.com`. Accessed: 2013-01-20.

[10] Reto Da Forno. Bug tracker. `http://bugtracker.keepcoding.ch/index.php?app=OppMap&mode=readonly`, 2013. Accessed: 2013-01-20.

[11] CSR. Csrs sirfstariv gps powers location awareness in new samsung galaxy s ii smartphone. `http://www.csr.com/news/pr/release/455/en`. Accessed: 2013-01-20.

[12] Broadcom Corporation. Broadcom introduces its most advanced gps solutions featuring glonass satellite support. `http://www.broadcom.com/press/release.php?id=s548713`. Accessed: 2013-01-20.

[13] gps-camera.eu Team. Gps-wissen. `http://gps-camera.eu/component/content/article/66.html?showall=1`, March 2012. Accessed: 2013-01-20.

[14] gps-camera.eu Team. Gps-empfang trotz schlechtem wetter? `http://tinyurl.com/b5vyxbo`, December 2010. Accessed: 2013-01-20.