Android Map Application

Staffan Rodgren
Abstract

Android Map Application

Staffan Rodgren

Nowadays people use maps everyday in many situations. Maps are available and free. What was expensive and required the user to get a paper copy in a shop is now available on any Smartphone. Not only maps but location-related information visible on the maps is an obvious feature. This work is an application of opportunistic networking for the spreading of maps and location-related data in an ad-hoc, distributed fashion. The system can also add user-created information to the map in form of points of interest. The result is a best effort service for spreading of maps and points of interest. The exchange of local maps and location-related user data is done on the basis of the user position. In particular, each user receives the portion of the map containing his/her surroundings along with other information in form of points of interest.
Contents

1 Introduction ................................................. 7
  1.1 The problem .......................................... 7
  1.2 The aim of this work ................................. 8
  1.3 Possible solutions ................................... 9
  1.4 Approach ............................................. 9

2 Related Work .............................................. 11
  2.1 Maps .................................................. 11
  2.2 Network and location ............................... 12
    2.2.1 Location-bounded information ................. 12
    2.2.2 Ad-Hoc connectivity approaches ............... 12
    2.2.3 Floating Content .............................. 13

3 Background ................................................. 14
  3.1 Vector and raster maps ............................. 14
  3.2 Components ......................................... 15
    3.2.1 OsmAnd ......................................... 15
    3.2.2 Haggle ........................................ 15
    3.2.3 The theory behind the tiles ................. 17
      3.2.3.1 The tile system .......................... 17
      3.2.3.2 Tile Servers .............................. 18
      3.2.3.3 Zoom levels ................................ 18
      3.2.3.4 The Mercator projection ................. 19
      3.2.3.5 X and Y .................................... 20
      3.2.3.6 Derivation of tile names ................. 22
      3.2.3.7 Direct formula ........................... 22
      3.2.3.8 Sub tiles .................................. 23
Chapter 1

Introduction

Today maps accompany many of our everyday activities, maps are accessible freely and easily. Thanks to the Smartphone technology, everyone can have maps in his/her pocket ready to use. Along with maps, comes a plethora of different location-related services that are very popular and taken for granted. Almost every kind of telephone owner is a potential user of map services. From the taxi drivers and people working with transports, to the common employee that needs to find a quicker way to the job or simply being working abroad, to the housewife planning to buy something in a shop she never visited before, to the young user that may get lost in some place, not to mention people loving to do sports in the outdoors\footnote{like cycling, running, trekking etc.}. Almost everybody uses maps on mobile devices.

Google offers the map service for free to its users and, in reward, it gathers information about the position of the users and can offer a whole new world of marketing services to the companies that subscribe for Google services.

Not only Google offer maps, Microsoft Maps is another example. There are even a lot of open source solutions, among those Mapnik, Cycle Map etc.

1.1 The problem

The main limitation to the services described above is the continuous need to access the Internet infrastructure.

There are situations where the availability of the Internet infrastructure is a problem, either because there is no infrastructure available or because the available infrastructure is too expensive to access, e.g., because of the high cost of data roaming\cite{25}.

In such a situation, the only way not to constantly depend on the Internet for maps, is to download in advance the needed map (or maps) when the infrastructure is accessible/affordable. Pre-downloading is a solution that takes time and
wastes a lot of space on the mobile device. Often, people don’t know that they will need a map. The need of maps arise in emergency situations. Who knows when an earthquake is going to occur? Or when the traffic compels you to find a quicker way to the job? It is important to make maps available at anytime.

Tourists are another category of mobile device users that is definitely going to need maps when the infrastructure is not available or too expensive. When abroad, the connectivity is either hardly affordable or, depending on which location, not available at all. Still, a tourist needs maps to orientate and would take advantage from region bounded informations[26]. It could be information about interesting places to visit, restaurants and other kinds of service. In fact, both those offering services and the ones who benefit them would find such an ad-hoc connectivity based platform very useful.

Users all over the world use maps not only in a passive way (i.e. merely reading the information from the map) but they want to publish personal content to make it available for other people interested in it. This would be the case in a disaster situation or as a free service for tourists. The information that users share is often not only location bounded but ephemeral and sometime fast changing. Relying on a central infrastructure for sharing ephemeral content is not always desirable since the content may never be read (WORN write-once, read never)[26]. It is inefficient to permanently save content of limited temporal and spatial validity on a central server. Still space saving is not the only concern. Saving personal information in central servers, poses other issues regarding the privacy and the protection of the individual integrity, especially, but not only, in countries where free speech is banned and where the infrastructure can not be trusted for the publishing of unhallo wed content. Decentralizing the maps would stop and make useless any attempt to censor and make map data unavailable. The data would just circulate and distribute where it is needed the most.

1.2 The aim of this work

What I want to do is to implement a system that overcomes the above mentioned limitations of the map services that are available today. I want to make maps freely available for the users of the system even when they can not access the Internet infrastructure, either because the infrastructure is not available or because the available infrastructure is too expensive to access. Even more, I want the system to make available user created location related information without the need to access any central Internet infrastructure. I want many users to be able to share the content provided by one or few users connected to the Internet and distributing the downloaded content (e.g. a group of friends trekking could find this functionality very interesting). The sharing of these information shall be secure so that users that can not trust the infrastructure, e.g., users living in countries oppressed by dictatorships that don’t allow free speech, would be able to safely use the system.
1.3 Possible solutions

There are many ways to assess the problem of map distribution. This kind of maps could be made available in travel places as train stations airports etc. via Wi-Fi or even Bluetooth in form of vector maps or in the term of raster images. Another way to share these maps is to have available copy stations that allow people to freely copy these maps on their telephones SD cards. All these alternatives requires both the creation of infrastructures for the distribution of these maps and/or an active participation of the user that will be compelled to do the transfer of the data on the telephone manually.

There is another way to share maps without an infrastructure that doesn’t require anything but the presence of the maps somewhere “near by”. Maps can be shared by ad-hoc wireless connections between mobile devices. The deploying of opportunistic connectivity technologies to share information is not a new idea, and the advantages are many. In travel places, like train stations, airports and harbours, there are many people, each one carrying a mobile device. Those places are particularly rich in ad-hoc connectivity opportunities. They would be the perfect places to deploy this kind of technology. In addition, the operational field of this application includes places like buses and restaurants where people sit still and then the probability of long lived ad-hoc connection is high. This work is a simple application of the opportunistic connectivity technology to share maps and location-related information.

1.4 Approach

I chose to use OpenStreetMap for the developing of this prototype. There is already advanced software that uses OpenStreetMap on android that gives almost the same services that Google does with Google Maps. This Software is called OsmAnd[1]. OsmAnd is available both as open source and as a commercial application. With OsmAnd is possible to visualize maps of different kinds, create own points of interest and use the Android mobile phone, almost like a GPS navigator. Beside these basic functions there are many other that are explained in the next sections. To share the content, I chose a component called Haggle. Haggle is a service to share any kind of file via ad-hoc connections. It is a publish/subscribe based system and it takes care of all that concerns the maximizing of the probability that a file reaches the ones interested in it, the buffering and the sending. The Idea is to spread local maps and local user data based on the user position. Each user should get maps containing his/her surroundings along with other information concerning those areas in form of points of interests. In order to do this, I want to make Haggle and OsmAnd communicate in such a way that, if Haggle is installed in the mobile device, then it will hand the maps and location-bounded information to OsmAnd. The advantage in using an already existing software is that there is already a great number of users all over the world taking advantage of OsmAnd’s functionalities. In
addition, the infrastructure that handle the OpenStreetMap data for OsmAnd is already deployed. This means that the users accustomed with OsmAnd won’t have to learn a new system and that I can rely on a platform that is been on the open source playground for quite a long time. This gives us the confidence to assume that it is a good, stable platform with relatively few bugs, user friendly, in the league of Google Maps and other fully commercial software.
Chapter 2

Related Work

2.1 Maps

Today there are many services that make maps available on Smartphones\(^1\). The thing that is common to those services is that they are designed to be used by connecting to a central server via an available Internet infrastructure. What I want to do instead is to use the data that those services provide and share it between mobile devices without the need that each device connects to the Internet. There are many softwares that support OpenStreetMap on Android\(^2\), OsmAnd is, in my opinion, the most complete and nice looking of all. In addition, it is a GPL\(^3\) license, which is an open source license that allows modifications. The common feature between OsmAnd and other applications that use Google Maps, is that, in order to see those maps, the system has to download each portion of the map from the server, the same way Google Maps does. An alternative that OsmAnd gives is the possibility to download in advance vector maps. In the open source repository is even available an application to create personal vector maps from the available OpenStreetMap. This is a very good feature but it doesn’t solve the problem of getting the maps if one doesn’t already have them present on the mobile device and wants to get

\(^1\)Google Maps, Microsoft Maps, Mapnik, Cycle Map, CloudMade, MapQuest Open Aerial, Migurski’s Terrain etc.

\(^2\)Like AndNav2, AndRoad, Andronic, Ape@map, AQ GPS Hiking, ARnav, BiCycle, BackCountry Navigator, Big Planet Tracks, BikeTor, BikeCityGuide, BTC Mapper, CycleStreets Android App, Dedee GpsLogger, Endomondo, ForeverMap, Gaia GPS, GoPenS, Gosmore, GPS Logger for Android, GPS-Tracks, GpsMid, gvSIG Mini, Hermes, JPSTrack, KeypadMapper, komoot - Outdoor Routplaner, Layar Reality Browser, LocA, Locus, MapAndroid, MapDroid, mapFactor Navigator free, MapQuest, Maps, Mapforge, MapsWithMe, Mapzen POI Collector, Maverick, My Tracks, Navatar, NavDroid, NavFree, Naviki, Navit, Navitel, Open GPS Tracker, OpenFixMap, OpenSatNav, OpenTrail, OruxMaps, osm-android, OsmAnd, OSMMapTuner, osmAnd, OsmPad, OSMTracker for Android, PocketNavigator, RMaps, Run.GPS, Sports Tracker, Trafficman Maps, TravelDroid, TrekBuddy, TripAdvisor, Turbo GPS, Vespucci, VGPS, ViewRanger GPS, YourPocketMap, ZANavi etc.\(^2\).

\(^3\)GNU General Public License
it without accessing some Internet infrastructure. Moreover, OsmAnd doesn’t make possible to share points of interest that are saved locally in the phone. The standard setting when creating own points of interest is to trying to upload them on the central server which requires authorization. Local editing of the points of interest is allowed when Internet connectivity is not available or if the user manually changes the default settings to save the points of interest locally instead. In any case locally stored points of interest stay on the phone. This application takes away this constraint allowing the locally saved points of interest to be shared as they were a part of the map.

2.2 Network and location

In the past years, many studies have been conducted on location-bounded information and ad-hoc connectivity. This work is based on those studies.

2.2.1 Location-bounded information

The idea to attach virtual information to physical spaces is not new. Carter et al. [10], in 2004, conducted a study on digital graffiti and showed the possibilities offered by this idea. People could use PDA to annotate content on public plasma posters. The author of the content could modify the content as well. The result is a “person-to-place-to-people-to-persons” content transfer. Other examples are [13, 20, 30]. The applications are many, e.g., the participants to a meeting arriving in a conference room, could receive a notification, if the meeting has been moved to a new location, as soon as they enter the conference room. Another application could be making available on the spot, information about culturally relevant places, for tourism purposes. Similar approaches can be used to offer information around incident places etc.

2.2.2 Ad-Hoc connectivity approaches

The Ad-hoc network environment is characterized by many partitions and the end-to-end connectivity is not guaranteed. Because of this, traditional applications can not be used. There are two different approaches to ad-Hoc connectivity, the forwarding-based approach and the flooding-based approach[16]. Among the forwarding-based protocols, we have the direct transmission approach which does a single copy, the message is only delivered to the recipient. Another kind of approach is the location-based. With this kind of approach, the nodes forward to the nodes that are closest to the recipient the message destinate to it. A third kind approach is the knowledge-based. Here is the knowledge about the network determining which nodes are to be forwarded the messages to. Among the flooding-based approaches we found the Epidemic[31] routing and the Estimate/prediction routing. While the Epidemic forwards the messages to any reachable node, Estimate/prediction (e.g. PROPHET[22]) uses knowledge of
the history of the contacts to choose which nodes to forward the messages to. Haggle[25] use an approach similar to Epidemic but it promotes fast spreading of the most interesting content by using weighted interests and also implements mechanisms to avoid traffic overload in node reach networks.

2.2.3 Floating Content

In a more recent research, Ott et al. [26], explored the concept of “Floating Content”. This approach combine ad-hoc connectivity with the idea of digital graffiti\(^4\). The basic idea is that content that is present somewhere near by, is available without the need to know in which device it is actually stored. With “Floating Content”, the annotation of urban spaces is achieved by creating place specific data for applications. Data is kept “alive” around anchor points[26], in circular areas of a certain radius. The advantage is that, as long as some device is present in such an area, the information is available. This system is totally independent from any central server. The service can be used to share any type of information that is territorially bounded, like information for tourists or marketing information about shops in the surroundings. In case of catastrophe, such a system can increase the probability that vital information reaches the people who need it, even when the infrastructure is out of order. This work is the ground on which my idea is based. This conception is shared by other works that don’t concern maps in the specific but that assess the problem of data sharing in opportunistic networks such as [9, 11, 12, 14, 17–19, 21, 23, 24, 26, 27].

\(^4\)without the need of plasma posters or other kinds of central servers
Chapter 3

Background

3.1 Vector and raster maps

There are two types of maps on the market, vector and raster. The advantage of raster maps is that, almost every on-line service that provides maps uses raster maps. Raster maps are simple pictures. The world map is cut into tiles and different zoom levels are available. Each zoom level has its set of tiles. This kind of tiles are already available everywhere. Another type of map is the vector map. The advantage of the vector maps is that they are lighter and the data is stored in layers. They are “intelligent” maps where the data can be selected and potentially aggregated. In a few hundreds of megabytes, big countries can be stored with all the information about streets, buildings, points of interest etc. This is not possible with raster maps neither it is possible to aggregate and select data from them (only the location is easy selectable). Mobile applications, like OsmAnd, use compressed binary formats to store vector maps. OsmAnd uses its own format which unfortunately is poorly documented. At the beginning of this work I’ve been trying to understand the OsmAnd “obf” format with scarce success. Beside this fact, vector maps have other drawbacks. They are not as available as raster and the ones available are not cut into tiles. A few hundred megabytes are an enormous amount of data to be shared ad-hoc on the fly. The first thing one would be compelled to do, in order to use vector maps for ad-hoc communication purposes, would be to start a service providing vector tile maps. The process of reading and extracting data from these maps is time consuming as well. In addition, vector maps are hard to modify in case of disasters that may change the geography of a place (e.g. floods, earthquakes etc.). For these reasons (the lack of documentation, availability, usability and time consumption), I decided to use raster maps to develop this prototype. This doesn’t forbid that, in a near future, this prototype could be modified to be able to share vector tile maps as well. Since the communication model is the same, to spreading vector tiles tiles is exactly the same thing as spreading raster tiles. Haggle doesn’t make any difference between different kinds of file. The
only thing that would be possible to achieve with vector tile maps, that it is not possible with raster maps, is simplify the data of each tile, allowing just the meaningful information to be left and putting together similar information. This process is called data aggregation. In other words it would be possible to pass only the most important data in order to spread more tiles in less time and in order to save space.

3.2 Components

3.2.1 OsmAnd

OsmAnd\[1\] (Fig. 3.1) is an open source software for android with many functionalities. The offline functionalities allow to visualize the raster maps (preloaded on the PC or in Android), this is the most commonly used function. It allows to see maps just by scrolling on the screen with a finger and zooming with two. Another function allows to find specific points of interest and see them on the map. Just by holding a finger on a point on the map it is possible to create own points of interest. The search function allows to find items via post code, get the description to reach a point and locate your position via GPS. Touching the compass needle on the top left corner of the screen (Fig. 3.1, second half), makes rotate the maps according to the compass. It is even possible to save routes and tracks to gpx\[1\]. Other interesting features like name transliteration for language with non-Latin alphabet and multi language support and possibility to add new languages are available. The application provides also a translation help. Other services are the possibility to look for near by public transport stops, and the direct transport routing service. Route help via car or bicycle and voice guidance are available, and many other functions.

The online features allow on line route calculation for cars, bicycles and pedestrian, the download of the raster maps from different sources, editing points of interest directly from the phone (after successful login) etc.

3.2.2 Haggle

Haggle\[25\] was introduced to give mobile devices the ability to communicate without the need of an infrastructure. This system guarantees a better delivery rate by delegating the content dissemination i.e. the nodes in the ad-hoc network receive not only what they are interested in but they silently receive even content they have no interest in. In this way they are able to pass relevant data forward to the other nodes making indirect delivery possible. Haggle differs from other opportunistic data dissemination platforms [14, 18, 19] in the sense that it takes in consideration the relevance of the content and guarantees faster spreading of relevant content. In other words, Haggle saves bandwidth while propagating what is relevant to the users. This characteristic makes it very desirable as

\[1\]GPS eXchange Format: data format for the storage of GPS data
Figure 3.1: The OsmAnd application
a tool for sharing a great quantity of items in the most efficient way possible. Each node in the Haggle network uses a node description that contains the node interests. When node A encounters node B, it gets B’s description and gives to B the part of the stored content that B is interested in, along with other relevant content. Haggle has APIs written in different programming languages. When B receives something B is interested in, B’s Haggle notifies it to the application level with an event, allowing the application using Haggle to take action. The other objects are forwarded silently. The objects that don’t correspond to an interest, age and are eventually erased after a period of time while the objects of interest are kept alive. The interests, like the attributes of each object, are of the same type. Haggle allows only perfect string matching between the interests and the attributes. Another of Haggle’s feature that is particularly desirable for the developing of this system, is the encryption of the data objects. Haggle assess the problem of the security of the content by encrypting the data so, even if the data is disseminated by delegating intermediate nodes, the data is safe all the way from the sender to the recipient.

3.2.3 The theory behind the tiles

To fully understand the choices made in integrating OsmAnd with Haggle, we need to introduce and briefly explain the way OpenStreetMap works and explore the advantages and limitation that comes with this clever but very simple structure[4-6].

3.2.3.1 The tile system

The OpenStreetMap system uses a structure called “Slippy map tile names”[6]. This structure organize the world map in tiles and in zoom levels. The higher the zoom level the more tiles are needed to cover the surface of the world. This structure correspond to an actual directory structure in the file system, that allows to catalog the tiles and to query them, simply by querying the tile URL as a sub domain of the hosting domain. This folder structure is mirrored into the client cache so that the same idea is reused locally, avoiding, in this way, the need to always re-query the server for something that already is downloaded. Each tile is a 256 x 256 pixels PNG file, each zoom level is a directory, each column is a sub directory, and each tile in that column is a file. The file name format is “/zoom/x/y.png” (Fig. 3.2) and all the servers that host OpenStreetMap maps use this kind of structure. In this way, the URLs of the tiles are the same

---

2e.g. Java
3Attributes have a name [String], a value [String] and a weight [long]. While the attributes of the objects have all weight equals one, the interests can have different weights, allowing to discern between more or less interesting objects.
4Haggle uses SQL light that allows none of the traditional SQL jolly characters e.g. “%” to match interest to attributes.
5In section 3.2.3.7 it is showed the mapping between the zoom-x-y coordinate system and the latitude-longitude coordinate system.
Figure 3.2: The slippy map structure[6]

(or similar) everywhere. This universality of the OpenStreetMap system is a fundamental feature, that allows one system to easily utilize any OpenStreetMap service without changing its core functions.

3.2.3.2 Tile Servers

Diverse kind of OpenStreetMap[6] are available on the Internet, here is some example:

<table>
<thead>
<tr>
<th>Name</th>
<th>Zoom levels</th>
</tr>
</thead>
<tbody>
<tr>
<td>OSM Mapnik</td>
<td>0-18</td>
</tr>
<tr>
<td>OpenCycleMap</td>
<td>0-16</td>
</tr>
<tr>
<td>OpenCycleMap Transport</td>
<td>0-18</td>
</tr>
<tr>
<td>CloudMade</td>
<td>0-18</td>
</tr>
<tr>
<td>MapQuest Open Aerial</td>
<td>0-11</td>
</tr>
<tr>
<td>Migurski’s Terrain</td>
<td>4-18, US-only</td>
</tr>
</tbody>
</table>

3.2.3.3 Zoom levels

The tile system is organized in different zoom levels[6]. Each zoom level cover the same surface with different amounts of tiles. The higher the zoom level the more tiles are needed to cover the world surface. Zoom levels differ in quality
Figure 3.3: The Mercator projection[4]

and quantity of detail. At zoom zero it is possible to see just the contours of the continents and of the countries, at zoom 2 and 3 it is possible to see the names of the different countries, at zoom 4 the regions starts to be visible, at zoom 5 it is possible to see the major roads, and the cities, as the zoom increases to 6 and 7, roads, lakes and villages can be seen, the detail level keeps increasing with the zoom, by zoom 14, the topography of a city like Uppsala is perfectly visible and zoom 15 to 16 gives all the details that a typical driver would need, with buildings etc. The maximal zoom level is 18. Here is a table that describes the relation between zoom level and number of tiles.

<table>
<thead>
<tr>
<th>Zoom</th>
<th>Tiles per side</th>
<th>Amount of tiles</th>
<th>Area of a tile</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 tile</td>
<td>1 tile</td>
<td>510,072,000 km²</td>
</tr>
<tr>
<td>1</td>
<td>2 x 2 tiles</td>
<td>4 tiles</td>
<td>189,269,547 km²</td>
</tr>
<tr>
<td>2</td>
<td>4 x 4 tiles</td>
<td>16 tiles</td>
<td>74,008,039 km²</td>
</tr>
<tr>
<td>n</td>
<td>2n x 2n tiles</td>
<td>2^n tiles</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>4096 x 4096 tiles</td>
<td>16,777,216 tiles</td>
<td>95.5 km²</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td>2^32 tiles</td>
<td>0.373 km²</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td>17,179,869,184 tiles</td>
<td>0.093 km²</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td>68,719,476,736 tiles</td>
<td>0.023 km²</td>
</tr>
</tbody>
</table>

3.2.3.4 The Mercator projection

The Mercator projection (Fig. 3.3) projects the globe onto a cylinder and was created in 1569 by Gerardus Mercator, a Flemish geographer and cartographer[5]. It was chosen as a standard for nautical purposes. The reason is that it represents the loxodromes (lines of constant course), as straight segments. The consequence of projecting a sphere onto a cylinder is, of course, the distor-
Figure 3.4: The Mercator projection and the size distortion[3]

Fig. 3.4 shows the Waterman projection[7], also called Butterfly projection. This projection is more realistic with respect to the real area proportions between countries. Comparing Fig. 3.4 with Fig. 3.5 can help to better understand how the Mercator projection exaggerates the area of the countries towards the poles[6].

3.2.3.5 X and Y

The latitude/longitude system is a continuous system that needs to be discretized in order to resolve the surface of the world to a finite number of tiles. Each tile has a $x$ coordinate and a $y$ coordinate[6].

---

6e.g. Antarctica (with an area of 14.0 million km$^2$[8]) is not bigger than Africa (with an area of 30.2 million km$^2$[29])
Figure 3.5: The Waterman projection[7]
These coordinates are constructed to describe the world map as a square rather than a sphere:

- $x$ goes from 0 (left edge is $180^\circ W$) to $2^{zoom} - 1$ (right edge is $180^\circ E$)
- $y$ goes from 0 (top edge is $85.0511^\circ N$) to $2^{zoom} - 1$ (bottom edge is $85.0511^\circ S$)

The number 85.0511 is the result of $\arctan(sinh(\pi))$. By using this bound, the entire map becomes a very large rectangle.

### 3.2.3.6 Derivation of tile names

The tile names are derived from the latitude and the longitude by re-projecting the coordinates to the Mercator projection[6],

- $x = lon$
- $y = \log(tan(lat) + sec(lat))$

(lat and lon are in radians)

transforming the range of x and y to 0 - 1 and shifting origin to top left corner,

- $x = \frac{1 + \frac{x}{2}}{2}$
- $y = \frac{1 - \frac{y}{2}}{2}$

calculating the number of tiles across the map, $n$, using $2^{zoom}$, multiplying $x$ and $y$ by $n$, rounding the results down to give tile $x$ and tile $y$.

### 3.2.3.7 Direct formula

The direct formulas for constructing the name of the tiles are the sequent[6]:

**xtile** :

$$\left\lfloor \frac{\text{lon} + 180}{360} \cdot (2^{\text{zoom}}) \right\rfloor$$

**ytile** :

$$\left\lfloor 1 - \frac{\ln(\tan\left(\frac{\text{lat}}{180} + \frac{1}{\pi}\right) + \sec\left(\frac{\text{lat}}{180}\right))}{\pi} \cdot 2^{\text{zoom} - 1} \right\rfloor$$

In this way it is possible to shift between the continuous system of latitude and longitude to the discrete system of the tiles. As we will see, this, on one hand, makes it possible to reduce the complexity of the issue of visualizing the map.
with different grade of granularity depending on the zoom level, but on the other
hand, sets a limit to the accuracy in deciding what we want to see and what
we don’t. Fortunately this system is as accurate as it is meaningful to be, but
some considerations on the level of accuracy will be necessary to engineer the
integration between Haggie and OsmAnd.

3.2.3.8 Sub tiles

The relation between zoom levels is a consequence of this construction and is
pretty straight forward. If one is looking at tile $x,y$ and want to zoom in, the
sub tiles are (in the coordinate system of the next next zoom-level)[6]:

This gives us a QuadTree-like structure. The advantage of such a structure is
that the coordinate reference system within a square is easily calculate from its
upper left corner and this applies for each square (Fig. 3.6).
Chapter 4

Implementation

OsmAnd and Haggle are two complete pieces of software. OsmAnd is a fully implemented Android application with advanced graphic capabilities that gives the user a very nice using experience. Haggle is an already functioning Android service proven to be useful upon which different applications have already been developed, one of these is PhotoShare.[23]. In such a comfortable situation, my task is only to bring OsmAnd and Haggle together and make them collaborate smoothly. The key issue is to make Haggle know what is important for the user to receive.

4.1 What’s important

The first step is to understand which tiles and which information is important for the user. There are many different ways to look at this problem. The first consideration is that for the user is important everything that is around her/his position. In this sense we would say that Haggle should be told that everything around the user is to be set as an interest. Let’s suppose now that an user in Stockholm decides that she/he wants to look on the Australia map to plan a trip there. There are two different ways to look at this situation in order to decide how should Haggle behave to give the user the best kind of service. One approach is to consider as “important”, any place the user is looking at on the map visualizer. This would make Haggle subscribe for the Australia tiles and hopefully give the desired map to the user. The problem with this approach is that, assuming that this functionality is used mostly when one already is abroad, we could say that planning a trip to Australia when already abroad is not likely to be the case. Now let’s examine another use case. Suppose that a user is in Thailand and that an earthquake has already destroyed half of the city in which the user was enjoying the holidays. In such a situation, the surroundings are of crucial importance. If the screen of the user’s mobile device was accidentally set to Australia before the earthquake stroke, the user, that mistakenly downloaded
the map of Australia, has now nothing to rely on to get vital information for his/her survival. This could result in a catastrophe. For this reason I choose to consider as “important” the surroundings of where the user is rather than the surroundings of where the mobile device’s map visualizer happens to be focusing on. Now we have another concept to define: what does near mean?

4.2 Near and far

The concept of near and far is not well defined, it is fuzzy. We need to discretize it and decide how to construct a well defined concept that is more useful for instructing Haggle about what the user wants at a given time. What Haggle accepts as meta-data, are only strings. This meta-data is necessary for the spreading. Each tile need attributes in order to be passed from device to device, thus we need to find a proper algorithm to “name” the tiles that allows Haggle to choose which tiles to pass among all the available ones. To start with we have to consider how Haggle sees the objects that are “floating around”[26] in the Haggle network. As pointed in the Haggle section, data objects consist of weighted attributes categorized by their name and specified by their value (both strings) plus a data payload. Names and attributes can only be matched as an exact match and no SQL “jolly character” is allowed. With this in mind now consider the tile domain. When we look at PNG maps, we look at a collection of tiles that are differently named and that are specific for the zoom level we require. A way to look at this problem is to name the tiles with their complete name i.e zoom +”some binding character” +xtile +”some binding character” +ytile. If the binding character is “_” the result is zoom_xtile_ytile. In this way we name each tile individually and we can be very precise about what we want. Unfortunately the problem with such an approach is that we are compelled to notify the interest for each single tile in each zoom level we are interested in, so the question is if there are cheaper ways to achieve a satisfying system for communicating with Haggle. The answer is obviously yes. A more interesting way to assess this problem is by looking at the latitude and the longitude. If we look at this two real values we see that if we have a set of points on the map, let’s say $P_1, P_2, P_3, ..., P_n$, each $P_k$, will own a pair of latitude and longitude coordinates ($lat_k, lon_k$), where $lat_k$ and $lon_k$ are of the form $W, a_1, a_2, ..., a_m$, where $W$ is the whole number of the latitude/longitude and where each $a_p$ is an integer. We consider the points $p_i$ and $p_j$ to be the same point if they have the same set of coordinates.

If we look at Fig.4.1, the squares contained in the big square, have the upper left corner at respectively $P(0.0, 0.0)$, $Q(0.1, 0.0)$, $R(0.0, 0.1)$, $S(0.1, 0.1)$. If we just “see” one decimal digit, each point with $x$ and $y$ coordinates between 0.00 and 0.09 will belong to the first square, each point with $x$ coordinate between 0.00 and 0.09 and $y$ coordinate between 0.10 and 0.19 will belong to the second square, each point with $x$ coordinate between 0.00 and 0.09 and $y$ coordinate
between 0.10 and 0.19 will belong to the third square and each point with \( x \) and \( y \) coordinates between 0.10 and 0.19 will belong to the fourth square. Now, the more digits we take in consideration, the sharper our “sight” becomes and the less digits we take in consideration, the fuzzier, and, as a consequence, the more points will be considered as the same point.

If we consider the set of all the points having \( Lat \) as whole number of the latitude, \( a_1a_2a_3...a_n \), as the first \( n \) decimal digits in the latitude, \( Lon \) as whole number of the longitude and \( b_1b_2b_3...b_n \), as the first \( n \) decimal digits in the longitude, we could name this set as:

\[
Lat, a_1a_2a_3...a_n, Lon, b_1b_2b_3...b_n
\]

Such a set identifies a rectangle. By choosing different values for \( Lat, Lon \), the \( a's \) and the \( b's \), we can construct different sets of the same type.

Moreover, it is easy to see, that one time we have chosen \( n \), no point can belong to more than one set and each point is going to belong to, at least, one set. In other words, in this manner it is possible to create a partition of the map that will be arbitrarily accurate. The bigger is \( n \), the more subsets we’ll get.
Now we can define the concept of granularity:

**granularity**: the granularity is the amount of decimal digits that are used to define the latitude and the longitude of an area on the map.

**NOTE**: In contrast with the tile system, in which each zoom level of the same area has its characteristic tile numeration (as shown in section 3.2.3.8), the granularity system is independent from the zoom level. This is particularly convenient when it is not known which zoom levels are available.

Back to the problem of defining what is near and what is far, we can now sketch a first definition of these concepts by saying that, for any given point $P$ and a given **granularity** $n$, everything in the rectangle $P$ belongs to, is near $P$ and everything outside is far. This is a good approximation but there is still some problem that makes this definition too different from the common sens of “near” and “far”. Lets suppose that $P$ is on the edge of its rectangle. Then this definition clashes with what we would have expected to be considered near $P$ according to the common sense.

The solution is, of course, to include in the surrounding rectangles in $P$'s neighborhood\(^1\) (Fig. 4.2). The corners of any tile are points and each of these points will belong to just one rectangle of ours for each given $n$. So if we name the tiles with the name of the rectangles their corners belongs to, we'll have a way to

---

\(^1\)Note that the squares in Figure 4.2 are granularity grid cells. Each square will contain one or more tiles, depending on which zoom level the tiles belong to.

---

27
define, given a point \( P \) that identifies the position of the user, what is interesting for the user, by choosing what is near, according to our definition of “near”.

If \( P \in \text{Lat}_{a1a2a3...a_n-Lon}, b1b2b3...b_n \), \( P \)'s neighborhood will contain:

\[
\{ \text{Lat}_{a1a2a3...a_n-Lon}, b1b2b3...b_n,\text{Lat}_{a1a2a3...a_n-1\cdot10^{-n}}-\text{Lon}, b1b2b3...b_n, \text{Lat}_{a1a2a3...a_n-1\cdot10^{-n}}(\text{Lon}, b1b2b3...b_n-1\cdot10^{-n}), \text{Lat}_{a1a2a3...a_n-1\cdot10^{-n}}(\text{Lon}, b1b2b3...b_n-1\cdot10^{-n}), \text{Lat}_{a1a2a3...a_n+1\cdot10^{-n}}-\text{Lon}, b1b2b3...b_n, \text{Lat}_{a1a2a3...a_n+1\cdot10^{-n}}(\text{Lon}, b1b2b3...b_n+1\cdot10^{-n}), \text{Lat}_{a1a2a3...a_n+1\cdot10^{-n}}(\text{Lon}, b1b2b3...b_n+1\cdot10^{-n}), \text{Lat}_{a1a2a3...a_n-1\cdot10^{-n}}(\text{Lon}, b1b2b3...b_n-1\cdot10^{-n})} \}
\]

So far we can find what tiles are near, regardless of the zoom level. At the same time, this definition takes away from us the zoom information contained in the tile’s tile path. This is indeed a very drastic simplification and some consideration is to be made about whether to keep or to discard the zoom information when it comes to tile attributes and interest definitions.

### 4.3 What is a reasonable level of granularity?

Of course the minimum level of granularity is zero. Zero granularity corresponds to roughly dividing the map by the whole number of the latitude/longitude. Zero granularity is rough but works well. The question now is: what is the maximal value that is reasonable to have? Of course it depends on what the user want to use the map for but, regardless of the map usage, a granularity that sets up a grid in which no tiles can fit is meaningless. In this sense, it is important to establish a reasonable upper bound of the granularity level for the raster maps. As mentioned above, the maximal possible zoom level supported is 18 and, according to how the tile system works, at that level each tile will stretch over the minimum possible area. Hence, looking at the number of decimal digit in the latitude and longitude that are constant when moving between two adjacent tiles will give us an estimate of how big \( n \) can grow before it stops to make sense. Since the Mercator projection, projects the earth that is roughly a sphere to a cylinder, the amount of area corresponding to a tile is different depending on the latitude. In the specific, the areas near the poles will get more stretched and exaggerated than the areas near the Equator. This means that a tile corresponding to an area far from the equator will cover a smaller surface. So let’s look at the first tiles at the beginning of the map.

In Table 4.1, we see that between two tiles of the ones covering the least area the difference in latitude and longitude is in the order of \( 10^{-3} \) which means that choosing a granularity of 3 will make us distinguish single tiles at the poles when the zoom level is maximal. What happens when the granularity is too
<table>
<thead>
<tr>
<th>Tile</th>
<th>Lat</th>
<th>Lon</th>
</tr>
</thead>
<tbody>
<tr>
<td>18_1_1</td>
<td>85.05101030905541</td>
<td>−179.998762670898438</td>
</tr>
<tr>
<td>18_2_1</td>
<td>85.05101030905541</td>
<td>−179.99725341796875</td>
</tr>
<tr>
<td>18_1_2</td>
<td>85.05089183547521</td>
<td>−179.99862670898438</td>
</tr>
<tr>
<td>18_2_2</td>
<td>85.05089183547521</td>
<td>−179.99725341796875</td>
</tr>
</tbody>
</table>

Table 4.1: North pole tiles at zoom 18

Figure 4.3: An example of a too tight grid
big? If we think about the granularity as the density of a grid splitting the map in small squares (or rectangles), if I’m at the center of a tile and my grid cell area is less than one ninth of the area covered by the tile, then no one of the corners of the tile will cross me or my neighborhood and we will not get the tile (Fig. 4.3). The grid becomes too tight. In other words, the tile neighborhood of a point isn’t big enough even to contain a single tile. This means that one grid that has cells that covers an area that is exactly one ninth of the area covered by the tiles belonging to a certain zoom level, will be useful for all the higher zoom levels but it will just fit for the first lower one. This happens because, the next lower zoom level will have tiles covering sixteen times the area of the cells of my hypothetical grid. With granularity two, it is likely not to get a tile belonging to a zoom level less than fourteen, even if the tile covers the neighborhood of the user position. For granularity one, the lower limit is zoom nine and for granularity zero is six. This behavior automatically excludes uninteresting and unwanted zoom levels from the interest space.

If we look at tables 4.2 and 4.3, we see that if we choose granularity one or two we will safely be able to use zoom levels from ten (granularity two is useful from zoom 14). But what kind of precision in term of meters and kilometers we get with granularity values of one or two?

---

2This comes from the QuadTree structure of the tile system.
3It is still possible if some corner of the tile happens to fall in the neighborhood of the user, i.e., the user is very near to some of the corners of the tile.
Let’s see what happens at the Equator. We have that:

The distance between two points \( P(0.0, -179.8) \) and \( P'(0.01, -179.8) \) is about: 1.11 km.
The distance between two points \( P(0.0, -179.8) \) and \( P'(0.0, -179.81) \) is about: 1.11 km.
The distance between two points \( P(0.0, -179.8) \) and \( P'(0.1, -179.8) \) is about: 11.1 km.
The distance between two points \( P(0.0, -179.8) \) and \( P'(0.0, -179.9) \) is about: 11.1 km.

So a granularity of one gives a neighborhood of about 33.3 km \( \times \) 33.3 km and a granularity of two gives a neighborhood of about 3.33 km \( \times \) 3.33 km.

With this data, I choose to use granularity one as default, since such an area will cover little more than a big city which is just about the useful area to know about if one is in travel or has to make decision in an emergency situation. It is worth to mention that far from the Equator the area shrinks because the longitude becomes “tighter”. In Uppsala, for example, we have:

The distance between the points \( P(59.0, 17.8) \) and \( P'(59.1, 17.8) \) is about: 11.1 km.
The distance between the points \( P(59.0, 17.8) \) and \( P'(59.0, 17.9) \) is about: 5.73 km.

4.4 Granularities and user profiles

OsmAnd can be run in different modes. Those modes are the so called user profiles. Four user profiles are available:

1. The “Pedestrian” profile
2. The “Bicycle” profile
3. The “Default” profile
4. The “Car” profile

Each profile corresponds to a different kind of usage of the map. In particular, when one drives car, it is possible to cover greater distances in comparison to an user cycling or walking. In this sense, it is reasonable to make the granularity correspond to the choice of the profile. I choose to set the granularity to one for the “Default” and the “Bicycle” profile, to two for the “Pedestrian” and to 0 for the “Car” profile.
4.5 The actual attributes

Now that we have defined a way to classify tiles containing different portions of the same area, we can use these results to start giving the tiles their attributes. The attributes of a tile are derived from the rectangles hosting it (or part of it). Note that the tile system and the granularity system don’t have to be in phase. If we are in the situation prospected in Fig. 4.4, it would be desirable to get tiles one, two, three, four and seven\footnote{Along with five, six, eight and nine.}. If we just give each tile the attribute derived from its upper left corner, it will cause the user to miss the above mentioned tiles, since their upper left corners are outside his/her neighborhood. The solution is to give the tiles all the attributes derived from all the four corners\footnote{Which means that tile one will have, as attributes, not only the ones derived from the rectangle containing tile one's left corner but even the ones derived from the rectangles containing tile two’s, five’s and four’s ones}. I said in section 4.3 that granularity one is the default but that I assume that users would be able to choose between zero and two. This decision affects the way tile attributes are set. Suppose that just one user is downloading the map of Uppsala from the Internet and this user is surrounded by ten users that can not access the Internet. Suppose that the user downloading from the Internet chooses the default granularity of one but that the other users have different choices from zero to two. If the one downloading from the Internet just names the tiles according to her/his chosen granularity, the other users with different values of granularity will not be able to get the tiles. The reason is that while the attribute of the downloaded tiles would be of the form “\textit{Lat}, a_1 \_ \textit{Lon}, b_1”, the users with granularity two will have an interest in tiles with attribute of the form “\textit{Lat}, a_{12} \_ \textit{Lon}, b_{12}”, while the users with granularity zero will have an interest in tiles with attribute of the form “\textit{Lat} \_ \textit{Lon}”. The solution is that each tile when downloaded is given the attributes corresponding to each possible choice of granularity so everyone will find the desired tiles. As a result, each tile is given twelve\footnote{In reality they are thirteen, since I attach a path attribute to know where the tile has to be stored in the slippery tile structure} attributes (three granularity levels times four corners).

4.6 How to set Haggle’s interests?

As defined above, I want to derive the interests from the neighborhood. There are many ways to assess the problem of the interests. For example one way is to give the user the interest just for the rectangle he/she is in but to give each tile, not only the attribute of its rectangle, instead each tile could have the attributes of the surrounding rectangles as well. This approach has two problems. The first problem is that, increasing the amount of attributes per tile, it increases the number of the operations that are done each time a tile is downloaded. This wouldn’t increase the complexity asymptotically but it will in practice slow down the tile visualization process. Another problem with this approach
is that it makes impossible to weight the interests, i.e., inform Haggle that the rectangle the user is positioned in is more interesting than the surroundings and that its tiles are to be handed first. For these reasons, I decided to abandon this idea and decided that each tile shall have just the attributes for its rectangle and that the interests should be set on the containing rectangle along with the surrounding rectangles. If one user is in an area whose center is $P$, then the interests will be the rectangle which $P$ belongs to plus the eight surrounding square for a total of 9 interests. The amount of the tiles corresponding to each interest depends on the granularity. For example, if $n$ tiles have the attribute “84.99_ – 179.29”, then $10 \cdot n$ tiles will have the attribute “84.9_ – 179.2” and $100 \cdot n$ tiles will have the attribute “84_ – 179”. The interests are a constant as long as $P$ moves in the same square, when $P$ change square, interests are recalculated so the total amount of interests is always nine.

4.7 Points of interest

As mentioned before, this application doesn’t just handle tiles. I assess even the problem of adding personalized information to the map in form of points of interest or POI. The way I solve this problem is that each POI, when created as a local one, other than saved on the local database, is saved as a file that has a name of the form “Lat.$a_0a_1a_2a_3a_4$-Lon.$b_0b_1b_2b_3b_4$.mine.poi.odb” where $Lat$ and $Lon$ are the whole numbers of the latitude and the longitude and $a_0...a_4$ and $b_0...b_4$ are respectively the first five decimal digits of the latitude and the longitude. Each POI is given the attributes of the rectangle containing it according to the three levels of granularity. In this way the same solution will solve the problem of the tiles along with the problem of the POIs. When
the POIs are received from a fellow device, Haggle adds the point of interest on the device’s database and keeps the received file for retransmission (note that each file may contain more than one point). Some changes were done to the way the internal OsmAnd’s database saves the points of interest, in order to maintain the indexes in a consistent state, when receiving points of interest from other devices. The problem is the clashing of the indexes that occur when database objects have the same id. In addition, if the same point (with the same coordinates) is received from two different phones, only the most recent one is kept.

4.8 Weighted interests

As mentioned in the background, one of the most interesting ability of Haggle is to decide which content to forward faster, in relation to their importance. This aspect is important in particular when the application has to get meaningful information in presence of short lived connections. Hence it is important to get the most interesting things first. Recalling that the application pays its attention on the rectangle that hosts the position of the user plus the surrounding rectangles, I state that the most important square, at least in most of the situations, is the very one that contains the user. My idea is to express this with weights. I choose the Bell curve to weight the interests. The choice has only to do with the shape of the curve, which describe the decreasing interest when getting far from the central rectangle.

$$\text{weight} = \left\lceil \text{maxweight} \cdot e^{-\frac{1}{2} \left( \frac{(x-x_0)^2}{\sigma_x^2} + \frac{(y-y_0)^2}{\sigma_y^2} \right)} \right\rceil$$

where:

\[
\begin{align*}
\text{maxweight} &= 10 \\
\sigma_x &= 0.01 \cdot |PQ| \\
\sigma_y &= 0.01 \cdot |PR| \\
Q \text{ is the point } (x_0 + 10^{-g}, y_0) \\
R \text{ is the point } (x_0, y_0 + 10^{-g}) \\
g \text{ is the granularity} \\
P(x_0, y_0) \text{ is the top left corner of the central rectangle.}
\end{align*}
\]

This formula gives a natural distribution of the interests. As we see in figure 4.5 the weight value decrease when going away from the central point. Of course
Figure 4.5: Interests around the point (17.0, 59.0) with granularity = 1

we take the ceil of this expression since the weights are integer but the effect is still the same.
Chapter 5

The System

The main goal of this system is to allow mobile devices, not connected to the Internet, getting data from some mobile devices who does (or did) and sharing these data with each other (Fig. 5.1). The data is floating in the ad-hoc cloud and is coming from the source that in attached to the Internet (or was). After the data is in the cloud, the source doesn’t need to give the data to each mobile device. The data starts to circulate and will actually arrive to everyone. The more copies of the data, the faster the data moves. Since the copies increase exponentially, the data will spread at an exponential rate. Haggle has limitations that avoid the overloading of the ad hoc connection. One of the advantage of using Haggle, is that, this powerful system makes intelligent decisions to keep the data floating in the right directions without overloading the network and with moderate energy consumption. In Fig. 5.2 we can see a schema of the software system. Here is a brief description of the classes highlighted in figure 5.2:

- **MainMenuActivity**: is the main activity. It initiate Haggle and starts the menu from which **MapActivity** is started.
- **MapActivity**: is the activity that visualizes the map. It is responsible for the localization of the user and the definition of the interests.
- **OsmandApplication**: is the main application, it handles OsmAnd’s settings and owns the Haggle handle. Here, action is taken when Haggle gets files via ad-hoc connection.
- **AbstractOpenstreetMapUtil**: is an abstract class that defines the basic actions when interacting with OpenStreetMap. (e.g. here is defined what to do when adding a point of interest to the map).

---

1This doesn’t want to be a strict UML diagram, just a view over the packages and the classes that are affected in order to do the integration between Haggle and OsmAnd and the way they interact with each other. All the classes in this schema are classes I created or modified.
**Figure 5.1:** The system deployment: four phones of which just one having access to the Internet infrastructure.

- **AmenityIndexRepositoryOdb**: Is the type of the database of the points of interest. It contains the primitives to handle the saving and the retrieving of the points of interest from the SQLite database.

- **ResourceManager**: this class is responsible for the management of the resources. Here tiles and points of interest are downloaded and reloaded. The repositories are loaded and instantiated here as well.

- **HaggleEnhancedTileDownloader**: this class is the class that contains the primitive for downloading the tiles. It is *HaggleEnhanced* because it also hands the tile to Haggle once they are downloaded.

- **OsmandsHaggleAlgorithms**: Here it is collected almost all code I wrote for the integration between Haggle and OsmAnd.

- **Objects.db**: this is the database that keeps track of the published data objects to avoid re-publishing.
Figure 5.2: A schema of the system. Each description box explains the role in the integration process of the class to which it refers (note that the classes also have other functions within OsmAnd).
Chapter 6

Evaluation

Now is time to look at the algorithms and to evaluate their efficiency.

6.1 RELOAD

When the application starts and when the user change profile an initialization thread is started. This is important because both when the application starts and when the profile is changed, there may be items in the cache that are not yet published on Haggle. This thread looks at the internal cache, searching for POIs and tiles that already are on the system to hand them to Haggle.

6.1.1 Theoretical evaluation

This is important to make Haggle aware of the existing data that was created before the Haggle process started or when Haggle was not interested in them. This functionality doesn’t serve directly the user owning the mobile device that calls RELOAD but this procedure is launched to increase the amount of objects that are shared by Haggle and to increase the life expectancy of the objects “floating”[26] in the area around the user. Here I assume that, since tiles and POIs files doesn’t vary in dimension\(^1\), the operation of reading, copying, moving or downloading tiles and POIs takes constant time (Θ(1)). Moreover, I assume that, since the names of the files generated are all of the same length for each type of file, the process of comparison between names, takes constant time as well.

\(^1\)The tile files have all the same weight in bytes as well as the ones of the POIs
The algorithm for the initialization is:

**Algorithm 6.1.1: Reload(Handle h, File Path)**

```plaintext
for each File f ∈ Path.listFiles()
    if f.isFile()
        then AddFileToHaggle(h, f)
    else reload(h, f)
```

I assume that AddFileToHaggle takes a constant amount of time. This assumption is a simplification since, as the Haggle database grows, the time necessary to add a data object to Haggle, slowly increases. For this analysis we'll assume that the amount of data is sufficiently small. This assumption is reasonable within a normal usage of the application I developed. The additional operations that AddFileToHaggle does more than adding data objects to Haggle are also to be considered to have a Θ(1) complexity, since those additional operations are to compare a constant amount of strings of constant length.

Within those assumptions, reload will do a recursive call for each directory and a Θ(1) operation for each file. If we look at the tile structure under the application directory we see that the tiles are under "tiles/<server>/x/y.png" and the point of interest are all under "POI/". So for the tiles, the algorithm will do 3 recursive calls and one constant time operation per tile. If the time for the recursive call is \( k_1 \) time units and the time of the constant time operation is \( k_2 \) time units, we will spend \( 3 \cdot k_1 + k_2 \) time units for each tile group under each "x" folder. If we assume that the tiles are equally distributed (i.e. each "x" folder contains the same amount of tiles), we can assume that \( k_2 \) is the same for each "x" folder. For the points of interests, the application will do one recursive call and, after that, all the points of interest will be in the same directory. So it will take \( k_1 \) time units one time and \( t_1 \) time units for each point of interest. In addition to this, we will have to consider an extra time of \( k_3 \) time units due to possible other directories under the application path. \( k_3 \) is constant since no other directory, under the application's path, changes its content during the application's runtime. If we have \( p \) tiles and \( q \) points of interest we have:

\[
T(\text{Reload}_{\text{tot}}) = T(\text{Reload}_p) + T(\text{Reload}_q) + k_1 + k_3
\]

and if \( k_1 + k_3 = t_2 \), we have:

\[
T(\text{Reload}_{\text{tot}}) = t_0 \cdot p + t_1 \cdot q + t_2
\]

if we define \( c_0 = min(t_0, t_1, t_2) \) and \( c_1 = max(t_0, t_1, t_2) \) we have:

\[
0 \leq c_0 \cdot (p + q) \leq T(\text{Reload}_{\text{tot}}) \leq c_1 \cdot (p + q) \forall p, q \geq 0
\]

From this we can conclude that \( T(\text{Reload}_{\text{tot}}) = \Theta(p + q) \). If \( p + q = n \), we have:

\[
T(\text{Reload}_{\text{tot}}) = \Theta(n)
\]

where \( n \) is the total amount of objects to reload into Haggle
6.1.2 Practical evaluation

Here we evaluate the actual time that is needed to run \texttt{RELOAD}.

From the reloading of points of interest we get:

Re-Loaded 148 files, 145 Pois and 0 tiles. Time elapsed: 381 ms.
Re-Loaded 148 files, 145 Pois and 0 tiles. Time elapsed: 357 ms.

From the reloading of the tiles we get:

Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 393 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 439 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 492 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 697 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 615 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 571 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 494 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 448 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 566 ms.
Re-Loaded 148 files, 0 Pois and 145 tiles. Time elapsed: 383 ms.

From the reloading of no tiles or points of interests we get an average of 3ms for 3 files. With an average of 511.8ms for 148 files of which 145 tiles and 0 POIs, 381.2 for for 148 files of which 145 POIs and 0 tiles and an average of 3ms for 3 files which are neither tiles nor POIs. Taking away the time for the 3 unknown files, the average is 508.8ms for the tiles, for the POIs we get 378.2ms. Since we have 145 POIs and 145 tiles we can calculate the time per tile and per point of interest. If we assume that the time of one recursive call is negligible, we get 3.53ms per tile and 2.63ms per point of interest. To just look at a file and not do anything we get 1ms. So if we would have \( n \) tiles, \( m \) points of interests and \( k \) other files in the application's directory we will spend \( (n \cdot 3.53 + m \cdot 2.63 + k)ms \) to do the reload. This is obviously a considerable amount of time. For this reason the task of reloading runs in a separate thread so the user doesn't notice this effort.
6.2 Getting the tiles

6.2.1 Theoretical evaluation

If we are in the situation that the nearby mobile device has all the tiles we want, the amount of time needed for getting everything (under the assumption that the connection doesn’t fall) is linear in the area that we require. This conclusion comes from the fact that the tiles are bi-dimensional, so, if we double the area, we will need double as much tiles to cover it. This result has to be multiplied for each possible zoom level (19 times, from 0 to 18). If POIs are not available, then the time complexity will be $O(Area)$. The area is proportional to the granularity:

$$\text{Area} \approx (3 \times 3 \times 10^2 - \text{granularity})^2 \text{km}^2$$

at the equator

so the computational time $T$ in an optimal connection and without points of interest would be at most: $k_0 \cdot (3 \times 3 \times 10^2 - \text{granularity})^2 + k_1$, for some $k_0, k_1$. From this result, we can say $T = O((3 \times 3 \times 10^2 - \text{granularity})^2)$ or $T = O \left( \frac{1}{10^2 \cdot \text{granularity}} \right)$. The presence of points of interest doesn’t change the picture asymptotically. In the worst case, the points of interest can not be more than linearly proportional to the area, since, recalling that we don’t have infinite precision (just double), the files of the points of interest, also represent very tiny rectangles on the map. If they are many, they may influence the computational time the most, but the complexity is still linear in the Area.

6.2.2 Practical evaluation

The average time for the mobile device to process a tile\(^2\) and get it in place is 8.48ms, this value is calculated on 46 downloads, with values between 1ms and 32ms. If we have all the possible zooms, it is possible to calculate how many tiles we have to download for a given area and zoom. The amount of tiles for granularity 1, which give us an area of $(33 \times 33) \text{km}^2 = 1089 \text{km}^2$, is 62276\(^3\). we will need an average of 8.8 minutes to get everything. In reality the last zoom levels are seldom full downloaded since no user needs that grade of magnification and, in addition, not all kinds of OpenStreetMap service have them. The average user is interested in zooms between 10 and 15. For just those zooms we get a total of 973 tiles that will take an average of 8.3 seconds to be made available. These values are lower bounds anyway, since the time of the pairing is not taken in consideration, neither is the time that Haggle spend to send the tiles to the receiving telephone. It is extremely hard to measure an average time that include the above mentioned factors since we would need to synchronize

\(^2\)From the moment the tile arrives to Haggle to the moment the tile is available for OsmAnd, which includes the time for reading the attributes of the tile, and the copying of the same from Haggles directory to the slippy tile structure under OsmAnd’s directory.

\(^3\)And not 62285, recall from section 4.3 that we get the tiles starting from zoom nine.
Environmental factors will also vary significantly. Table 6.1 gives an idea how the amount of tile grows with the granularity and the zoom level. In an average situation, the tiles of the zooms 10 to 16 are the most likely to be available so with granularity 1 it is very likely to get tiles with significant information.

<table>
<thead>
<tr>
<th>zoom</th>
<th>Amount of tiles</th>
<th>Amount of tiles</th>
<th>Amount of tiles</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>18</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>71</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>285</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1140</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>4561</td>
<td>46</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>18245</td>
<td>182</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>72978</td>
<td>730</td>
<td>7</td>
</tr>
<tr>
<td>16</td>
<td>291914</td>
<td>2919</td>
<td>29</td>
</tr>
<tr>
<td>17</td>
<td>1167655</td>
<td>11677</td>
<td>117</td>
</tr>
<tr>
<td>18</td>
<td>4670620</td>
<td>46706</td>
<td>467</td>
</tr>
<tr>
<td>tot</td>
<td>6227499</td>
<td>62285</td>
<td>636</td>
</tr>
</tbody>
</table>

Table 6.1: Amount of tiles per zoom level and granularity

the clocks of the telephones and have both telephones connected to a debugger.
Chapter 7

User experience

Thanks to OsmAnd’s advanced user interface, the user experience is very satisfying. The work done to avoid blocking OsmAnd’s processes while processing the tiles, makes this application almost as smooth as OsmAnd with the big advantage of using Haggle, if it is present on the phone. Otherwise the application works even without Haggle. The application doesn’t slow down. Since the application prioritizes the tiles that are closest to the user, the result is that the user gets the most important tiles the first. This will make the experience of getting tiles via ad-hoc connection more similar to the one of querying them from the Internet. A problem that can be a little unpleasant is the continuous vibrating of the phones when interacting with Haggle. This is a feature that is kept in this experimental prototype to facilitate testing. No one of these UI is done by me, the reason showing them is just to show where the options for the Haggle enhancement can be selected.

7.1 Control of the new features via the UI

The user can control the new features of OsmAnd by using the user interface. In the specific, it is possible to choose the wideness of the interest area by changing user profile and enable the off line POI editing from the general settings (Fig. 7.1 and Fig. 7.1). The points of interests can be easily visualized on the map (Fig. 7.3).
Figure 7.1: User profile and offline POI
Figure 7.2: Different area wideness correspond to different user profiles
Figure 7.3: Points of interest on the map
Chapter 8

Issues

During the integration work to bring together OsmAnd and Haggle, I had to deal with a lot of technical problems.

8.1 Data aggregation and vector tiles

In the original description of this work, it is mentioned the aim to do data aggregation by the means of deciding what to keep in the map and what to discard, in order to diminish the amount of data floating[26] in the ad-hoc cloud. To do this, it is necessary to manipulate the tiles themselves which, in a first phase, gave me the idea of using vector maps. OpenStreetMap, as one downloads them directly from the server, are indeed vector maps. One of the problems with this approach is that vector maps are not cut into tiles. This means that, in order to use this kind of maps, it should be provided a service that makes vector tile maps available for the customers. Nonetheless it would have been worth the effort, since one time one cuts a vector map right, the vector tiles would contain them selves the information about where they belong on the Earth map. This would have spared us the need to deal with the slippy tile map structure. Unfortunately, to save space, OsmAnd uses a compressed format which is poorly documented and therefore very opaque. After weeks of attempts, I managed to begin to understand how to gather data from a compressed vector map, and chose which layers to extract. Assuming that the map already is cut into tiles, the problem is that such a data aggregation seems to take a considerable amount of time. The attempts done on a bigger computer, resulted in time much bigger than a second which is not acceptable for an operation to do on the fly. The alternative was to write my own way to compress maps and change the coding into OsmAnd. This however was never the purpose of this bachelor thesis. Due to this insight and to the lack of time, I decided to drop this idea and concentrate my efforts to the tile system which is already available and both easier and faster to manage.
8.2 Default profile icon

The “default” profile icon is the same as the “pedestrian” one but the “default” profile has the same area wideness of the cycle profile. This is confusing but the choice to give a granularity of one to the default profile is due to the extremely small area corresponding to granularity two.

8.3 Issues with Haggle

I’ve been trying a lot of different algorithms to instruct Haggle about what to share. The one that is implemented in this final version is the one that performs the best. Some of the choices are due to apparent misbehavior of the Haggle component. These issues may not be directly due to bugs in Haggle but caused by the Java API. I state this since, testing Haggle using the C primitives doesn’t seem to cause as many problems as I encountered using Java.

8.3.1 The interests

As explained in the previous sections, Haggle is an interest-weight driven publish/subscribe system. Unfortunately, there is some limitations on the amount of interests the module can handle. Haggle stops giving information about new data objects at the application level, when the amount of interests gets bigger than 10. This behavior is noticeable even in PhotoShare[25]1. Debug traces show that the problem is the notification of the new data object on the application level.

To solve this problem I engineered the granularity-based interest definition which limits the amount of interests to 9. Having more interests would have given the possibility to choose the zoom levels of interest as well and not just the area. I tried to modify the Haggle C code to solve this problem but without success.

8.3.2 Unregistration of interests

Another misbehavior is the fact that Haggle stops giving information about new data objects at the application level, when interests are unregistered.

To solve this problem, I decided to make the position of the user (and not what the user is looking at) determine what is important. With an area of \(1089\, km^2\) it is less probable that the user will move so much to make the application change the interests for Haggle during the same session of usage and even if it happens, chances are that the unfortunate user would have been able to get some tiles before Haggle stops sharing.

---

1PhotoShare is an application to exchange photos by using Haggles publish/subscribe system. The photos are taken one by one and the user adds interests and attributes manually.
8.3.3 Duplicate publishing cause file deletion

The last thing that I found is that, publishing a file two times leads to the deletion of the file and to the impossibility to add the same file again until the application data of Haggle are totally wiped out (this is possible to do manually in the Smartphone). Since it is not possible to know if an object is already in Haggle or not, it makes impossible to directly check before publishing. Since this behavior persist even between different instances of the Haggle daemon (i.e. shutting down Haggle and starting it again doesn’t help), the solution would be to create a SQLite database containing the objects that are added, in such a way it would be possible to avoid publishing something that is already published. The drawback of this solution is that if, for some reason, the database is not synchronized with Haggle (e.g. Haggle has been reinstalled but the database is old) the application will not be able to republish and share the old objects. **Since the publishing is done each time a tile is downloaded, Haggle will destroy every tile being downloaded more than once.** This behavior will impede OsmAnd from caching the tiles, which will cause severe performance degradation. The fact that it is not possible to see into Haggle is a good thing because it guarantees security. Unfortunately the presence of this unintuitive behavior beside this perfectly reasonable and needed security measure, creates a serious problem for the correct integration of these two pieces of software.

The solution to this problem is to create a SQLite database of the published objects. Unfortunately Haggle objects are not serializable, which means that they can not be stored into the database. The database contains instead the destination path of the objects. This solution is partial, since the application can not use this database to “clean” the Haggle object repository, nonetheless, it works very well to avoid double publishing. The only thing the user is compelled to do is that, when manually erasing the data of the Haggle service from the device, it is necessary to remove the file “objects.db” from the OsmAnd’s directory.

8.3.4 C pipelining

Another problem that I experienced, was the fact that since the actual Haggle primitives are written in C, their instructions are pipelined. which means that when OsmAnd tries to launch threads that register objects into Haggle, all this concurrency cause problems, since many registrations start to interleave.

I have encapsulated the primitive for publishing data objects into synchronized methods. The method that is called for publishing is not synchronized itself so the OsmAnd’s method that calls it doesn’t get blocked. Instead, the publishing method, in its turn, starts a synchronized method that calls the C primitive, in a thread. In this way, the caller is free but the publishing is done sequentially.
8.3.5 Bluetooth

It would have been interesting to test this application in a “real world” situation, when the infrastructure is not available. I wanted to try to make two phones contact each other via Bluetooth. This is a feature that is included in the Haggle component and, according to the tutorial on Google Code, if one have two rooted phones (which I had), to enable Bluetooth it is sufficient to the operations sowed in Fig 8.1. Unfortunately I didn’t manage to make the Bluetooth connectivity in Haggle to function.

8.3.6 Lack of documentation

Another problem with Haggle is the lack of documentation. To understand how the primitives work and figure out the expected behavior, is a matter of trial and error. This caused me unnecessary and time consuming efforts that could have been spared if the Haggle functions would have been well documented. Valuable has been the information I could retrieve from the issues reported in the Google Code repository, where many of the bugs I had to deal with are reported and explained. There, it is possible to find even applications that test some of these misbehaviors.
Chapter 9

Conclusions

9.1 Future work

In future it would be interesting to do more integration testing to perfect the algorithms and go towards the developing of a better integrated application. There is still some instability, rare but consistent crashes can be experienced. Some check into the Haggle code and some debug is probably necessary. Beside this, it should be interesting to make possible to share other kinds of information, not only map tiles and points of interest. Another interesting direction is the use of vector maps. It could be the case to develop a better set to encode vector maps, allowing faster computation and making possible to look closer into data aggregation. The next step is the integration with Twimight\cite{15}, the disaster mode Twitter. Since Twimight has the location attached to the messages, it would be interesting to visualize the Twimight messages on the map\footnote{I have already implemented a beta functionality to read the first 200 tweets in the Twimight database and set them as POIs in the OsmAnd's database. This functionality is not tested and not enabled in the delivered version of this integrated software therefore it is not on this report.}. Of course, to visualize a whole flow of messages on a map, is not feasible, but in combination with a simple lexical analyzer that divides the messages in topics, groups of messages could be visualized per topic and place. In particular, when something important happens, it is likely that near by the event, many messages talk about the same topic. Point of interests with the amount of messages and the topic of the messages could be visualized on the map, and everything, Twimight messages, point of interests and maps, would be spread in an ad-hoc manner.
9.2 Applications

The main advantage of this system is that the infrastructure is not crucial for the sharing of the information since the mobile devices communicate with each other in an ad-hoc fashion. The challenges are in the best effort service. Users are not guaranteed that the wanted maps and information will be actually delivered and the result can be that some one will get an incomplete map with gaps and sometime some important information will be lost. At the same time, incomplete communication is better than no communication at all.

This kind of distributed map sharing has many applications in diverse situations. It is comfortable to always have the possibility to look up in a map whenever one travels or does activities in the outdoor. For some years ago map were expensive and people had to go and buy them. Now it is free and there are many solutions available on line that offers free maps to the users.

The infrastructure may not be available because of a catastrophe like an earthquake or a tsunami. After the Fukushima disaster it became clear that the uses of social media is important to spread information and the information wave was faster that the tsunami’s one[28]. In these situation it is crucial to be able to get updated information about the terrain so the people that are affected by a disaster and are isolated from the Internet (and the rest of the world) can get information about place of shelter or inaccessible places, closed roads and information about the actual situation in particular places. It is vital to facilitate the process of making decisions about where is the best place to go to, in order to increase the chances of survival.

The possibility to share map located information is not only useful in extreme situations. For example, users in a shopping center could share opinions about the diverse boutiques, just by locating some description on the map, in correspondence of the boutique position on the map. This open the possibility of free advertisement of products and services. If two or more friends do trekking, it is useful to be able to share maps. Another big area of application for this kind of service, is everywhere the infrastructure, however available, can not be trusted, e.g. in places where the freedom of speech is not guaranteed. Ad-Hoc sharing defies the censuring and can be used to coordinate efforts. Finally, with more and more people using social media that need a server, there is a problem of space saving. As said at the beginning of this work, to use the infrastructure for volatile content is not desirable, and, with the increasing number of users, it is imaginable that the future is to move many of the services that today borne the servers, towards the ad-hoc cloud. Whenever it is not vital that all the messages arrives and that they arrive right away, it is thinkable to employ ad-hoc networks.
9.3 Summary

Thanks to the granularity architecture and the use of Haggle, this work achieves the basic premise of sharing map information opportunistically. By tagging the tiles with intelligent names, I implemented a non-naive sharing system that does zone as well as zoom level selection, excluding uninteresting data from using bandwidth and wasting precious time. The choice of not doing active zoom selection is dictated by the possibility that, selecting zoom levels in a too rigid way, may make the system require zoom levels that are not available in the surroundings and, in this way, loose valuable information. After all, getting tiles with a detail level that is too sharp or too fuzzy is still better than not getting any tile at all. Once instructed correctly, Haggle guarantees a fast and cost effective\(^2\) sharing that is very likely to match the user expectancies.

\(^2\)in terms of energy consumption and bandwidth utilization
Bibliography


56

