Abstract

Within recent years, mobile phones have become carriers of numerous smart applications and the potential of these mobile devices is even greater than it might look at first glance. Behind the combined pieces of information that many mobile devices possess lies hidden meta information that accounts for more than the mere sum of its parts, e.g. beginning and end of a traffic jam. Using our framework, researchers and scientists can create versatile requests for information and send them to mobile clients within a couple of minutes. For instance, environmental data could be sensed periodically without any user intervention. Prototype models of several information requests have been created to demonstrate the various aspects of possible public sensing scenarios, bearing in mind the users' concerns regarding data security and privacy. Our framework does not only manage creating and sending of specifically designed tasks, it also dispatches their executions on the client side, sends back the response, and interprets it. Therefore developers can fully concentrate on the content of sensing information.

Deutsche Kurzbeschreibung

In den letzten Jahren wurden mobile Telefone Träger verschiedenster intelligenter Anwendungen und das Potential dieser mobilen Geräte ist sogar größer als es auf den ersten Blick scheinen mag. Hinter den kombinierten Teilinformationen, die viele mobilen Geräte besitzen, liegen versteckte Metainformation, die größer sind als die bloße Summe ihrer Teile, z.B. Beginn und Ende eines Staus. Forscher und Wissenschaftler können mit unserer Rahmenarchitektur verschiedenartige Informationsanfragen erstellen und diese mobilen Clients innerhalb einiger Minuten schicken. Umweltdaten könnten z.B. ohne Benutzereingriff regelmäßig gesendet werden. Prototypmodelle verschiedener Informationsanfragen wurden erstellt um die unterschiedlichen Aspekte möglicher gemeinnütziger Messungsszenarien zu demonstrieren. Dabei hatten wir die Bedenken der Benutzer im Bezug auf Datensicherheit und Privatsphäre im Blick. Unsere Rahmenarchitektur koordiniert nicht nur das Erstellen und Versenden speziell designter Anfragen sondern verwaltet auch deren Ausführung auf dem Client, sendet die Antwort zurück und interpretiert sie. Deswegen können sich Entwickler völlig auf den Inhalt der aufgezeichneten Information konzentrieren.

Contents

Abstract **2**

Deutsche Kurzbeschreibung **3**

Contents **4**

1 Introduction **7**
- 1.1 Motivation 7
- 1.2 Overview 8
  - 1.2.1 Related Work 8
  - 1.2.2 Contribution of this Thesis 9
  - 1.2.3 Thesis Outline 9
- 1.3 General Remarks about Writing Style and Layout 10

2 Examples of Applications **11**
- 2.1 Meeting in Town 11
- 2.2 Sensing Environmental Data 11
- 2.3 Baby Phone 13
- 2.4 Playing Games 13
- 2.5 Live Traffic Information 13
- 2.6 Picking Someone up Just on Time 14
- 2.7 Using a Mobile Device to Unlock Objects 14
- 2.8 Do these Roads Cross? 15

3 Fundamentals **17**
- 3.1 Converting Geographical Coordinates into Cartesian Ones 17
- 3.2 Storing the Tasks 17
  - 3.2.1 Hardcode in the Source Code of the Server Class. 18
  - 3.2.2 Read the Tasks from an XML File 18
  - 3.2.3 Write a Parser 19
  - 3.2.4 Outsource the Creation of Tasks to `*.java` File 20

4 Privacy and Security **21**
- 4.1 Security Concerns 21
  - 4.1.1 Why? 21

4.1.2 Data Collectors . . . 21
4.1.3 Specific Public Sensing Concerns . . . 22
4.1.4 Data Security Aware Public Sensing . . . 22
4.1.5 Implementation of Encryption . . . 24

5 Architecture **26**
5.1 General Precedent Thoughts . . . 27
5.2 Abstract Fundamental Architecture . . . 27
5.2.1 Tasks . . . 27
5.2.2 Server . . . 29
5.2.3 Client . . . 29

6 Implementation **30**
6.1 Why Android? . . . 30
6.2 Package Structure . . . 30
6.3 Shared Classes between Server and Client . . . 31
6.3.1 Message and Its Subclasses . . . 31
6.3.2 GUI Classes . . . 34
6.3.3 Input/Output . . . 37
6.4 Server's Classes . . . 38
6.4.1 The Core Class: `Server` . . . 38
6.4.2 `CreateTasks` . . . 40
6.4.3 `TransferTasks` . . . 41
6.5 Client's Classes . . . 41
6.5.1 The Core Class: `Client` . . . 41
6.6 Creating the Tasks . . . 43
6.7 The Life of a Task: An Exemplary Overview . . . 44
6.7.1 Creation . . . 44
6.7.2 Sending . . . 44
6.7.3 Reception . . . 44
6.7.4 Delay Due to Location or Time . . . 45
6.7.5 Execution . . . 45
6.7.6 Awaiting User Interaction . . . 45
6.7.7 Sending Back the Task . . . 46
6.7.8 Displaying the Sensed Data . . . 46
6.8 Using the Menu . . . 46
6.9 Practical Remarks . . . 46
6.9.1 USB Debugging . . . 46
6.9.2 Means of Information Exchange . . . 46

7 Usage **48**
7.1 Starting the Server . . . 48
7.1.1 Bundling the Required Packages in a `*.jar` File . . . 48
7.1.2 Adding Android and Google Maps to the Class Path . . . 48
7.1.3 Start the Programs of the Server Package . . . 48

7.1.4 Arguments Passed via Command Line 49

8 Tests **50**

8.1 Connection Loss 50

8.2 Field Test 50

8.2.1 Introduction 50

8.2.2 Setting up the Tasks 51

8.2.3 Performing the Test 53

8.2.4 Evaluation 53

8.3 Usability 54

8.3.1 Tests on People without Previous Android Knowledge 54

9 Peek into the Future **56**

9.1 The Big Picture 56

9.2 Extension Possibilities of Our Framework 57

9.3 How to Shape the Future 58

Bibliography **60**

Nomenclature **62**

Chapter 1

Introduction

1.1 Motivation

‘Can you make phone calls with it as well?’—This sarcastic comment that an owner of a smart phone might hear while showing off the fancy features of his new device reveals a lot about the speedy evolution of so-called smart phones within recent years. Apps for all sorts of things can be downloaded so that the original functionality—namely making phone calls and sending messages—is reduced to merely one of many things that can be done.
Since an ordinary home computer is still able to calculate faster, has access to more storage, is connected with more convenient input devices, a bigger screen and better sound system—the question arises why smart phones are something that companies earn a fortune selling whereas PCs, which are superior to these phones in virtually all aspects, do not have sell growth rates nearly that high. Two key advantages of mobile devices are crucial to their success:

- First of all their mobility. Being able to have access to somewhat limited functionality right on demand wherever you are by just taking your phone out of your pocket is in a practical sense superior to having to wait till you are at home regardless of the advantages the home computer has.

- The second advantage is that these devices have inbuilt sensors, the most important one is position detection with GPS. The device knows where it is and therefore the work of acquiring that knowledge is shifted from the user to the device. For instance, I do not have to tell the software where the train station I want to travel from is, my phone knows where I am at.

Of particular interest to not only the end user but also to scientific research groups and companies is the collected and combined information of numerous cell phones. They form a net of local sensors spread out all over the place which can be used to inexpensively gain e.g. current weather data which can be fed into forecast systems. Another application would be smooth traffic control which can be achieved by combining the collected information of many devices. According to the principles of distributed agent systems, behind a collection of many single bits of data there

lies hidden meta information that can be extracted with suitable algorithms whose results account for more than the mere sum of the single information bits.
There exists a wide range of thinkable applications such as sensing all kinds of environmental data or querying the user to retrieve information that he has access to right now. One especially promising application is live traffic surveillance and control since knowing the more or less precise location of a traffic jam is more than any single car driver can know; a combined effort of several mobile devices in cars can achieve it, though. The ideas we will show are similar to a traffic jam reporting system which gives the user the chance to report a traffic jam [1]. We go beyond their ideas and envision a system without user interaction as will be demonstrated later on.
This is just one of many possible applications for distributing sensing tasks to several mobile device users where the public reaps the benefit of it which is called *public sensing*. Looking at traffic sensing we see that public sensing has the potential to benefit society with negligible efforts for the individual which is characteristic for public sensing.
Even though the efforts of the phone users are small compared to the advantages, one should nevertheless continue to strive toward highly efficient energy consumption and automatization without the required interference of users since computers ought to do their utmost to bridge the gap between machines and humans.
Programmers of public sensing apps should stick to the principle that public sensing *"may not interfere with normal operation of a mobile phone, i.e., the energy consumption [...] needs to be restricted."* [2].

1.2 Overview

1.2.1 Related Work

In the following, some key papers of several fields of mobile sensing will be shown. How they relate with our project will become obvious without further explanation during the course of this thesis.
MobGeoSen [3] focuses on monitoring environmental data and provides the users with cartographic displays and gives them the opportunity to annotate with text and photos.
NoiseTube [4] allows users to share measured noise data with others.
Is a cell phone microphone only capable of detecting noise volume? The developers of *SoundSense* [5] convince us that even a standard cell phone microphone is capable of doing far more. This project uses the microphone input to distinguish between several kinds of sound. Once it can detect the sound of a car from within, it could automatically start sending its location to traffic surveillance systems on a regular basis, just to mention one example.
Not only sound can be measured, however. With appropriate sensors, e.g. carbon monoxide pollution can be sensed and drawn on a map [6].
How a mobile object could be tracked and traced automatically while minimizing energy consumption of mobile nodes has been theoretically looked into already as

well [7].
The well-known *OpenStreetMap* [8] is one example of an already fairly developed and established result of numerous sensings.
The *MetroSense* [9] architecture developers envision the future of public sensing and ask themselves *"how to propel sensor networks from their small-scale application-specific network origins, into the commercial mainstream of people's every day lives"*.

1.2.2 Contribution of this Thesis

This thesis is part of the second version of the *Spontaneous Virtual Networks* project *(SpoVNet 2.0)* which creates overlay networks on top of the traditional IP-based network infrastructure to enable flexible and adaptive network communication.
The goal of this thesis is to build an Android-based system that manages and disseminates context collection tasks to mobile phones which execute these tasks based on spatiotemporal constraints and coverage requirements. The results of the task execution are sent to an infrastructure-based central server.
Using our framework drastically simplifies the process of public sensing and therefore enable researchers to work very efficiently since they can totally focus on the *information* of public sensing and do not have to bother about the *means of transmission*. Figuratively spoken, our system provides developers and users with forms and all they have to do is enter the parameters and return it to the postman who will transport the messages, schedule their execution, and return and interpret the answer.

1.2.3 Thesis Outline

After having presented the motivation for using mobile devices for public sensing tasks, this thesis will show some diverse example applications in Chapter 2 on page 11. After having introduced some fundamental concepts in Chapter 3 on page 17, we will go beyond the theoretical technical possibilities of what can be done and will discuss the practical concerns of what should be done and what needs to be regarded with special sensitivity to users' security and privacy issues, see Chapter 4 on page 21. Then we will design an architecture in Chapter 5 on page 26 bearing in mind the applications envisioned earlier. Next we will shown our implementation of the system in Chapter 6 on page 30 and go into detail explaining the applications that we wrote already. In Chapter 7 on page 48 we will demonstrate an exemplary use case. To wrap up our project, we will carry out some tests in Chapter 8 on page 50 and show a real life scenario. Finally, in Chapter 9 on page 56, we will first try to imagine the role of public sensing in the future and later on discuss ideas how we can shape it.
For an explanation of abbreviations see the nomenclature on page 62.

1.3 General Remarks about Writing Style and Layout

For simplicity's sake the user of a mobile device and the developer of applications will sometimes be referred to as 'he' which is to be read gender neutral not meaning to exclude females, of course.
Java code, especially class names, will be written in `typewriter style`. Source code blocks, however, were deliberately not set with a monospace font for the sake of better legibility.
References to literature are consecutively numbered and the number is displayed in squared brackets.

Chapter 2

Examples of Applications

Even though we focus on 'public' sensing we now want to demonstrate sensing applications not only from this field. Figure 2.1 shows the difference between *personal*, *social*, and *public* sensing. The adjectives denote who the collected data is for: an individual, a social group, or the general public.

2.1 Meeting in Town

The most basic useful application thinkable would be the server's request to a client to reveal its current location. If a server knows the locations of several mobile devices, it can calculate a place in town that is about the same distance away from both where they can meet. Or the server can simply show both users' locations on a map to leave it up to them where to meet. If the city is unknown to at least one of them so that they have no common knowledge to use to explain to the other one where they are, a calculated meeting point to which the navigation system can guide is especially helpful.

2.2 Sensing Environmental Data

One of the most promising applications for public sensing is collecting real time information about noise volume, temperature, humidity, air pollution, atmospheric pressure, and the like. For weather forecasts the principle 'The more, the better' is valid. The more detailed information one can supply a simulation program with, the more accurate results will come out of it. Therefore using mobile devices as numerous little sensors spread all over the place could first improve weather prediction and second reduce costs because the number of expensive weather measuring devices can be reduced.
One has to bear in mind, though, that the collected data by mobile devices is less accurate: a phone could be in a pocket or the temperature sensor could be covered and the like which both influences the measured temperature significantly. The in-pocket-case could be ruled out by making sure it takes the measurement only if the measuring time is framed by two user actions within a short period of time. The

Figure 2.1: People-centric sensing applications can be thought of as having a personal, social, or public focus [10].

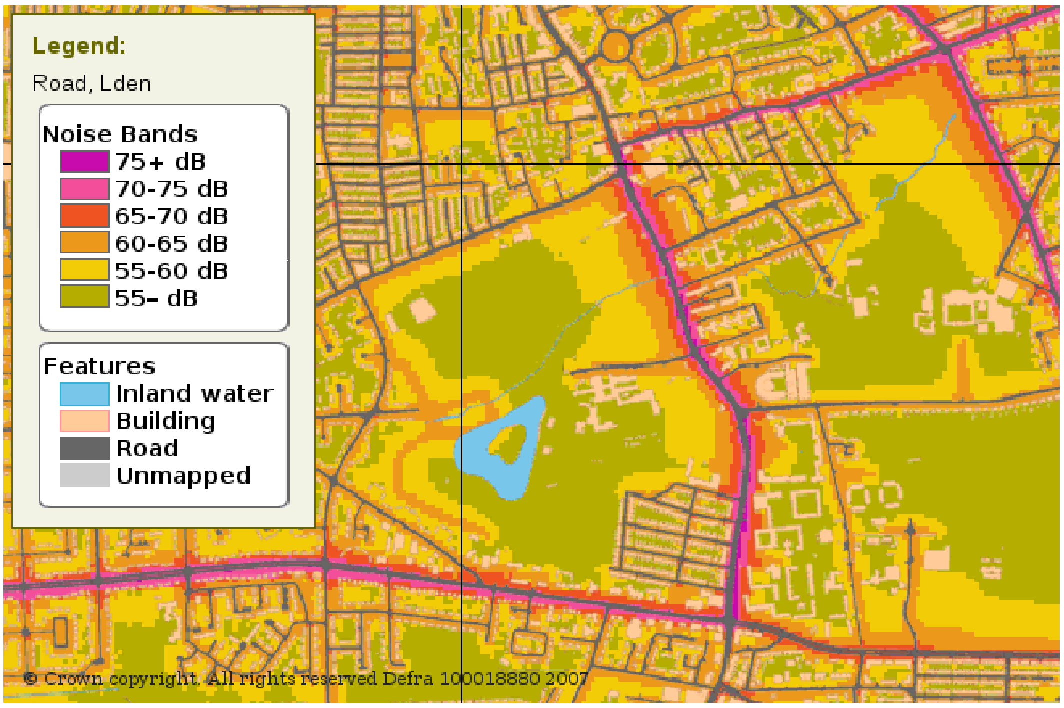


Figure 2.2: A noise map of Manchester [11]. The highly frequented roads stick out clearly.

temperature sensor should furthermore be located in an area that is normally not covered when holding the device. The measuring can also be done with calibrated measuring devices and the collected data can be transmitted via RFID chips to a mobile device and from there to the server using standard telecommunication infrastructure [12]. The disadvantage of inaccurate measured data can so be omitted. Concerning the usage of noise level maps, Weinschrott et al. [12] write:

> *"City administrations need statistics to develop a city according to the behavior and the needs of its inhabitants. For example, a noise level map of a city supports decisions on traffic-reducing measures to increase the quality of living conditions."*

Figure 2.2 shows an example of a noise level map.

2.3 Baby Phone

Once loudness can be sensed, it will be easy to define a threshold and notify the server or another phone if it is crossed. So the parents will know whether their baby is asleep or crying and will be notified in the latter case. Instead of sending a message that the threshold has been crossed, the system could also trigger livestreaming of audio or even video to the phone that is with the parents. To rule out other noise sources such as thunder or a train passing by one could only ring alarm if the high frequencies that are distinct of baby cries are loud enough.

2.4 Playing Games

Somewhat similar to the board game *Scotland Yard* some people could play a game in real life. Everyone has a cell phone that communicates with the server. One person is chased by the others who play cops and every couple of minutes the absquatulator's[1] position on the map is revealed to the others. Whenever he has walked e.g. 100 meters this information is sent to the cops as well. The cops need to work together, ideally via cell phone calls, to find the person. Playing in a city taking the subway could be allowed as well, maybe for a limited number of stops.

2.5 Live Traffic Information

We see this as one of the most promising and useful applications. In general, the more densely populated an area is, the more traffic jams occur and the more cell phones there are per square kilometer. Therefore it seems ideal to make these thousands of available cell phones in cars send their location and calculated speed to the server on a regular basis. All the server needs to do then is the following:

1. Find out if the cell phone is on a road. It should take several location snapshots for that to rule out people walking over a bridge and the like.

[1]The one being chased.

2. Compare the reported speed with the speed limit or recommendation. The lower the ratio between reported speed and speed limit is, the stronger the advice to other drivers becomes to take another route. Give special attention to traffic lights, though, do not count speed reports in front of them.
3. Extrapolate the whole road given this speed information linked to several location points. Take into consideration that newer snapshots should play a more important role than older ones.
4. Feed the gathered traffic information into navigation systems.

A light version of this traffic jam reporter [13] involves the user to tell the server that he has just reached the end of a traffic jam; end defined as his car being the last one in line. So this version has two disadvantages to the one we propose: First, it requires user involvement, and second, that it is only boolean—either traffic jam or no traffic jam.

2.6 Picking Someone up Just on Time

The problem is well-known to most of us: Often when someone picks up a number of people living in different places with the car, someone has to wait for the other one since in practice a precise pickup time can hardly be adhered to due to the unpredictable traffic situation or other unexpected delays. So either the car driver has to wait until the one to pick up is ready to go or this person is ready before and has to wait for the car to finally arrive.
If both have a mobile device with appropriate software, though, the situation can look entirely different. The driver's location can be displayed on the other person's phone and he can either estimate for himself how long it will take the driver to arrive or the phone of the one being at home can send its location to the driver's phone whose navigation system will respond with an estimated time of arrival. As soon as this falls under a threshold, the person will be asked to get ready to be picked up.

2.7 Using a Mobile Device to Unlock Objects

There are many objects that we lock and unlock every day: Our houses, cars, computers, bikes, and lockers. What if the cell phone could be turned into some sort of general key for numerous devices? As soon as I approach the door of my house, my cell phone will realizes that I have entered my property, sends a message to the server which passes it on to the 'house management computer' that has 24/7 internet access and unlocks the door. Of course a number of questions arise: What if...

- ...the battery of my cell phone is empty?
- ...I cannot get GPS or other location information?
- ...my phone gets stolen?

- . . . my house management computer looses internet connection or power or has another hardware or software failure?

Even though the first two scenarios could be solved by supplying the password to the management system via another internet connection and loosing the phone is actually less of a tragedy with regards to unwanted intruders into the house than loosing the physical key since one can easily take away the access rights from the phone—the need for a back door in the figurative or literal sense is obvious. This could be a fingerprint sensor or a number block where one has to enter a pass code or an ordinary physical key stored in the house of a friend or relative.
One of the great advantages is that the phone has the potential to be the key for everything that has a computer with an internet connection attached to it. Door lock systems that already have permanent connection to a central server, using e.g. magnetic card readers, can easily be modified to grant access using the phone's location information.
One issue arises, however, namely the unavailability of GPS information inside buildings. Several companies are constantly working towards eliminating this disadvantage by supplying location information based on cell tower locations and wireless LAN signal strength of geographically known senders. [14] [15].
Using GPS or other location providers might seem like an overkill if all we want to find out is whether a mobile device is within a certain range of something else; having a look at the big picture, however, convinces that using existing technology infrastructure is superior to establishing new systems and having to carry around only one device is more convenient than two.
A drawback, however, is that GPS or another location finding system has to be switched on constantly which is fairly energy consuming.

2.8 Do these Roads Cross?

Putting our feet back on the ground of reality, now we do not want to think visionary ideas any more but show an application that is actually already up and running, namely the question to the cell phone user whether the two roads he is very close to right now do actually cross or if one runs over the other. Especially for traffic routing this is essential information because if they do not cross, the navigation system better does not suggest users to drive from one to the other one. Google Maps relies heavily on satellite images and it is very hard to determine if two roads cross or do not cross looking down on them from a satellite.
So whenever it is not known yet if two roads intersect, a question can be sent to the users in the area around. As soon as the cell phone is very close to the possible crossing, the task becomes flagged as 'executable'. After the user has confirmed execution and after having clicked on *Show Map* he sees Figure 2.3. In Figure 2.4 he has just clicked to select the arrangement.
In the course of this thesis we will have a detailed look at the different aspects which lie behind this application.

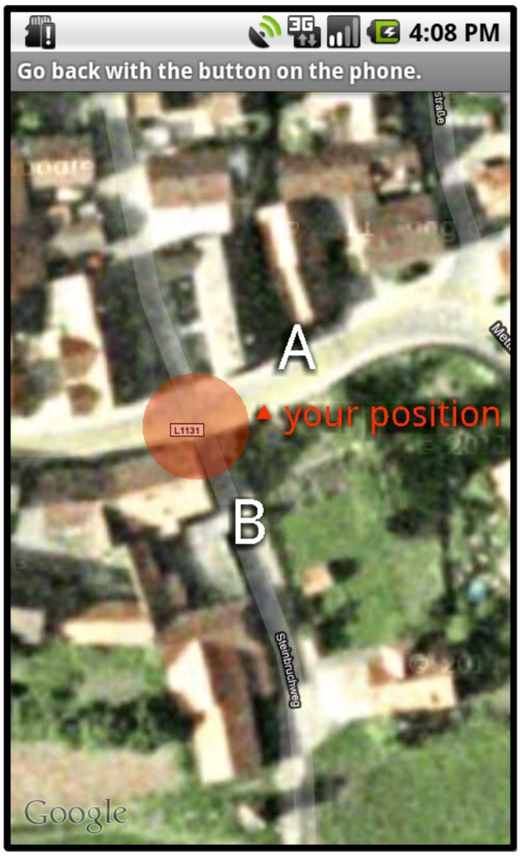
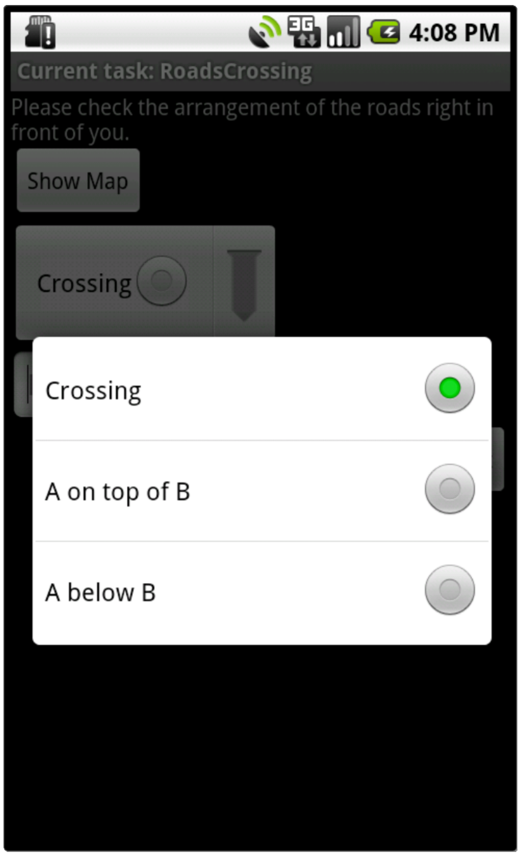


Figure 2.3: Now remember which street was 'A', hit *back*, and answer.



Please check the arrangement of the roads right in front of you.

Crossing [✓]
A on top of B
A below B

Figure 2.4: Hidden under the spinner is a text field to enter a comment, *bridge*, for instance.

Chapter 3

Fundamentals

In the following we will discuss some concepts that we will rely on later on.

3.1 Converting Geographical Coordinates into Cartesian Ones

If we want to show a map with overlaid text or graphics, e.g. distances, we have to be able to translate a length unit into geographical latitude and longitude. We will now look at calculating geographical points that are a given distance away from the origin. All points that fulfill this constraint are located on a circle. The position of a specific point on the circle is determined by an angle from the horizontal axis to the right defined as zero going in the mathematically positive direction.
The distance between two degrees of latitude is always about 111 kilometers, but the distance between two degrees of longitude depends on the latitude of the position: On the equator it is also about 111 kilometers, on the poles it is 0. Using the formula below we can compute an approximate value of the distance between two degrees of longitude[1]:

$$dist = 1.11 \cdot 10^5 \cdot \cos(\pi \frac{currentLat}{180})$$

Then we divide the radius by this distance. The result is the distance in degrees. We would add[2] it to the value of the latitude to move this distance eastwards; since the point is normally not directly eastwards, however, we multiply it with the cosine of the angle and do respectively for the longitude distance using sine.

3.2 Storing the Tasks

Several possible ways of telling the server which tasks to send to the client come to mind:

[1]The cosine is set to radiant and the distance is in the SI standard unit meter.

[2]Crossing the the prime meridian and the 180th meridian have to be calculated with special care using modulus.

1. Hardcode in the source code of the server class.
2. Read the tasks from an XML file.
3. Write a parser.
4. Outsource the creation of tasks to a `*.java` file.

In the following we will give reasons why we opted for the last alternative. To comprehend this decision, one has to bear in mind that we are building a prototype and not a huge project funded by a company. Therefore rewriting classes such as `Calendar` was not an option since it might look easy at first glance, having a closer look at it, however, reveals many complex functionalities such as leap year algorithms, time zones and daylight saving times, just to mention a few.

3.2.1 Hardcode in the Source Code of the Server Class.

Due to lacking flexibility this would be only a good idea while testing since the people who will send out tasks will not want to recompile the class whenever they have made some changes to the tasks they want to send. And changes will be rather frequent and be it only changing the time frame of execution a couple of minutes.

3.2.2 Read the Tasks from an XML File

There are four methods of doing this that we have looked into:

JDOM

The big disadvantage here is that we would have to reprogram our tasks to make them ready for *JDOM* to put them into XML because it cannot just take objects and convert them to XML, every single variable needs to be converted to XML individually. On top of that, whenever a user creates a new task, he would have to do the same thing. This is why we looked into something that takes entire objects and puts them into XML files without our assistance.

JAXB

JAXB does have this functionality, it is not available for Android, however, since it depends on classes that Android does not provide. Even if it were available, though, we would still have to add an empty constructor to each class we have programmed and set up an initialization method to pass on the parameters. This would be a manageable effort but since we also use already existing classes such as `Calendar` and do not want to recode their use[3] to make it applicable for *JAXB* this is not the solution either.

[3]We would need to use the standard constructor and pass the variables via `set` methods.

```java
@Root
public class Example {

    @Element
    private String text;

    @Attribute
    private int index;

    public Example() {
    }

    public Example(String text, int index) {
        this.text = text;
        this.index = index;
    }
}
}
```

Listing 3.1: An annotated class for *Simple*

Simple XML Serialization

Simple XML Serialization [16] is available for Android and we can give it an object and tell it to serializ0 ite; there is drawback, however: We have to annotate the elements, that is add the words behind the '@' as demonstrated in Listing 3.1. This would be doable for our own classes yet not for the already existing classes we use and would be a burden on developers of future classes.

Create XML Manually

Even worse in terms of effort would be creating the XML file by hand. On top of that, one would have to implement workarounds for saving existing classes such as `Calendar`, for instance.

Taking into consideration that so far we have been facing massive problems of inefficient effort when trying to create an XML file, we should try to find another method also since we have not even looked at creating Java objects from the XML yet which would make us face similar problems.

3.2.3 Write a Parser

Whether the file format we want to store our objects in is XML or some other text based format, the problems mentioned above remain.

3.2.4 Outsource the Creation of Tasks to `*.java` File

This is how we solved the issue with the following advantages:

- No restrictions with regards to the use of existing classes.[4]
- No need to annotate the classes we created.
- No need to have an empty constructor and extra methods to set variables.
- A clear and straightforward way of modifying or creating a task. Something that can be written in one line in Java might take ten lines in XML.

So despite of the disadvantage that the created files cannot be edited manually like XML, we decided to save them as serialized objects. If developers want to store them in a different kind of way they can do as they desire and set up a program that converts them to serialized objects and sends them to server.

[4]One restriction is of course the requirement to be serializable, which is not a restriction after all because all messages have to be serializable anyway. If they were not they could not be sent as we will show in detail later on.

Chapter 4

Privacy and Security

From a programmer's perspective it would be easiest if all phones would constantly broadcast their location to the server and to other phones and whoever wants to find out someone's location can simply search him in a huge database on the internet comparable with Google Maps. So why should this simple access be artificially restricted?

4.1 Security Concerns

4.1.1 Why?

Throughout history and all over the globe, common people have been subdued, observed, and controlled by authorities. Some leaders, kings, emperors, dictators, and even democratically elected politicians have at all times abused their power and misused their subjects for their own sakes. Some religious leaders and sect gurus have been and are still exploiting their followers and control their lives. One key aspect of successful control is information about the victim. Dictatorships like the former German Democratic Republic and the Nazi regime spanned a tight control net in order to find and isolate or eliminate enemies of the system. Gaining the same information about individuals has been simplified a lot with modern communication systems. Therefore the concerns about data security deserve to be taken very seriously, even in the democratic western world.

4.1.2 Data Collectors

There are many companies that earn money by collecting, interpreting, and selling data. Even though this could be discussed in an abstract manner, we want to make this issue more tangible by exemplarily looking at the one outstanding company which also provides the OS for our applications: Google. Supplying mankind with great software for free on the one hand and on the other hand storing humongous amounts of information about every user mainly to show individually tailored advertisements on the internet. The more money they earn, the more and the better software they can provide to increase their coverage on the users. When someone is

using Google Groups or GMail and is logged in, this finetunes this person's search preferences and interests profile. Obviously also Android is geared towards this aim with the inherent extra advantage of providing information about the user's location as discussed in the beginning.
How far this forced subjection under Google's rules has gone already demonstrates the first commercial Android end device, HTC's *G1*: Without creating a Google account it cannot even be used [13]. Here we see the strings attached to the 'free' OS: The manufacturer does not pay with money, the user 'pays' with his personal information, though.

4.1.3 Specific Public Sensing Concerns

Who is actually bothered with data security? It is mainly the people who deal with it every day on a deeper level than the user level. We dare say that the majority of cell phone users do not think about who has knowledge about their location. These average phone users need to be told by the software providers which personal information is revealed to whom in the same way that users without native knowledge need to be made aware of e.g. how to create a safe password. Even though from a legal perspective the users have accepted the Terms of Service, most likely most of them still have no clue what they accepted. So exploiting users with below-average IT knowledge should not happen, even if it is legal.
In the following paragraphs we will discuss how to reap the benefits of public sensing technology without broadcasting personal information to everyone according to the well-known principle 'As much as necessary, as little as possible'.

4.1.4 Data Security Aware Public Sensing

General Idealistic Guidelines

One vital element of a privacy aware public sensing arrangement is a server whom the client trusts that his information is neither sold to a third party nor misused internally. For instance, location information that is sent for the purpose of traffic jam prevention is to be used for traffic jam prevention only, unless the user explicitly allows other usage. This also means that the user decides which personal information the server is allowed to disclose to other clients. Applied to the scenario where someone picks up someone else with the car, the default setting should be to only disclose the car driver's location within a small time frame around the estimated time of arrival.

Google's Approach

Latitude is Google's user location management system. You can reveal and disclose your location to friends on an individual basis. So are you on the safe side if you hand pick the friends who are allowed to know where you are? No, a glance at the Privacy Policy [17] shows you that big brother is still watching you. The privacy policy states that:

> *"If you use Google Latitude on a mobile device, in addition to other information, we collect battery life information and tie it to your Google Account."*

'Why would I care about Google knowing about my battery life?', one may think, and continue reading. One should not read over *'in addition to other information'*, however, which loosens all restrictions with regards to what kind of information they collect. Thus everyone who uses Google *Latitude* allows Google to collect any information imaginable.
It also does not come as a surprise that Google does not comply with the rules we postulated for data security aware public sensing—which we made up before we read the legal documents. The default setting is not least disclosure of information and sometimes it is not even possible to restrict it:

> *"Certain of our products and services allow you to opt-out of certain information gathering and sharing or to opt-out of certain products, services, or features."*

Google's Terms of Usage [18] is far from only using the information for explicitly granted ends:

> *"By submitting, posting or displaying the content you give Google a perpetual, irrevocable, worldwide, royalty-free, and non-exclusive license to reproduce, adapt, modify, translate, publish, publicly perform, publicly display and distribute any Content which you submit, post or display on or through, the Services."*

In simple words, they can do with the collected data whatever they want, also selling to whoever they want:

> *"You agree that this license includes a right for Google to make such Content available to other companies, organizations or individuals with whom Google has relationships for the provision of syndicated services, and to use such Content in connection with the provision of those services."*

What if someone paid a lot of money to get the location of someone in order to kill him? Why should Google not disclose the information, assuming they are not told that the person will be killed?
To put it in a nutshell, Google's Privacy Policy and Terms of Service state that, with regards to Google, the user has no privacy whatsoever. All kinds of information that exists can be collected by Google, they can do with it whatever they wish, shout it from the roof tops or sell it secretly to whosoever wants it.
One has to keep in mind that the majority of smart phone users—since that will become the vast majority of all people in developed countries—are just users who do not read privacy policies because they do not have time, do not want to be bored, do not care, and want to start using their new phones. So legally it is clearly their fault. Time will show if awareness of privacy and data security will grow, if people will adapt to being comfortable with disclosing information about themselves, or if many will still reveal more private data than they actually want to.

The Need for Encryption

Personal information sent by a phone to a server, e.g. someone's location, can also be received by a third party. In order to prevent this, we need to encrypt all data sent from both server and client. Are mobile telecommunication signals not encrypted anyway? Normally they are, the GSM standard allows to switch off encryption at times with heavy net usage, though [1]. In terms of serious security this is as good as no encryption at all, so we have to implement our own encryption. On top of that we need to make sure that both server and client can be uniquely identified and that no one can fake being another client. If that were not the case, someone could send a big number of messages to the server saying there is a traffic jam on that road. The server would broadcast this information and the navigation systems would lead people around this road so that the traffic jam faker could drive on an empty road.

4.1.5 Implementation of Encryption

To encrypt we should use the asymmetric RSA system that uses the fact that it is an extremely costly operation to undo the multiplication of two very high prime numbers if they are both unknown and can therefore be made extremely safe with high prime numbers. Every user has a public key that is shared and can be known by everyone. The sender encrypts the message with this public key and sends it. It can be only decrypted with the receiver's private key which he must keep secret. This ensures that the client's location information sent to server A can neither be decrypted by server B nor by any other client.
Since the public key is known to everyone, though, the previously mentioned fake traffic jam scenario is still not prevented. We suggest to take a user's password or make him create a new one and compute a cryptographic hash value of this password using e.g. SHA-2[1]. This SHA-2 value makes it impossible to get the password out of it without major brute force attacks which means that the password is not known to the server. Whenever a client or a server sends a message to someone, along with the serial number of the phone his personal SHA-2 code is sent with it encrypted with the counterpart's public RSA key. Therefore every user can be identified uniquely without any possibility to enable someone else to pretend to be him, assuming his password is not disclosed.
A server and a client can either accept all incoming messages or they can require a previous external registration to prevent spam messages from both servers to clients and vice versa.
If large quantities of data are sent from server to client or the other way round, we can apply a common trick. It might be reasonable to only encrypt the key of a symmetric encryption algorithm such as Twofish or AES using asymmetric encryption since this is way more costly to apply than its symmetric counterpart. The rest of the data is

[1]The popular MD5 has been broken and SHA-1 has the same theoretical problems. Therefore the cryptographic hash algorithm of our choice would be SHA-2. It is crucial that the encryption is one-way so that there exists no function to retrieve the password given the hash code of it. Second, the probability that two passwords create the same hash code must be minimized to reduce the chance of brute force attacks [19].

encrypted using the relatively cheap symmetric algorithm which is nevertheless very secure due to the unavailability of the asymmetrically encrypted symmetric key. For an implementation of an anonymous public sensing system see AnonySense [20] which preserves users' anonymity yet provides the server with sensed data.

Chapter 5

Architecture

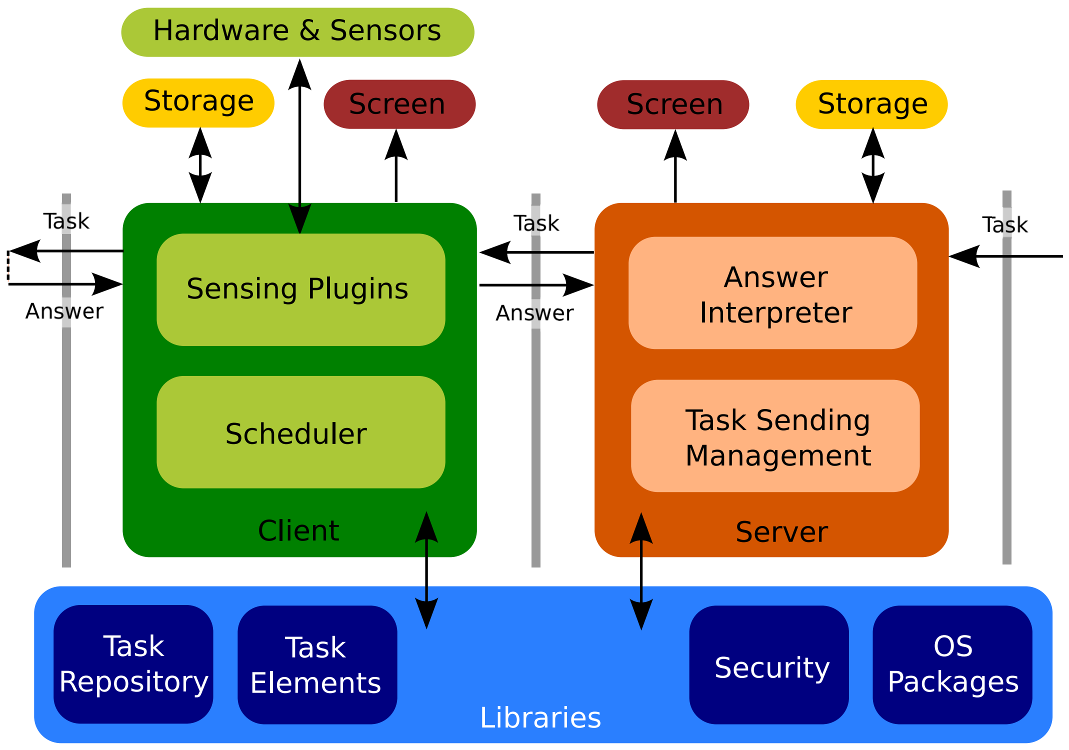


Figure 5.1: The main components of our public sensing system.

Figure 5.1 shows the general structure of our public sensing system. *Public sensing* is the process of distributing requests to gain information for the profit of the public and receiving the returned results. We called the requests *tasks* and the results *answers*. Obviously there has to be some kind of communication channel between the devices that request information and the ones that return them. A device that mainly sends out tasks is called a *server* and a device that mainly receives them and answers them is a *client* which is often mobile. The 'mainly' indicates that even though this will be the setup in most of the cases, there is no restriction that a

client cannot send tasks or that a server cannot give answers. For simplicity's sake, throughout this thesis we will stick to the standard model of a server sending tasks and a client responding. Furthermore a client will be a mobile device, e.g. a cell phone. There are situations where it would be good if the client was capable of sending out tasks as well. The dashed line on the very left of Figure 5.1 means that a client can also send tasks to itself.
A *user* refers to an end-user of a mobile device and a *developer* is someone who manages sending out tasks from the server, changes parameters of tasks, or creates new tasks.

5.1 General Precedent Thoughts

The aim was to provide an architecture consisting of a server program and a client app that communicate with each other. The distinction between this foundational infrastructure and the actual public sensing task was deliberately drawn very sharply. The reason for this is that once the abstract communication structure is up and running, a programmer of another app does not need to understand the details of data transfer, he can simply use the provided classes and concentrate on the individual sensing task according to the well-known principle of encapsulation.
On the other hand, the developer of the abstract infrastructure does not need to be bothered with details of how the precise sensing task looks like.
This is why we want to provide a public sensing communications architecture that is completely independent of the tasks it supports to run. If we do not want the programmers of tasks to hit one wall after another and become frustrated because they cannot do the things they want to do, we have to design the architecture to cover the vast majority of possible requests.

5.2 Abstract Fundamental Architecture

5.2.1 Tasks

What Is a Task?

On an abstract level, a task is one participant's attempt to gain knowledge the other one possesses or can acquire. We explicitly do not restrict it to a server requesting information from a client since we want to design the architecture to fulfill the needs of as many task developers as realistically possible. In public sensing most of the time the server will send out the task to the client to get the information, but it could also be the other way round, that a client requests information from the server about another client's position or about an RFID sensor's location, cf. [12].

Tasks with and without User Interaction

Public sensing tasks can be divided into two groups: One case involves the user to provide the data, e.g. ‘Do these two roads that you see in front of you and that look on the map like an intersection do actually cross or does one run on top of the other?’. In the other case, the device answers without user interaction (e.g. ‘How loud is it where you are now?’). In this case the question is not supplied as text but in code which the device can ‘understand’.

Common Execution Constraints

What both have in common is that every task has an expiration date and an indication of whether it should be run once or periodically. To enable later extension, every task also sends the minimum number of required devices[1] in the area around needed for public sensing and a specification of how big and where this area is. On top of that, execution is restricted to a time frame in which it is allowed.

Executing Tasks

If we wanted to minimize transmission data regardless of the costs, we would only send a very small ID to the server which would then execute the code that belongs to this ID. Since the advantages of a different method outweigh the little additional data traffic, though, we opted for supplying a task with all the ‘tools’ necessary for its execution. With ‘tools’ we mean the code that carries the information of what to do when the task is executed. The huge advantage of having tasks that do not depend on a client that knows their ID and what is hidden behind it is that tasks unknown to the client can be run as well. If this was not the case, for each new and yet unknown task not only a new task class but also a new client class would have to be programmed and deployed to the mobile device.
Not all theoretically acquirable information can be gained without changing the software on the client side, however. Tasks that need to access hardware like sensors may require an imported package or some method on the client side that they can attach to. This hardware could be present in the device or could be plugged into it. This concept is similar to the operator concept of smart homes, cf. [21], where an operator *“can be represented as a black box where some processing is effectuated on a set of inputs, giving as a result a set of outputs.”*. A sound operator, for instance, supplies raw sound data. A sound volume operator would then build on the sound operator’s return values and calculate an average sound volume.
Nevertheless many tasks, especially a theoretically infinite range of user interaction GUIs, can be executed by the client without prior knowledge. For implementation details see Chapter 6 on page 30.

[1]For sending environmental data with imprecise sensors a minimum number of devices might be necessary to compute a fairly accurate average value, for example.

5.2.2 Server

Characteristic of public sensing is the fact that one server collects information from several attached mobile devices. Therefore we need to implement a multithreading system. The server should also be able to send several different tasks to the same client. Since the server does not know in what order the tasks were executed on the client or if some were lost due to bad connectivity, every answer or response to a task contains the ID of the task that was answered to tell the server which question was answered. So e.g. it is sufficient to send the index of a selected item, using the ID the server will find out the question. Attaching only an ID and not the whole task object to the answer reduces its size.

5.2.3 Client

As mentioned before we want to enable several tasks to be executed on the same client and thus need multithreading here as well, also because we want to send and receive messages all the time in the background. Incoming tasks are put into a waiting list that is checked frequently for tasks to execute. Several parameters determine whether a task is executable or not. For details see Section 6.5.1 on page 43. The final decision is up to the user, though. He can choose to run this type of task always or never or decide for each incoming task of this type individually. For a discussion of security aspects see Chapter 4 on page 21.

Chapter 6

Implementation

6.1 Why Android?

Our reasons to choose Android are similar to the ones of end user device manufacturers [13]:

- Comprehensive software platform: Many key applications exist already and can be used.
- Free: No license fees have to be paid.
- Open source: Big parts of the Android platform are published under open source license so that programmers can dig deep into the source code and Android can tap the innovation power of the open source community.
- Equal rights for all apps: Third party apps and pre-installed programs are treated equally which enables new innovative apps to enter the market easily.
- Approved technologies: Programming is made easy with the provided plugin for the popular development environment *Eclipse*.
- Supported programming language: With *Java* a modern, wide-spread, and portable language has been chosen with all its advantages.

6.2 Package Structure

Bearing in mind the wide range of applications, we are now about to implement a prototype with Java that supports the majority of thinkable distributed sensing scenarios. Everything is organized as follows:

- Java project `LiteViewCollection`
  - Package `answerLayouts`
    Library: Several GUI elements that can be changed by the user to answer a task, e.g. text edit.

  - Package `liteGraphics`
    Library: Geographical location and area elements.

- Java project `PublicSensingServer`
  - Package *`publicSensingServer`*
    Application: This contains the applications that runs on the server.

- Android project `PublicSensingClient`
  - Package *`android.publicSensingClient`*
    Application: Here we store the program that runs on the client and several classes needed by both server and client.
  - Package `android.publicSensingFinalTasks`
    Library: All non-abstract tasks that can be sent are to be stored here.

The two packages in italics contain the two classes `Server` and `Client` whose methods start the programs on server and client. The library packages are imported and `PublicSensingServer` imports `android.publicSensingClient` as well since some classes that are needed by both server and client are stored there.
The source code is saved in Unicode (UTF-8).

6.3 Shared Classes between Server and Client

6.3.1 Message and Its Subclasses

`Msg`

The abstract super class `Msg` is spelt in its abbreviated form to prevent confusion with other classes called `Message` and avoid the dragging along of the package names. Two likewise abstract classes are derived from `Msg`: `Task` and `Answer`.

`Task`

The following constrains are needed to create a new `Task`:

- Expiry date. 'null' means it will never expire[1].
- How many seconds shall we wait until we execute it again? '-1' signifies that it is not periodic and shall be executed only once.
- A `TimeFrame` that restricts the execution time. A `TimeFrame` takes start and end date and time. At the present stage, the period is set fix to 'daily' which means that only the hours, minutes, and seconds will be looked at to determine executability. Still we give `TimeFrame` a full `Calendar` object to ease future extension to dynamically change it to e.g. hourly, weekly, or monthly.
  `null` means no time frame restriction[2].

[1] There could still be a `TimeFrame` set that restricts execution, however.
[2] That does not necessarily mean no timewise restriction, however. There could still be an expiry date set.

- An array of geographical areas (`AreaLite[]`) in which execution is possible.[3]
- How many devices need to be in the area so that execution makes sense?[4]

`Task` implements `TaskRunnable` and therefore every object derived from it has to have a method called `run()` which will be called for execution of the `Task`.

`TaskUserInteraction`

This class is derived from `Task` and takes an array of `ViewLite` elements[5] which are responsible for the interface displayed when the `TaskUserInteraction` is run. `TaskUserInteraction` has an empty method `run()` already so that its subclasses do not have to implement it since they are dealt with separately because we have to control the execution of tasks with user interaction to prevent unorganized popping up of new GUIs.
A simple example of `run` that returns a new `Answer` object with only the location is shown in Listing 6.1. First it requests the current instance of `Client`, the private `Client myClient`, and then it gets the location of it.
In Listing 6.2 we see the constructor of a subclass of `TaskUserInteraction`, `TrainLateness`, which asks the user how late the train is. Several objects of subclasses of `ViewLite` are set up (lines 5-11). They will be shown in this order from top to bottom to the user.

Duplicate Detection

If the connection between server and client dies and comes alive again later on, at the present state the server does not know that the tasks it is about to send to the client are there already[6]. For the server every new connection is the same and it starts sending tasks. In order to not litter the display of the client with duplicates and even worse have them execute more often than desired—since each task manages its execution times on its own and therefore n tasks would be executed n times—we allow only tasks that have not been received yet to be enlisted. This check is performed on the client side at the reception of each new `Task`. The server also uses the same kind of duplicate detection before sending tasks, cf. Section 6.4.1 on page 38.
This leads us to an equality check algorithm and therewith to the question: When are two tasks equal? Using the standard method `*firstObject*.equals(*secondObject*)` is not possible because as shown in Section 6.6 on page 43 two objects with precisely the same initial parameters do not have to be the same object using this operator

[3]Some developers *"assume that the context server stores the positions of the corresponding [RFID] sensors."* [12], so an array of areas is required.
[4]Not used yet but necessary if mobile devices are to decide autonomously without recheck with the server if execution shall take place or if a minimum number of sensors is required to get a reliable average value.
[5]View elements such as text edit, time picker, etc. Cf. Section 6.3.2 on page 34
[6]For large quantities of tasks or big task objects a query from the server to the client about which tasks are there already would increase efficiency.

since they could have be created during two different calls of the creation class. Therefore we have to actually compare all relevant parameters on our own by overriding the `equals` methods.
First we had to make sure that all initial parameters—that is all non-abstract classes, pre-existing or self-written, that we use in constructors of any subclass of `Task`—can be properly checked for equality with their counterparts, assuming both objects are instances of the same class. If this is not the case, we can return 'not equal' without further checks.
We allow the objects in the constructors to be `null` to indicate something special (e.g. no expiry date) and therefore have to check if they are `null` before we perform any other checks. We must not return 'not equal' if one element is `null`, however, since obviously both being `null` means they are equal. Equality check for arrays like `AreaLite[]` is done with `Arrays.deepEquals(`*`firstArray`*`,`*`secondArray`*`)`.
`Task` overrides the equals method and returns `true` if all variables are equal to their equivalents in the other `Task`. At the present stage, `TaskUserInteraction` overrides `equal` as well and returns the value of its super class (`Task`). The final tasks—that is the classes derived from `Task` or `TaskUserInteraction`—also pass the request for equality check down the line. If desired, an extra parameter check can easily be added, though. Future developers should note that they have to override the `equals` method to ensure proper comparison when creating a new final task.

Our Prototype Tasks

- `GetLoc`: This simple task returns the location of the mobile device. The server can interpret it in several ways, open the door, for instance.
- `SenseLoudness`: Our application of loudness sensing takes the raw data from the microphone and calculates an average value which is then sent back[7].
- `GetBluetooth`: This senses for visible bluetooth devices and returns an array with the devices' names, MAC addresses, and the signal strengths. The retrieved information can be used to communicate with other mobile devices.
- `TrainLateness`: Using this task, the user can report a belated train and supply additional information such as estimated time of arrival.
- `RoadsCrossing`: The user is asked about the arrangement of two specific roads that might cross.

`Answer`

In general, if a `Task` is sent out, an `Answer` is expected as response. These parameters are passed to an `Answer` object:

[7]Since converting this value into decibel would require massive calibrating and is very different for different phones we leave this task for future developers that specify in this direction.

```java
@Override
public Answer run() {
    LocationLite locLite = Client.getCurrInstance().getCurrLocation();
    return new Answer(−1, locLite, null, getId());
}
```

Listing 6.1: The `run` method of the task that only gets the location.

```java
public TrainLateness(Calendar expiry, int periodicSecs, TimeFrame timeFrame,
      AreaLite[] area, int requiredDevicesAround) {
    super(expiry, periodicSecs, timeFrame, area, requiredDevicesAround, null);

    ViewLite[] views = {
            new TextViewLite("What is your destination?"),
            new EditTextLite("Please enter destination.", true),
            new TextViewLite("When was the train scheduled to arrive?"),
            new TimePickerLite(),
            new TextViewLite("What is the estimated arrival time?"),
            new TimePickerLite(),
            new SpinnerLite(new String[] {"Rough personal guess", "Official
                estimation.", "No clue" })
            };
    setViewsLite(views);
}
```

Listing 6.2: The constructor of `TrainLateness`

- The current location of the mobile device. This information is the very essence of public sensing and will be utilized for almost all sensing projects.
- An `Object` that may contain any additional information imaginable.
- A unique ID that tells the receiver to which question this is the answer.
- A time stamp of when it was executed.

An overview of `Msg` and its subclasses is shown in Figure 6.1.

6.3.2 GUI Classes

Views Shown to the User

If a user is involved in supplying the server with information, a user interface is needed. It would have been best to be able to put together any GUI element imaginable on the server and send it to the client where it is displayed. However, since even the most basic bricks such as `TextView` are not serializable, the whole building made up of these bricks cannot possibly be. Furthermore it is not (yet) supported

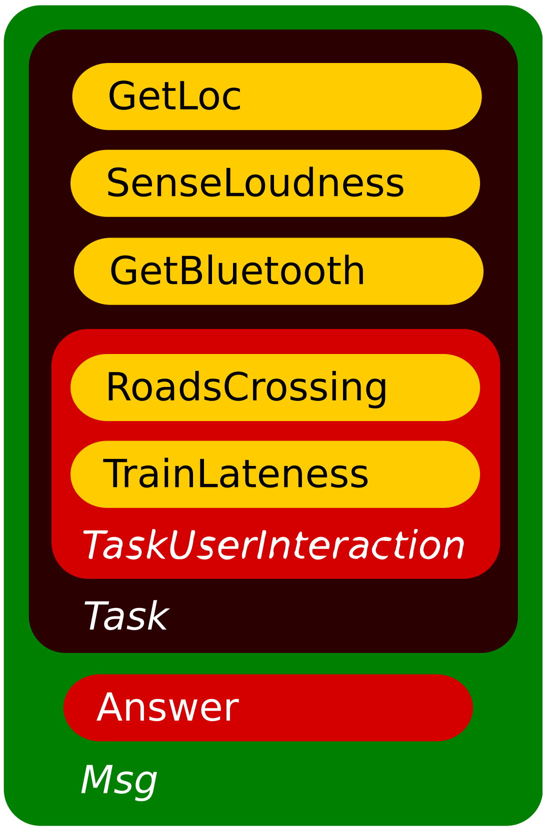


Figure 6.1: `Msg` and its subclasses. Abstract classes are in italics.

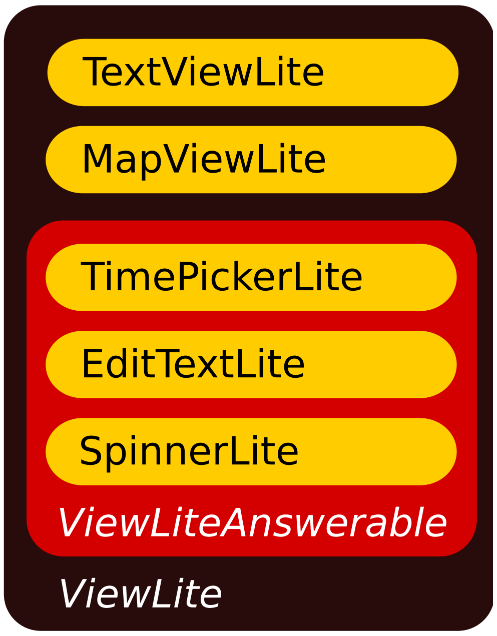


Figure 6.2: The `ViewLite` classes. Abstract classes are in italics.

to read external non-preprocessed XML files and display them. If it was, we could create an XML file on the server or an attached emulated client and send it to the mobile device. Due to these two restrictions, however, we thought it best to rewrite the relevant parts for our purposes in so-called light classes which are serializable. We called them like the Android original class with the suffix `Lite`. Customizability is currently restricted to necessities such as the text of possible items to select in a `Spinner`[8]. For our needs it boiled down to five bare bone rewrites of GUI elements divided into two categories. Figure 6.2 shows a visual representation and below there are details for each class:

- Information to the user
  - `TextViewLite` displays text. You can specify if you want the text to be a hint so that it disappears if clicked on it.
  - `MapLite` creates a button that shows a full screen map when clicked on.
- Ways of answering.
  The following classes are not direct subclasses of `ViewLite` but of `ViewLiteAn-`

[8]A way of selecting one element out of many, similar to radio buttons.

`swerable` which is a subclass of `ViewLite`. This means they implement the interface `Answerable` and therewith `getAnswer()`

- `SpinnerLite`: The user can choose one from many text items.
- `EditTextLite`: Takes user-created `String`.
- `TimePickerLite`: Lets the user pick a time.

These bricks can be combined in any order and number of occurances in a vertically scrollable `LinearLayout`. `MapViewLite` can only be used once due to complicated location object management if one wanted to display several locations[9]. Showing only one map in one task should suffice all practical demands, however, since one task ought to be one straightforward question. At the very bottom there are always two buttons labeled *quit* and *submit*. Hitting *submit* the entered information is sent to the server together with the `Task` that has been answered and *quit* exits the program.
`public String[] getAnswer(String index)` returns a one-dimensional `String` array to avoid parsing in a possible future extension if several items can be chosen.
In the tasks that show a GUI, additional methods should be written to make it easier to retrieve the needed information. Listing 6.3 shows an example. It uses a two-dimensional `String` array because each `ViewLite` element returns a one-dimensional array already.[10]

```java
public static String getRoadsArrangement(String[][] answ) {
        return answ[2][0];
}
```

Listing 6.3: The answer of the third `ViewLite` element (index 2) is requested and the second index is 0 since all other ones are null.

`MapViewLite` is different from all the other classes since it does not show a map but only a button that will show a full screen map if clicked at. This is better than a small map squeezed in between the other GUI elements since the user never has to see both the map and the rest at the same time. That is why `MapViewLite` creates only a button.
`MapViewLite` is given several parameters at creation which are used for pointing the user to the streets the server is interested in. Most important is the center of the possible intersection. The position of the two labels 'A' and 'B' is determined by

[9]When the `ViewLite` elements are inflated and the View elements are created, for `MapViewLite` a button is created and listener is added. If clicked it starts an `Activity` (A module of the whole program connected with one or many displays, called `Views`.) and passes e.g. the `LocationLite` of where the crossing is. This `LocationLite` has to be a global variable so that the listener can access it. Managing different locations with arrays would be possible but not necessary for our needs.

[10]The one-dimensional array of `ViewLite` elements of which each returns a one-dimensional array results in a two-dimensional array.

three variables: The distance in meters how far from the center the labels are to be placed and two angles that determine in which direction from the center the two labels 'A' and 'B' shall be drawn.
When the button is clicked, a new `Activity` will be started and these variables are passed along to `ShowMap`.

Geographical Location

For the same reason, namely lack of serializability, a new location class `Location-Lite` has been created. An object of it can be constructed with either two `doubles` as latitude and longitude or the original `android.location.Location`.

Geographical Area

One determining factor whether a `Task` is executable or not is whether the mobile device is in a specified geographical area. Following the same convention that we introduced before, we added the suffix `Lite`. Both `CircleLite` and `PolygonLite` extend `AreaLite`. Both also have the method `public boolean contains(LocationLite l)` that returns whether a geographical location is inside the area or not. The code for the polygon was taken from [22]. For the circle—that is actually an ellipse if the latitude and longitude values are regarded as two orthogonal axes—point (x, y) is within the ellipse if the following holds:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} \leq 1$$

a is the absolute value of the distance from the center to the point on the ellipse that is farthest east and b respectively north. These distances need to be in degrees of latitude and longitude, however, not in meters. Converting them is done using the formula shown in Section 3.1.

6.3.3 Input/Output

The class `IO` is the foundation stone of our architecture. All the communication between server and client is dispatched by it. When the server is started, it creates two instances of `ServerSocket` with port numbers. When the client accepted both requests, both the server and the client create an object of `IO`. If the client could not automatically establish a connection with the preset IP and port numbers, the user will be asked to enter or change the IP address and port number and then hit *connect*. To keep it simple, the port number of client-to-server communication is one higher than the number of server-to-client communication and only the latter can be modified by the user.

In order to get a better understanding of the underlying structure, the constructor and some of the public classes will now be looked at in detail.

The Constructor

`IO`'s constructor takes two `Sockets` (for ingoing and outgoing data) and sets up the `ObjectOutputStream` and its input equivalent. After that it starts a thread that regularly checks a queue for outgoing messages and sends them.

`public synchronized void toSendQ(Msg msg)`

Since `send` is a private method of `IO`, everything that shall be sent needs to be passed to `toSendQ`. The internal `SendThread` of `IO` will then send it. All subclasses of `Msg` are serializable and can therefore be sent.

`public Msg receive()`

In contrast to its counterpart `send`, `receive` is public so that server and client can have a thread to directly fetch a received message without an extra thread and the resulting additional delays. If `send` could be called directly, it would stall the system if large amounts of data were sent since it is a blocking method.

`public synchronized boolean removeFromRecQ(Msg msg)`

If a user explicitly chooses to never execute a task, it will removed from the list as if it was never received so that it does not litter the GUI any more.

`public synchronized Task getXTask()`

Whenever the client is connected to the server and is not busy with another task, it checks for auto-executable tasks in the background, that is, it calls this method and receives a `Task` to execute. For a precise definition of auto-executable in contrast to user-executable see Section 6.5.1 on page 43.

`public synchronized LinkedList<Msg> getAllMsg()`

This method returns all received messages. The client calls this method and filters out the `Tasks` to display them to the user who can then specifically pick one to run.

6.4 Server's Classes

6.4.1 The Core Class: `Server`

To avoid circular references in the build path, we store only the classes that only the server needs here and import all the rest.

The `main` Method

When this method is called, `Server` performs the following steps:

1. Set up the `ServerSockets`.

2. Read in a text file which tasks to send to each client. Remove duplicates, cf. Section 6.3.1 on page 32.
3. Create the `Tasks` using the information acquired in the previous step.
4. In an infinite loop do the following until interrupted:
   (a) Accept the `Sockets`. If there is no client or if it has not responded, wait until receiving a response.
   (b) Create an instance of `IO` called `myIO`. So if the server serves multiple clients, it will create a new `IO` for communication with each client.
   (c) Start a new thread that continually checks for messages in the inbox.
   (d) Add the tasks to send to the send queue using e.g. `myIO.toSendQ(roads)` where `roads` is an instance of `RoadsCrossing`.
5. Close `ServerSockets`.

The Receive Thread

Every thread, that is, every connection with a client, has its own instance of `IO`, therefore it has to be given to the constructor of the `ReceiveThread`.
If something is received that is not `null`, `react2msg(msg)` is called and the received message is passed along. If the received message is `null`, this is a signal that the connection has been lost. The receive thread breaks the loop then, kills the other socket as well, and dies.
Since the method that returns the received messages is a blocking method we do not need any delay here. After one message has been received it will wait for the next one.

React to Received Message

Having received a message, this object `Msg` is given to `private static Msg react2msg(Msg msg)`. Here we first check if it is an instance of `Answer` since so far we have only sent `Tasks` from the server and received `Answers` from clients, not vice versa. If it is an `Answer`, we want to find out to which `Task` it is the `Answer`. This is done by checking if the ID `String` that belongs to the answer matches with known IDs of subclasses of `Task`.
Let us assume we received an Answer to the task `TrainLateness` and have a closer look at what is done. Now as the next step a new ‘dummy object’[11] is created like this:
`TrainLateness train = new TrainLateness(-1, null, null, null, 0);`
Using this ‘dummy object’ we can get the `ViewLite` array of our received `Task` because at creation of `TrainLateness` the GUI elements such as `TimePickerLite` or `TextViewLite` are set up. For instance, if the answer to the third `ViewLite` element

[11]It is not possible to use a static method to retrieve the `ViewLites` since we want to access the different `ViewLite` elements which are unique for each `TaskUserInteraction`

```
System.out.println("The train from "+ a.getLocationLite() + "to
        + TrainLateness.getDestination(answ) + "is "
        + TrainLateness.howLate(answ)
        + "mins late and will probably arrive around "
        + TrainLateness.getRealTime(answ) + ".");
```

Listing 6.4: Displaying the received information to the user.

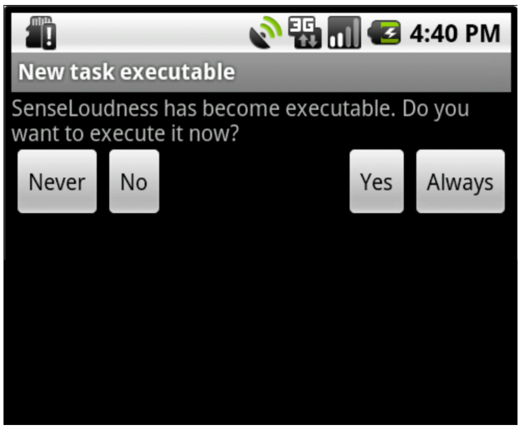
is *'A below B'* then this third element could be theoretically a comment if the third element was `EditTextLite`. That is why we have to find out what kind of `ViewLite` element the third one actually is and that is what we need the 'dummy object' for. The retrieved `ViewLite` array is then given to `public static String[][] interpretAnswer(Answer answ, ViewLite[] views)` together with the `Answer`. This method calls the methods `getAnswer()` of each of the the `ViewLite` elements. Their return values are then saved in a two-dimensional `String` array and returned. Now we have a two-dimensional `String` array which contains the answer. Somewhere. We do not have to fiddle around with the indices, however, but can simply give this array to static methods of the class that filter out the bits and pieces of information we need. For `TrainLateness` the code listed in Listing 6.4 produces an output like this:

> *The train from (lat=48.783781, lon=9.181682) to Berlin main station is 5 mins late and will probably arrive around 12:05.*

Appropriate software would then be able to find out that this information is about the train from Stuttgart to Berlin that was scheduled to depart at noon and would feed it into the real time traffic surveillance system.
But why this detour of first passing the `String`s of the `Answer` to another method just to receive yet another `String` array? Can we not just read the `String`s directly? For some `ViewLite` elements this is true. For others like the `SpinnerLite`, however, what is sent from client to server is '6', for instance, which means the seventh element has been chosen. The `SpinnerLite`'s method `getAnswer()` interprets it correctly though and returns the `String` of the seventh element. For the sake of simplicity and generality all `Answer`s have to 'go through' the method `interpretAnswer(...)` even though some return unaltered.

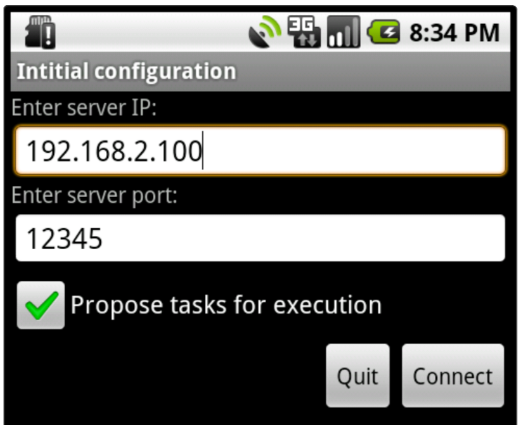
6.4.2 `CreateTasks`

As shown in detail in Section 6.6 on page 43, this class can be edited by the user, creates new tasks, and stores them on the harddrive in a serialized form. Since this class can be called independently from the core class `Server`, tasks can be supplied at runtime.



SenseLoudness has become executable. Do you want to execute it now?

Figure 6.3: The background thread found a task to execute.



Enter server IP: 192.168.2.100

Enter server port: 12345

✓ Propose tasks for execution

Figure 6.4: We are not connected yet.

6.4.3 `TransferTasks`

This class transfers serialized `Tasks` from a file system location via a socket connection to the `Server`. The parameters folder path, IP address, and port number are given in this order. The big advantage this class brings is that someone who wants to supply a task does not have to have access to the server but can simply establish a socket connection. Additional security measures can be set up on demand.

6.5 Client's Classes

6.5.1 The Core Class: `Client`

Initialization

The method `onCreate(Bundle savedInstanceState)` starts the `LocationManager`. We use the same `LocationListener` to check for both GPS and network provider location information. After that, it checks if the server with the preset IP and port number is available. If yes, it will show a list of received tasks but as soon as it finds an executable task, the user will be asked if he wants to execute it (see Figure 6.3). If no connection can be established with the given socket parameters, the user will be faced with a configuration screen where he can alter IP and port number (see Figure 6.4).

Establishing the Connection

Using the IP address and port number, the client now tries to establish the connection in `connect()`. It creates an instance of `IO` which other activities can access using `public static IO getCurrIOInstance()` since it is crucial that all input and output is dispatched by the same instance of `IO`.
Furthermore a thread that frequently checks for incoming messages is started. It also calls `startNewTask()` to make it ready for executing a new task. Whenever

this method is called, it means that the previous GUI is finished and a new one can start.
Switching to the `View` that shows the received tasks concludes the initial setup of a client that has just been created.

One GUI Task at a Time

There are two ways a task can be executed: First, the program looks for an executable task in its queue and grabs the first one. Second, the user clicks on a task to explicitly execute this one.
We want to have strictly non-parallel executions of tasks so that no task that is asked to be executed can push aside another task. This concept is important for tasks that require user interaction but right now we treat all tasks the same[12]. The goal of this synchronous approach is to prevent the popping up of tasks whenever they become executable. The user might be busy answering one `TaskUserInteraction` and we have to give him the time he needs for that and prevent by all means that the task is gone all of a sudden.
Neither an automatically called task can supersede a manually clicked one nor the other way round. We also want to forbid the thread that periodically executes tasks to do so if another task is running already.
To clarify our non-parallel approach we have to say that it does not mean that the user has to wait until the data has been sent to the server until the next task can be executed. As soon as a task calls `startNewTask()` it tells the scheduler: ‘I am off, the stage is free for another task!’ Note that this can be right after starting a thread that may continue sending data for a long time. This means that even though the task is not finished from the device’s perspective—it still needs to send the answer—as soon as the user hits *submit*, a new task can be shown since the screen is ready to show a new GUI.
Given this principle of giving each task the full attention until it is done, we thought the most elegant way of achieving this would be that whenever a task is about to finish (and only then!) it calls `startNewTask()` which starts a thread that checks for new executable tasks. It cannot just call the next task in the line since there might be none which can change all the time, however, with the reception of new tasks, with moving into an execution area, or with progression of time. Doing it this way, we do not need to set a flag whether tasks can be executed and have threads continually check this flag which would result in artificial and unnecessary delays.

[12]If desired, the principle of non-parallel execution can be interpreted in a less strict way: All tasks that do not require user interaction and that have been given permission by the user to always execute, could be moved to another thread that runs in the background and frequently executes the appropriate tasks in parallel to the thread that deals with the other tasks that require user interaction.

Reacting to Received Message

The method `react2msg(Msg msg, boolean allowX)`[13] decides what to do with a received message. For now, only if the `Msg` is a `Task` we react to it. If the user set the execution permission for this task to 'always' or if the task was explicitly clicked by the user, it will be directly executed. For tasks that require user interaction a new `Activity` is started which creates the GUI. For tasks without user interaction the `run()` method is called which takes care of all the rest and returns an `Answer` which is sent back to the server. If a task is set to be executed periodically it calls the `run()` method every time it executes and creates a new `Answer`.
If the task that has been asked to execute does not have the permission to do so yet, `showNewXTask` is called which prompts the user to decide whether to execute it or not. If the task is then given the permission it will enter `react2msg(...)` again, this time with the permission required for execution.

Auto-executable vs. User-executable

We defined two kinds of executability because even when a task is flagged as 'not executable' for the automatic periodic execution check, the user can still invoke it by clicking on it since it does make sense not to forbid the user to override his previous decision but also not to bother him by asking again to early. Therefore:

$$\text{set of auto-executable tasks} \subseteq \text{set of user-executable tasks.}$$

In contrast to parameters like time and location which determine both executabilities, auto-executability is only granted if the user has not clicked 'no' when asked to execute or his 'no' is longer back than a specified period of time.
If it is not a periodic task, that is, it is to be executed only once, it is only auto-executable if it has never been executed.
If it is a periodic task it may be run automatically if it has never been executed yet or if the last execution time dates back longer than the task's period of execution.

6.6 Creating the Tasks

As mentioned before in Section 3.2, we simply serialize the objects that have been created and write them into a file. The file name is the class name followed by a dot, a time stamp, another dot, an index[14], and the extension `.ser`. We chose the dot to separate the class name from the time stamp because a dot is a legal character in a file system name but does not appear in class names. We could have also used the space character but spaces in file names should be avoided, especially when working with command terminals. Unambiguously separating the class name from the time stamp enables us to easily extract both of it for possible future usage.

[13]`allowX` is `true` if execution is definitely allowed either because it was clicked by the user or because this task has been given the right to be always executed without further questioning.
Note that there are two different methods called `react2msg`—one in `Server` and one in `Client`.

[14]The index ensures that two tasks will not have the same file name which could happen without the index if they were received in the same second.

This file is then read from the main program. Doing it this way, the main program can remain unaltered and whenever one needs to modify or create a task, all changes can either be written into the existing file `CreateTasks.java` or this file can be copied into another folder, modified, compiled, and executed.
The server is given one folder which it frequently checks for new tasks. The server remembers which tasks were sent to each client[15] and checks this folder for new tasks and sends them to the client. Neither server nor client have to be restarted to supply the client with a new task which makes it very convenient for both users and developers.
When a class is changed and there are still serialized objects of the old class there, the computer will realize it and throw an error. We catch it and print out a separate error message to the user and tell him the most probable reason for this error. The solution is to delete the old files and create the tasks anew. The source code of our class that creates the tasks is shown in Listing 8.1. If there is a parameter supplied when executed, this is interpreted as the folder path where to put the created serialized objects.

Supplying Tasks via Another Socket

On the server side, a third socket is opened through which serialized `Tasks` can be sent. This has the advantage that people who want to send out tasks do not need access to the server but can simply create the tasks on their device and send them to the server. Security and identity check measures can be applied if needed. Furthermore the server could be easily set up to receive tasks from multiple senders.

6.7 The Life of a Task: An Exemplary Overview

6.7.1 Creation

A new task is born when the required parameters have been entered into `CreateTasks`, the `*.java` file has been compiled to the `*.class` file which is executed then. It creates a separate `*.ser` file with the serialized task.

6.7.2 Sending

The `Server` class reads all `*.ser` files in the folder (either supplied via `args[]` or the the current directory) and sends them serialized to mobile devices if the connection has been established. The steps of deserialization and consequent serialization could be omitted, doing it this way ensures that the objects we have read are valid, though.

6.7.3 Reception

The receive thread of the client checks for incoming tasks all the time, reads our task, and puts it into the queue.

[15]Note that if the connection died and is revived again the server does not remember that it is now connected with the same client as before and treats it as if no tasks had been sent.

6.7.4 Delay Due to Location or Time

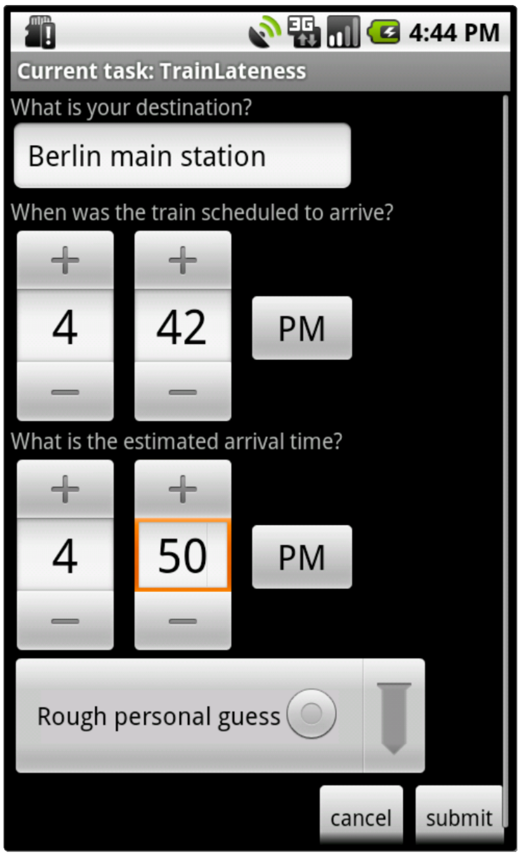
The task is not executable yet because the cell phone is not in the right location or we are not within the time frame of execution. This is shown by a grayed out task name next to the check box which does not react to clicks as well.

6.7.5 Execution

The automatic check for executable tasks picks our task and proposes its execution to the user who agrees. The user could have also clicked on the task.

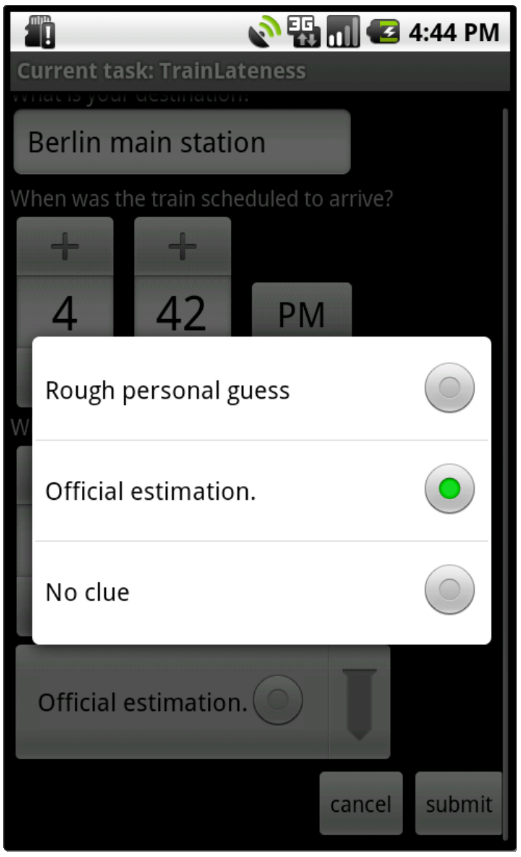
6.7.6 Awaiting User Interaction

See Figures 6.5 and 6.6 for the screens shown to the user.



4:44 PM
Current task: TrainLateness
What is your destination?
Berlin main station
When was the train scheduled to arrive?
4 42 PM
What is the estimated arrival time?
4 50 PM
Rough personal guess
cancel submit

Figure 6.5: The user entered the location and is about to change the estimated arrival time.



4:44 PM
Current task: TrainLateness
Berlin main station
When was the train scheduled to arrive?
4 42 PM
Rough personal guess
Official estimation. ●
No clue
Official estimation.
cancel submit

Figure 6.6: The time entered in the previous screen is an official estimation.

6.7.7 Sending Back the Task

Hitting *submit*, the answer is extracted and sent back to the server and the user either sees a request to execute another task or a list of received tasks like in Figure 6.8.

6.7.8 Displaying the Sensed Data

On the server side the task is examined and we can read the response:

> *The train from (lat=48.783781, lon=9.181682) to Berlin main station is 8 mins late and will probably arrive around 12:05.*

6.8 Using the Menu

Every Android phone has a physical button labeled *Menu* which is very useful for buttons that should be present somewhere but are used fairly seldom and therefore should not fill up the screen all the time. The subsequent images will show all available menus of our application and how they could be used.
The apparently endless stream of proposed tasks for execution wears the user out. So he clicks on the menu to end automatic execution proposal (see Figure 6.7). In Figure 6.8 we see that there are executable tasks but the user will not be asked any more. If he wants to receive proposals again, however, he can simply click on the menu and hit *Configure* to switch it on again. This leads him to Figure 6.9. As soon as he marks the check box ticked, a new suggestion pops up (Figure 6.10) if there are any tasks auto-executable right now.

6.9 Practical Remarks

6.9.1 USB Debugging

Using USB debugging with Eclipse is very convenient especially when testing functionalities such as microphone and bluetooth that are not supported by the emulator. After plugging the USB cord into the PC and the phone, Android will ask you to set the USB connection type. Normally one has to select *USB debugging* but on *Debian squeeze 6.0.2, Kernel Linux 2.6.32-5-amd64* it worked only when 'harddrive' was selected. The phone we used is an *HTC Desire*. With this model, sensing of bluetooth devices often resulted in loss of wireless connection. Since it worked perfectly with *HTC Hero* we assume some hardware issue beyond our control.

6.9.2 Means of Information Exchange

While testing, server and client communicated via wireless LAN since it was conveniently available with no extra costs. For practical use later on we assume a constant internet connection via UMTS of the mobile device.

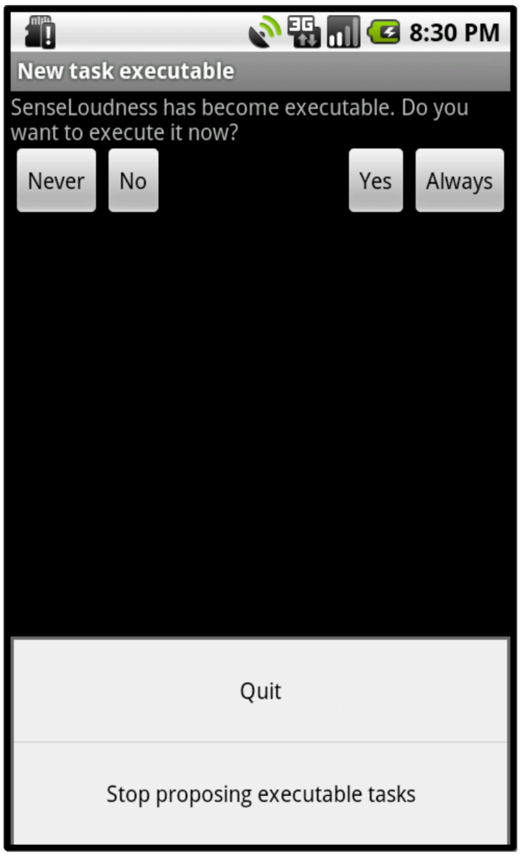

8:30 PM
New task executable
SenseLoudness has become executable. Do you want to execute it now?

Figure 6.7: The user wants to decide on his own which tasks to execute.

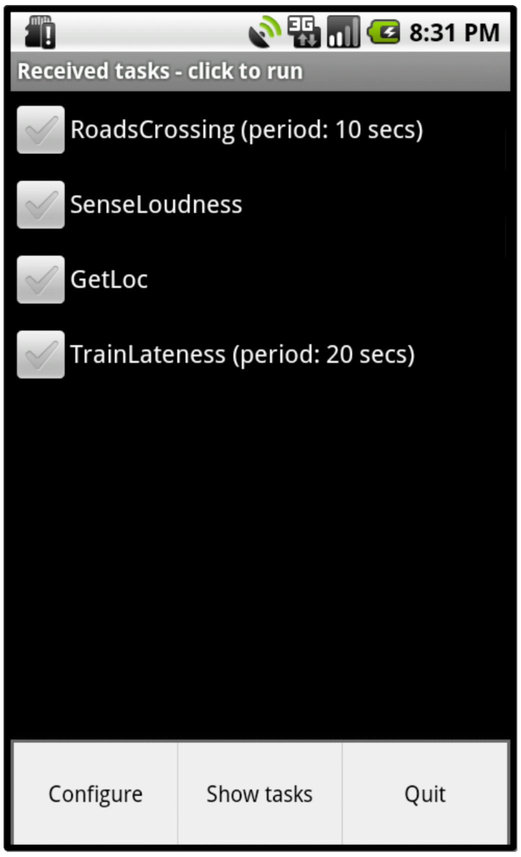

8:31 PM
Received tasks - click to run
RoadsCrossing (period: 10 secs)
SenseLoudness
GetLoc
TrainLateness (period: 20 secs)

Figure 6.8: After having changed his mind he hits *Menu* again.

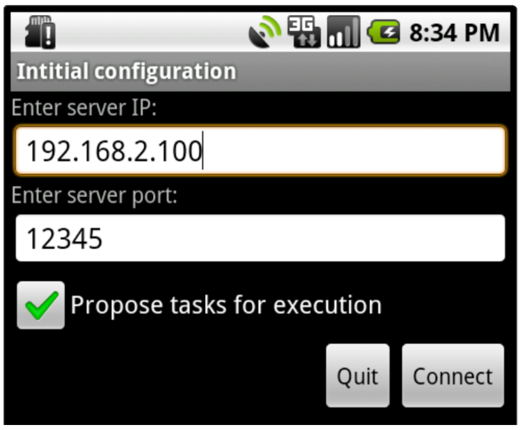

8:34 PM
Intitial configuration
Enter server IP:
192.168.2.100
Enter server port:
12345
✓ Propose tasks for execution

Figure 6.9: The user changed his previous automatic execution setting.

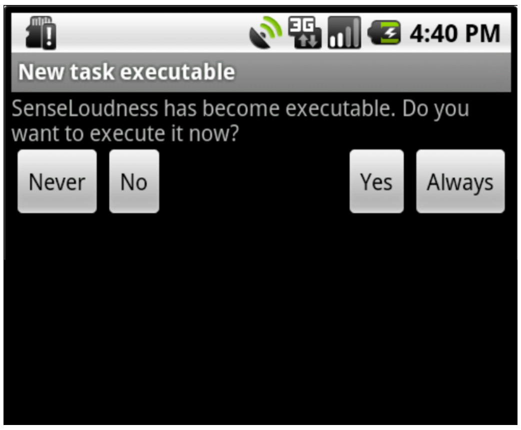

4:40 PM
New task executable
SenseLoudness has become executable. Do you want to execute it now?

Figure 6.10: Without intermediate click an executable task is shown.

Chapter 7

Usage

7.1 Starting the Server

7.1.1 Bundling the Required Packages in a `*.jar` File

In Eclipse we created `pub-sen-server.jar` of the server package and included all imported packages. We put this file in the folder where the server package is:

- `publicSensingServer`
  - `CreateTasks.class`
  - `Server$ReceiveThread.class`
  - `Server$ReceiveThreadTasks2Send.class`
  - `Server.class`
  - `TransferTasks.class`
- *`pub-sen-server.jar`*
- `GetBluetooth.Fri_Jul_08_12-04-26_CEST_2011.0.ser`
- `GetLoc.Fri_Jul_08_12-04-26_CEST_2011.1.ser`

7.1.2 Adding Android and Google Maps to the Class Path

The `*.jar` files of Android and Google Maps are not contained in `pub-sen-server.jar` and therefore have to be added to the class path. Depending on the OS and type of terminal there are several ways to do that [23].

7.1.3 Start the Programs of the Server Package

Assuming that required Android and Google Maps files have been added and that we are in the folder of the server package, the `Server` is started like this from the terminal[1]:

[1]Adding this `*.jar` to the class path as well is rather difficult since an 'application entry point' (a class of the package) needs to be set [24]. We have three classes that can be started, however, so would probably need to have three `*.jar` files.

```
$ java -cp pub-sen-server.jar publicSensingServer.Server
```

We can also specify another folder from where to read the serialized tasks[2]:

```
$ java -cp pub-sen-server.jar publicSensingServer.Server ./tasks
```

Passing a second argument[3] we set the number of the outgoing port[4]:

```
$ java -cp pub-sen-server.jar publicSensingServer.Server . 1489
```

The classes `CreateTasks` and `TransferTasks` are run in the same manner. Only the number and order or passed arguments vary, see below for details.

7.1.4 Arguments Passed via Command Line

	first	second	third
`Server`	source folder of tasks	port number	—
`CreateTasks`	target folder of tasks	—	—
`TransferTasks`	source folder of tasks	IP address	port number

[2]Note that `./` is the Unix way of specifying a relative path. On Windows consoles one would use `\`.

[3]The . indicates to use the current folder. On Windows use here `\` as well.

[4]In the `Server` class, the incoming port is this number plus one, the port where serialized tasks are received plus two.

Chapter 8

Tests

While programming one function, we thoroughly tested this one with different scenarios before we moved on with implementing the next function. In the end we artificially created test scenarios that did not appear in the normal process of programming but will most certainly happen in real life. The most important one is the following, namely the loss of connectivity in different stages of the app.

8.1 Connection Loss

The goal is obviously that the app must not crash whatever happens. Therefore scenarios like e.g. the loss of connection to the server have been tested extensively. Now to the best of our knowledge loss of connection to the server at whatever stage the app is in will not result in a crash but the user will be notified with a `Toast`[1]. In spite of not being connected, the tasks can still be executed. The `Answer`s are saved and will be sent as soon as the user reconnects. When a reconnect has been requested, the client checks if this is the first time it will connect to the server. If the client has been connected before, it will not create a new instance of `IO` but will use the old one and connect this one with the server again. Using the same object as before has the advantage that the send queue and receive queue do not have to be transferred from the old object to the new. Since reconnecting from the server's perspective is the same as establishing a brand new connection, it will send the same tasks to the client again unless told otherwise. On client side duplicates are filtered out, however (cf. Section 6.3.1 on page 32).

8.2 Field Test

8.2.1 Introduction

In order to combine three different tasks to a relatively realistic field test scenario, we decided to take the phone with us on a bike ride and make it sense the loudness every 30 seconds within a time frame of 4 p.m. to quarter past 4. As soon as we

[1]Notification to the user that pops up and stays for a short while.

are less than 20 meters away from the possible intersection, we want to be asked whether these roads cross. Entering the imaginary circle of 15 meters around the house door, we want our location to be sent to the server who would then open the door. Since our mobile device is connected via wireless LAN, it will receive the tasks at home and later on loose connection again. Right after reconnecting it will automatically send the gathered information to the server. Depending on the signal strength of the wireless router located inside the house, the last task (request to open the door) might or might not be sent from outside the door. On a real system this issue would not occur because we would not connect directly via wireless from server to client but over the internet using UMTS.

8.2.2 Setting up the Tasks

An Example: Sensing Loudness

We pass the parameters to the task `SenseLoudness` in the following order:

- An instance of `Calendar` when it shall expire. In our case we pass *null* for ‘no expiration’.
- The period of execution in seconds, *30*.
- An instance of `TimeFrame` within which execution is allowed. This takes the following objects:
  - An instance of `Calendar` that indicates the beginning. We chose *16:00* as hours and minutes and an arbitrary year, month, and day since right now the frequency is constantly set to ‘daily’ and therefore these variables are disregarded.
  - Another `Calendar` that marks the end of the time frame, *16:15* in our case.
- An array of `AreaLite` objects. We give no restriction locationwise and therefore enter *null*.
- The minimal number of devices around before execution can start. *0* in our case.
- How long shall a noise sample be? We give it *1.5*, the value is interpreted as seconds.

How this task and the other ones look like in source code can be seen in Listing 8.1.[2] After having written the tasks in the source code, the file needs to be compiled e.g. from the terminal using the Java compiler:

```
$ javac CreateTasks.java
```

[2]The latitude and longitude values for a location can easily be found e.g. on Google Maps by first navigating to the desired location, then right-click on it and click on ‘What’s here?’

```java
public static void main(String[] args) {
    // [...]
    Calendar cal = Calendar.getInstance();
    long millis = (cal.getTimeInMillis() / 1000) * 1000; // get rid of millis since
        they can't be set. Not doing so might result in unequal if millis are unequal
         when comparing equality of tasks.
    cal.setTimeInMillis(millis);

    // ===== YOU MAY MODIFY FROM HERE ON =====
    // set up execution areas
    PolygonLite p;
    double[] xpoints = new double[] { 48, 50.1, 49 };
    double[] ypoints = new double[] { 8.5, 8.5, 10 };
    p = new PolygonLite(xpoints, ypoints, xpoints.length);
    PolygonLite[] polygons = { p };
    LocationLite center = new LocationLite(48.991587, 8.938981);
    LocationLite house = new LocationLite(48.994784, 8.940993);
    CircleLite[] circleCrossing = { new CircleLite(center, 20) };
    CircleLite[] circleHouse = { new CircleLite(house, 15) };

    // set up execution times
    Calendar endC = (Calendar) cal.clone(); // cloned and overwritten later
    Calendar startC = (Calendar) cal.clone();
    Calendar expiryC = (Calendar) cal.clone();
    startC.set(2010, 0, 0, 16, 0, 0); // year, month, ...
    endC.set(2011, 0, 0, 16, 15, 0);
    TimeFrame tf = new TimeFrame(startC, endC);

    // final tasks
    LocationLite center = new LocationLite(48.991587, 8.938981);
    double angleA = Math.PI * .2;
    double angleB = -Math.PI * .35;
        Task[] tasks = {
                new RoadsCrossing(null, -1, null, circleCrossing, 0, center, 18, angleA
                    , angleB),
                new GetLoc(null, -1, null, circleHouse, 0),
                new SenseLoudness(null, 30, tf, null, 0, 1.5)
                };
    // ===== DO NOT CHANGE CODE AFTER THIS LINE! =====
    // [...]
}
```

Listing 8.1: `CreateTasks` which can be modified by the user

Then we start the program with the following command:

```
$ java CreateTasks
```

Since we supplied no folder it stores the `*.ser` files in the current folder (cf. Section 7.1.4 on page 49).

8.2.3 Performing the Test

We started the Java class `Server` on the server and clicked on the *PublicSensing-Client* app on the phone. After a couple of seconds the tasks were received.

8.2.4 Evaluation

A first glance at the user-executability showed us what we expected:
`GetLoc` was executable because we were in the same house to which we would return—and a couple of meters in front of our house door we expect this task to become executable to send a request to the server to unlock the door.
`SenseLoudness` was executable as well. When prompted with the question when to execute, we hit 'always'. It immediately took a sample of the loudness[3] and sent it to the server:

```
<-received Answer to SenseLoudness
It is an answer. Received at: Sat Jul 09 16:07:20 CEST 2011
Location: loc (lat=48.9949474 lon=8.9415535)
>> Loudness=272
```

`RoadsCrossing` was not executable yet since we had not come close to the possible intersection.
Then we disconnected the USB cord and made our way towards the crossing. Instead of seeing `RoadsCrossing` become active we saw the whole application crash, however. Back home we found out that it almost always crashed when the display was rotated and the view changed from portrait to landscape or vice versa because the current `View` became destroyed and asynchronous threads still referred to it. One line in the Manifest[4] fixed this issue. We wonder why it is not the default setting.
Our next attempt was more successful, the task `RoadsCrossing` became executable as planned a couple of meters away from the center of the crossing. We answered the question and hit 'submit'. Later after having reconnected this is what we saw:

```
<-received Answer to RoadsCrossing
It is an answer. Received at: Sat Jul 09 16:53:20 CEST 2011
Location: loc (lat=48.99158682701874 lon=8.938920462783813)
>> Crossing
```

[3]The loudness value of 272 in this case has only qualitative worth, the higher the value, the louder it was. Range and value heavily depend on the type of cell phone used.

[4]We added the line in italics to the activity section:

```
<activity android:name=".Client"
android:label=@string/app_name"
android:configChanges="orientation|keyboardHidden">
```

After a while `SenseLoudness` stopped its periodic execution because it was outside of the given time frame. When entering our property the task `GetLoc` became active and asked us whether to send our location to the server in order to open the door for us. In a real scenario where tasks can also be stored on the mobile device,[5] the user would set the execution permission to 'always' so that the door would open automatically without asking.
To sum it up, our field test fulfilled all requirements:

- Executability changed properly with moving to another location.
- The task restricted by a time frame stopped being executable.
- Storing the tasks in the queue and sending them once the connection was available again worked.
- The loudness was sensed periodically without user intervention.

8.3 Usability

8.3.1 Tests on People without Previous Android Knowledge

If our app is self-explaining and easy to use for people who have never used Android before, it should fulfill the requirements with regards to usability of all other users as well. Therefore we gave the mobile device to three people who had no previous experience. In general, without further advice they understood what to do. Some things were confusing, however:

The Keyboard Pops up

If there is a text edit on the screen, the keyboard pops up automatically and fills up half of the screen and leaves the users overwhelmed and puzzled about what to do. So we decided to switch the automatic keyboard pop up off.[6] Still, as soon as the user touches the text field, a keyboard will appear.

The User Notification Stays Too Short

The relatively long text to inform the user about lost connectivity and how to reconnect disappeared too soon so that the user could not read the whole message. We did not see a way to shorten the text so we made the `Toast` appear twice as long.[7]

[5]This would be fairly easy to implement since the algorithms that save and read serialized tasks are there already. We would only move them from the server package to the client package since there we store the classes needed in both or outsource to an extra package.

[6]This line added in the `onCreate(...)` method of the Activity did the job:
```
this.getWindow().setSoftInputMode
  (WindowManager.LayoutParams.SOFT_INPUT_STATE_ALWAYS_HIDDEN);
```

[7]For the display length there are only two options to choose from: short and long. We simply put the text in a loop so that it gets called twice

Do I Have to Comment?

The previous hint text[8] of a text field was *'Enter comment here.'*. A user was puzzled what to enter so we changed the text to *'Enter optional comment here.'*.

[8]The text that disappears if the field is clicked.

Chapter 9

Peek into the Future

9.1 The Big Picture

The rapid evolution of mobile devices can be compared with the upcoming of personal computers in terms of coverage of the holders. A couple of years from now, the vast majority of people will probably possess a cell phone that is capable of doing far more than just text messaging and making phone calls and therefore the percentage of cell phones that have the potential of being public sensing clients will approach 100%. Given that a fairly high percentage of people have a cell phone and carry it around with them almost wherever they go, the potential and sensing power to be unleashed is enormous.
So how can we tap this source best? Some technology enthusiasts or altruists with spare time might download the app even if they have no immediate gain, but in order to reach the masses we have to think about ways to compensate or reward a user for sensing data like temperature or a traffic jam. We suggest coupling collecting the information with reaping the benefit, thus everyone who agrees to send his anonymized location information will in return receive traffic jam warnings. To kick off a sensing project until we reach a minimum number of required sensing contributors we need other incentives, however. So we should make the app as easy and fun to use as possible in order to reach also people with below-average IT interest and knowledge. The GUI has to be visually appealing; icons are to be preferred over text. Another marketing strategy would be supplying personal or social sensing applications for free and later on suggest the installation of public sensing apps once users have built trust in the software provider. A financial compensation of the user for his efforts is also thinkable but might sum up to a significant amount of money since often many sensings need to be taken.
Our strategy would also be emphasizing the security and privacy issue by guaranteeing the user not to misuse or disclose their personal information. If possible, we would not even collect personal information but depersonalize as early as possible and use encryption for all potentially insecure communication channels.
The main challenge we see for the future will be to ensure that all mobile devices will not only be able to communicate with each other on a few standardized channels for text messaging, phone calls, and video calls, but will instead be capable of receiving

tasks, displaying them, and responding to them regardless of the operating system. Right now there are several rather separated nets of mobile devices that exchange information only within their group but not to mobile devices that run another OS. So the ideal solution would be a universal mobile device OS. An important requirement would be that it is open and can be used free of charge so that if there are no major drawbacks on the software side, it will be superior to other possible operating systems concerning the financial aspect as well.
Maybe the issue of multiple OSs will be resolved or become irrelevant if some mobile device operating systems are left behind because manufacturers move over to the ‘big ones’. Perhaps de-facto standardization like *jpg*, *png* or *pdf* will also be established in the mobile computing sector, e.g. *Java* and *XML*. Reverse engineering of *iPhone* apps could happen as it happened with Microsoft’s file formats. Another possibility would be e.g. running an Android virtual system in the background on one of Nokia’s *Symbian* based phones. The last and definitely least preferable option of choice would be rewriting the client side for another OS. Still that would be a manageable effort compared with the gain of millions of potential users. If interpreting serialized Java tasks is not doable on the client, one could reassess possible conversion to XML.
Regardless of how soon and how well inter-OS-communication will work, the huge potential of literally millions of smart phones with sensors in the hands of people who can assist in supplying information is astonishing and might revolutionize areas like live traffic surveillance and possibly locking systems as mentioned throughout this thesis. On the other hand, one has to respect the privacy of users and should not misuse their data. The first step is to raise awareness of what happens with their private information. If people feel deceived by cell phone providers they might become very skeptic toward new technology and therefore will not be willing to install public sensing apps.

9.2 Extension Possibilities of Our Framework

So far, we have one server that disseminates tasks to many clients. But why should there be only one server that sends out tasks? One client could also connect with multiple servers and the received tasks would be merged into one GUI.
At the present stage, the server can receive tasks to send out via a socket connection with another device. It would be fairly easy to implement the reception of tasks from several sockets. Furthermore, as proposed in the outline of the architecture, a client could be enabled to send out tasks as well.
There is one GUI element that we initially wanted to create until we could not think of a realistic application for it. This would be a way of selecting n out of m items from a given list. If the need is there to add this one to the `ViewLite` elements, it can easily be added. If further customizability of these GUI elements is desired, extra parameters can be supplied on demand.
If changes to the software have been applied that result in its incompatibility with the old version, the user should be prompted to update. The server could also query

the client at the very beginning to determine the platform version installed and then send the applicable versions of tasks.

9.3 How to Shape the Future

Most certainly the future will bring ideas of how to use public sensing into developers' minds that are unheard of now. Therefore we deemed it crucial to establish a framework that can be extended to suit the vast majority of developers' needs. The sample applications that we provided—retrieve location, sense loudness, report train lateness, report roads crossing, sense bluetooth devices—are useful indeed, but that was not the main objective to write them. They ought to be regarded as prototype models of what can be done with the architecture we wrote and therefore cover many of its functionalities. We have representatives of autonomous tasks and some that require user interaction; some need access to special hardware like the microphone or bluetooth sensor; some are designed for periodic execution like sensing the volume, others are only to be executed within a closely defined area, like the question if these two roads cross.
All of these tasks are mainly there to demonstrate what can be done and we made sure to use every GUI element that we introduced at least once. Once the framework of server, client, and optionally another device that sends tasks to the server side is up and running, changing the parameters of a new task is very easy and straightforward: Just add the required variables to the constructor of the task in the correct order, run the program and the server will pick them up at runtime and send it to the clients connected with it without restart of server or client.
Having boiled down the creation of a task to a simple one-line procedure should enable future developers to fully concentrate on the content of the task without having to bother about transferring, scheduling, serializing, and all the rest.
Creating a new class of tasks could be done in a couple of minutes if it is a question to the user that consists only of the already existing light versions of `View`—which should be the case for almost all imaginable questions. If hardware needs to be accessed, the creation of a new class might require deeper digging into the source code. Nevertheless the main architecture is present already and we hope that it will enhance the development of numerous public sensing apps now that researchers and other developers can pour all their creativity into the content and function of new public sensing systems. May this framework ease the transition to the era when smart phones linked with each other are capable of doing things unthought of nowadays, possibly even greater than today's desktop computers.
But users will still be able make to phone calls.

Declaration of Independence

I hold these truths to be self-evident, that I wrote this thesis independently and that I listed all relevant references below. The help of numerous internet tutorials, blogs, forums, and the like is greatly appreciated and cannot be listed individually due to its sheer enormity. This thesis or parts of it have not been published yet.

Patrick Leucht
Vaihingen/Enz, Germany
July 22, 2011

Bibliography

[1] A. Becker and M. Pant. *Android—Grundlagen und Programmierung.* dpunkt.verlag, 2009.

[2] H. Weinschrott et al. *StreamShaper: Coordination Algorithm for Participatory Mobile Urban Sensing.* Seventh IEEE International Conference on Mobile Adhoc and Sensor Systems (MASS'10), 2010.

[3] E. Kanja et al. *MobGeoSen: facilitating personal geosensor data collection and visualization using mobile phones.* Springer-Verlag London Limited, 2007.

[4] N. Maisonneuve et al. *Citizen Noise Pollution Monitoring.* Proceedings of the 10th International Digital Government Research Conference, 2009.

[5] H. Lu et al. *SoundSense: Scalable Sound Sensing for People-Centric Applications on Mobile Phones.* MobiSys '09, 2009.

[6] A. Steed et al. *Using tracked mobile sensors to make maps of environmental effects.* Springer-Verlag London Limited 2006, 2005.

[7] H. Weinschrott et al. *Participatory Sensing Algorithms for Mobile Object Discovery in Urban Areas.* IEEE International Conference on Pervasive Computing and Communications (PerCom), 2011.

[8] M. Haklay et al. *OpenStreetMap: User-Generated Street Maps.* Pervasive computing, published by the IEEE CS, 2008.

[9] A. Campbell et al. *People-Centric Urban Sensing.* Proceedings of the 2nd annual international workshop on Wireless internet WICON 06 (2006), 2006.

[10] A. Campbell et al. *The Rise of People-Centric Sensing.* IEEE Computer Society, 2008.

[11] `http://services.defra.gov.uk/wps/portal/noise`, July 22, 2011.

[12] H. Weinschrott et al. *Efficient Capturing of Environmental Data with Mobile RFID Readers.* Tenth International Conference on Mobile Data Management Systems, Services and Middleware, 2009.

[13] H. Mosemann and M. Kose. *Android—Anwendungen für das Handybetriebssystem erfolgreich programmieren.* Carl Hanser Verlag, 2009.

[14] http://googlepolicyeurope.blogspot.com/2010/04/data-collected-by-google-cars.html, July 22, 2011.

[15] http://arstechnica.com/old/content/2008/10/google-gears-enhances-geolocation-with-wifi-positioning.ars, July 22, 2011.

[16] http://simple.sourceforge.net, July 22, 2011.

[17] http://www.google.com/mobile/privacy.html, July 22, 2011.

[18] http://www.google.com/accounts/TOS, July 22, 2011.

[19] E. Sanfèlix. http://www.limited-entropy.com/crypto-series-hash-functions-sha2, July 22, 2011.

[20] D. Kotz D. Peebles M. Shin C. Cornelius, A. Kapadia and N. Triandopoulos. *AnonySense: Privacyaware people-centric sensing.* In Proc. ACM 6th Intl Conf. on Mobile Systems, Applications and Services (MOBISYS 08), 2008.

[21] M. Daoud. *Modeling Distributed Applications in a Smarter Home Infrastructure.* University of Stuttgart, IPVS, 2011.

[22] D. Finley. http://alienryderflex.com/polygon/.

[23] http://download.oracle.com/javase/1.5.0/docs/tooldocs/#general, July 22, 2011.

[24] http://download.oracle.com/javase/tutorial/deployment/jar/appman.html, July 22, 2011.

Nomenclature

AES	Advanced Encryption Standard
GPS	Global Positioning System
GSM	Global System for Mobile Communications
GUI	Graphical User Interface
ID	Identification
IP	Internet Protocol
IT	Information Technology
JAXB	Java Architecture for XML Binding
JDOM	Java-based document object model for XML that integrates with Document Object Model (DOM)
LAN	Local Area Network
MAC	Media Access Control
MD5	Message Digest algorithm
OS	Operating System
PC	Personal Computer
RFID	Radio Frequency Identification
RSA	Asymmetric encryption algorithm developed and named after Rivest, Shamir, and Adleman
SHA	Secure Hash Algorithm
SI	International System of Units
UMTS	Universal Mobile Telecommunications System
USB	Universal Serial Bus
XML	Extended Markup Language.