UNIVERSITY OF HERTFORDSHIRE
School of Computer Science

MASTER OF SCIENCE IN DISTRIBUTED SYSTEMS AND NETWORKS

Final Project Report

May 2015

"Android Motion Tracker: A live motion display of AMT-powered devices in range"

Loudovikos Geromitsos

Abstract

Motion capture or motion tracking is the process of recording the movement of objects or people. It is used in military, entertainment, sports, medical applications, and robotics (Wikipedia – Motion capture, 2013).

In filmmaking and video game development, where it became the most popular, it refers to recording actions of human actors, and using that information to animate digital character models in 2D or 3D computer animation. In motion capture sessions, movements of one or more actors are sampled many times per second and most often the purpose of motion capture is to record only the movements of the actor, not his or her visual appearance.

The motion tracking systems are divided into two categories depending on their implementation: optical and non-optical systems. The optical systems utilize data captured from image sensors while the latter usually rely on non-optical sensors that are placed directly onto the object or person that is to be tracked. For instance, magnetic non-optical systems rely on magnetic transmitters and magnetic receivers that are placed on the tracked object or person. The motion of a person is then measured in real time using magnetic fields (Google – Patents, 1998).

Similarly, the "Android Motion Tracker" (AMT) application aims to record the movement of other devices in its wireless vicinity that are running AMT software. The sensors in the case of our non-optical system are the GPS sensors on each Android-powered device that is being tracked, which are utilized by AMT to constantly receive coordinates, and which are built-in in most Android mobile devices (Android Developers – Motion Sensors, 2015). AMT will display a 2D representation of their spatial movements on the screen.

Prologue

In this book’s first chapter we will begin with an introduction to the project and then present its main objectives and analyse the final time plan.

In the next chapters, we will proceed onto the analysis and design of the software and then onto its implementation.

Finally, the entire project will be evaluated. At the end of this book you can find appended pictures, installation instructions, references and bibliography.

Acknowledges

I am indebted to the following people for their help and support:

To my family – thank you for supporting my studies.

Table of Contents

1. Introduction

This chapter begins with an overview of the problem dealt with in this project, and then presents the project goals and research question.

1.1. Overview

Motion capture or motion tracking is the process of recording the movement of objects or people (or living beings in general).

That definition is rather abstract and it only reveals the common goal of motion tracking systems; to capture and track the motion. Little does it say about how the motion is captured and tracked in real time or about the technical requirements of the motion tracking system.

There are many technologies that can be classified as motion tracking systems and they vary a lot in their implementation, their usability and their capabilities. They ultimately capture some kind of motion one way or another, but they may not work in the same environments and under the same circumstances, and they may have limitations, for example, in their range of operation, in their accuracy and in their performance in general.

In order to better understand the mechanics of motion tracking systems, it is important to categorize them and then proceed to look into some of the real-world systems. They are divided into two categories depending on their implementation: **optical** and **non-optical** systems. The former utilize optical sensors while the latter utilize non-optical sensors.

The optical systems usually utilize data captured from image sensors. For example, in the filmmaking industry it is a common practice to record the movements of human actors in order to use that information to animate digital character models. That is achieved by placing devices on the actors' clothes and bodies which can be read by image sensors in a studio.

The non-optical systems usually rely on non-optical sensors that are placed directly onto the object or person that is to be tracked. For instance, magnetic non-optical systems rely on magnetic transmitters and magnetic receivers that are placed on the tracked object or person. The motion of a person is then measured in real time using magnetic fields.

However, it is not always a requirement to install sensors on the objects or persons to be tracked. They don't always have to be part of the motion tracker's system. They can be discovered and tracked by its system, as long as they are part of the greater system that we call the "physical world". I personally refer to those systems as **real** motion trackers. They can discover moving objects or living beings and track them in real time by exploiting some physical property of theirs. That is what separates them from all the rest of the systems (optical or non-optical).

We will look into three of those in order to grasp the variety in their implementation, their usability and capabilities as well as their limitations. Those are the **sonar**, the **radar**, and the **heartbeat** sensor:

Starting with the sonar, it was originally an acronym that stood for "**SO**und **N**avigation **A**nd **R**anging". It involves using electroacoustic devices to emit pulses of sounds and then listen for echoes that were reflected back from objects. A computer can then calculate the distance from objects which reflected the sound waves based on the time that elapsed between the transmission of a sound and the reception of its echo. The sonar can discover any sound-reflecting objects and it can also track their movements and display them on a panel. It is usually called **active** sonar to distinguish it from another technology under the same name known as the **passive** sonar. The passive sonar listens for sounds and it is able to identify the type of object that produced them and to tell the direction from where the sound came, in contrast to the active sonar that listens for echoes of its own sounds. Sonars are used underwater to transmit sound waves through ultrasonic pulses. Active sonars can discover vessels travelling on the sea's surface, submarines and underwater objects such as big boulders. Passive sonars can identify the sound source; whether it is a human-made vessel or a sea creature such as a whale. Sonars are not used outside of the water because there are faster alternatives that make use of electromagnetic waves. The speed of sound is about 196k times slower than light in the sea water and as a result there is a noticeable time gap between the actual movement of an object and the reception of its echo. However, sound waves propagate well in the sea water and are able to penetrate it for dozens of miles whereas an implementation of electromagnetic waves would require an extremely powerful antenna for the same task. In regard to motion tracking systems, there are a few things to notice about the sonar. It is used in underwater environments, it may be able to distinguish objects from living creatures, it is error prone to disturbances in the sound waves, it has a pretty good detection range and range of operation (although not unlimited), and it has a few other limitations such as the speed of sound (Wikipedia – Sonar, 2015).

The radar system involves using antennas that work as transceivers (i.e. as transmitter and receivers) of radio waves. They emit pulses of electromagnetic waves, in contrast to the sonar that emits pulses of sound waves, and then receive a portion of those waves that are reflected back from objects. The radar is only an active system, meaning that it has to emit electromagnetic waves in order to discover and track the movement of objects. It is error prone to electromagnetic jamming. It takes advantage of the speed of light and it can be practically used in any environment where the electromagnetic waves can propagate (e.g. atmosphere, space etc.) However, it is not used underwater as the electromagnetic waves do not easily penetrate large volumes of water and are strongly absorbed by sea water (PhysLink, n.d.).

Motion trackers owe a lot of their popularity to video games and especially to First Person Shooter (FPS) games which typically display a radar HUD as part of the main in-game interface and where the player can see the movement of enemies and/or allies in a specific area. The most common type of radar in those games is the **heartbeat radar** which is essentially a microwave radar that can sense and locate the heartbeat pulses of other players. The closest implementation of a heartbeat radar in real life is NASA's FINDER (Finding Individuals for Disaster and Emergency Response), a device that has the size of a briefcase and which can tell the difference between human and animal heartbeats. It can detect humans and additionally this technology makes it possible to track them if they are moving. Obviously it cannot detect inanimate objects. The FINDER is currently a prototype and it has a limited detection range depending on the mediums between the human and the device. For instance, it can detect a heartbeat

from 30 meters away in the open air and through 6 meters of concrete (SlashGear, 2015).

Having seen the diversity in motion tracking systems, we need to consider the shortcomings and capabilities in the implementation of the system that is dealt with in this project. The main question of our research is: How to capture and display the motion of other devices in a specific range in our area?

We will look into that in the following section where we'll analyse the aims and objectives of our application.

1.2. Aims and Objectives

The aim of this project is to create a motion tracker application for the Android operating system using the Java programming language, and which should be able to record the movement of other peer devices (i.e. devices running the same application) in its wireless vicinity and display them live on the screen.

Thus, our target environment is the area around an Android-powered device that is running the "Android Motion Tracker" (hereinafter "AMT") application. How large that area will be depends on limits regarding the detection range. Our targets, the objects to be tracked, are other devices in the area that are running AMT and most likely smartphones and tablets.

Initially, the goal was to create a peer-to-peer (P2P) application where the AMT would behave both as a client and a server and would communicate with its peers in a wireless network. The program would be able to detect other devices in its wireless range that are running the same program (i.e. peers) and exchange information with the peers that are in range. It would constantly transmit GPS data of its host operating system to its peers via Wi-Fi or Bluetooth as well as receive the same information from them. This was not possible due to certain limitations that are explained in a later section during the analysis of the project, and as a result classic client-server architecture will be used instead, where all Android devices will communicate with a central web server.

The result of this project should be a working Java networking application for the Android operating system, which will consist of a GUI program (i.e. a program with a Graphical User Interface) that will mainly feature a radar display that will show the position of other devices in real time. The program on a specific device should be able to receive GPS information from satellites, send it periodically to a web server which will register the device and which will distribute that information to all other registered devices, and periodically receive updates of GPS coordinates of devices in range. After successfully receiving GPS data, an AMT-powered device will be able to calculate its current distance from the registered devices and display their position in a 2D map that will be updated in real-time. Ultimately it will display the live motion of its peers on the screen in respect to its own position.

To better visualize it, the main screen should display our own device as a dot at its center and the other devices will also be presented as dots that are moving towards or away from it in real time. The distance of the dots which represent the devices, will be derived from the GPS data that is being constantly sent to the server by the AMT-powered devices.

The research question that is addressed in this project is:

How to capture and display the motion of other devices in a specific range in our area?

Below is a list of the main objectives of the project:

- State and analyse the problem
- Create Use Case Diagrams to show what the system can do
- Identify the user groups of the system and each group's specific requirements
- Create Storyboards to demonstrate the Client's Graphical User Interface.
- Identify what data the system needs to store in order to fulfil the requirements
- Identify the classes of the system
- Create Class Diagrams to describe the structure of the system, show the system's classes, their attributes and their methods, and the relationships among the classes.
- Methodology. Describe how connections are carried out and the communication between the Clients and the Server.
- Receive information (latitude, longitude etc.) from GPS and/or GLONASS satellites
- Configure distance of detection zone (i.e. the area of motion tracking)
- View details about tracked devices (name, distance etc.)
- Live display of motion of tracked devices in a 2D map
- Evaluate the system
- Reference all learning material and code lookups used
- Complete the final report

1.3. What has been learned

Designing and implementing the system that was described above was quite the learning experience.

The application demanded the existence of a server that could handle concurrently multiple connections and support hundreds of users as well the existence of a protocol of communication between the server and the clients. This offered the opportunity to work for the first time on the communication between Java processes and PHP/MySQL based web services using the HTTP protocol. As a result, in the process of building the system, valuable knowledge was acquired in network programming. It contributed in a better understanding of those technologies and thus in improving flexibility of writing programs that enable processes to communicate with each other across a computer network.

An important learning phase was during the work on the AMT client, which required switching from familiar IDEs like NetBeans and Eclipse over to the Android Studio IDE which is the official IDE created by Google for developing Android applications using Java. Android Studio proved very difficult and mind-boggling to use at first, mainly because of having been used to working with NetBeans which is much simpler to use and has a cleaner look and feel. Although it was a matter of time to get used to the new environment after reading a few tutorials on websites such as OpenSignal (OpenSignal, 2013) and Vogella (Vogella, 2015), there were many technical problems that significantly slowed down the development of the AMT client. The first problem that came up was the inability of the Android Debug Bridge to detect Android devices that were connected via USB to the main PC (a desktop computer) that was used for the development and as a result work had to be transferred at times to a secondary laptop computer as it could successfully detect the devices and ultimately deploy an application package (APK) for live testing. But, even so, there were frequent problems with viewing the debugging logs and identifying errors in the code. For instance, sometimes the application would crash and there was no stack trace available to narrow down the problem and fix the problematic code.

It was also a first time developing an application for Android, a choice that was made in good faith of my Java programming skills. It was not as nearly easy as expected, but a few books came to my assistance. Namely, “Internet Programming with Java” by J. Panayiotopoulos, “Program Development in Java” by B. Liskov and J. Guttag, and most importantly “The Busy Coder's Guide to Android Development” by M. Murphy. The aftermath of all the studying and research was to gain knowledge on the applied Android technologies and configurations such as manifest files, layouts, Gradle scripts, Activities and lifecycle callbacks.

1.4. Time Plan

Work on this project officially began in early February of 2015. A detailed project proposal was ready by the middle of March. The table below illustrates the amount of time that was required for this project and for each task separately:

Task	Effort
1. Detailed Project Specification / Plan	30 hours
2. Requirements analysis	40 hours
3. UML diagrams	10 hours
4. Research	100 hours
5. Implementation / Programming	200 hours
6. Writing report	160 hours
7. Testing	30 hours
Total Hours	570 hours

It was estimated that the analysis and design of the entire system as well as the implementation of the server program would have been carried out by the end of March in order to proceed with the implementation of the client program.

During that period, many hours were spent on research, which where fitted in between the days for the analysis and design, and which involved reading material related to object-oriented design, graphical user interfaces, general Java programming and Android development. A very helpful book was M. Murphy's "The Busy Coder's Guide to Android Development". On the Internet, the "Android Developers" and "Stack Overflow" websites were excellent sources with quite a lot of helpful material that was applied while creating the AMT client. The initial time plan for the research was exceeded with more than a hundred hours spent.

Although the server program was completed only a few days after the estimated time, the implementation of the client was troublesome and it had to be combined with many tests and efforts which resulted in the implementation of the client being many weeks overdue.

It was estimated that the work on the client program would have been completed by the end of April; however that was not the case until the end of May. Meanwhile, work was being done on the project's report to make up for the lost time. It required approximately a hundred and sixty hours and most of it was completed on time by the mid of May, apart from several sections that would be added later on due to the unfinished client program.

The remaining time was used for correcting bugs – mainly on the client – and for testing, although tests were also being carried out throughout the implementation of the system, during the various coding phases.

2. Analysis and Design

In this chapter we discuss the motion tracking systems of the present and what we want to achieve in our own implementation. We give a brief of related implementations and justify the implementation for our own system. Finally, we proceed to discuss the design of the system where we identify the user groups of the system, present a use case diagram to show what the system can do and a storyboard to demonstrate the client's Graphical User Interface.

2.1. Present System

Nowadays, there are countless of motion detection and tracking solutions that make use of various technologies to detect and track objects and/or living beings (e.g. humans and animals). They are used for many different applications and in many different environments. For example, there are radars that detect and track aircraft, sonars that do the same for vessels and submarines, optical systems for computer animation and digitization of motion used by film making industries, heartbeat sensors such as NASA's FINDER and those are only a few that we previously examined from the many technologies and implementations in existence. Some of them are old news such as the radar, while others are relatively new technologies that used to be science fiction such as the heartbeat radar. Finally, as we'll see in the next section, there are plenty of applications of GPS trackers since smartphones and by extension GPS receivers have become so popular. Specifically for the Android OS, which is our target platform, there are hundreds of map-based GPS applications.

2.2. Problems in the Present System

Although there are numerous motion tracking systems to choose from, most of them are expensive and/or intended for professional or military use and are not widely known or accessible to the public. There are many affordable technologies, such as sonars with GPS units that cost only a few hundred dollars (Brad Wiegmann, n.d.), but still they are not widely known to the public, they are intended for specific tasks and most importantly they are far different from what we want to achieve throughout this project.

The rise and popularity of smartphones has given us cheap access to a variety of sensors such as GPS receivers, barometers, accelerometers, magnetometers, gyroscopes, pressure sensors, and wireless transceivers (e.g. Wi-Fi and Bluetooth adapters). A few of these sensors are enough to implement a cheap MT system with low requirements that everyone can obtain and use.

The Android Operating System, which is currently the leading OS in the smartphone market, is notorious for providing APIs and software development kits to work with all the aforementioned sensors. Most smartphones nowadays are equipped with most of those sensors, and at least a GPS receiver.

There have been numerous mobile applications that take advantage of the GPS receivers of smartphones to deploy GPS trackers. Real-time GPS trackers can be classified as motion tracking systems as they track the motion of the devices that are

running the application, although they are not motion sensors as they don't sense and discover any devices.

However, all these applications are map-based with focus on location. They utilize services such as Google's maps to display the tracked devices on a map in real-time, using their GPS coordinates.

What this project aims for, instead, is a radar-based implementation with focus on motion and if possible, additionally, on detection/discovery. Just like in FPS videogames and popular sci-fi movies (such as "Aliens") where a motion tracker displays the movements of allies and/or foes on a radar-like GUI. There aren't, at this time, any Android-based applications (or any applications for smartphones in general) that fit to the description.

2.3. Suggested Solution

The plan is to develop an Android application of a radar-like tracker using the Java programming language.

Ideally, the AMT should have been able to discover other devices in the wireless vicinity of the device running it and communicate with them in peer-to-peer fashion, even with Internet disabled. There could have been only two ways to achieve that – by using either Bluetooth or Wi-Fi for the device discovery and communication. The devices would then exchange their GPS coordinates which they would get through their GPS transceiver or the Internet according to their nearest cell tower and cellular tower triangulation (A-GPS).

Bluetooth would allow our device to scan for other Bluetooth devices, connect to them through service discovery, and transfer data to and from other devices (Android Developers – Bluetooth, 2015). We would want to discover only AMT-powered Bluetooth devices so each device would propagate special beacon packets to announce that it's running the AMT (InfoWorld, 2014). However, using Bluetooth wouldn't have been a viable solution since it only allows a limited number of communicating peers and since class 2 or 3 devices such as smartphones can only achieve short-range connections of about 10 meters (Security Laboratory 2007; CNET 2012). A master device can communicate with up to seven slaves in a piconet (Wikipedia – Bluetooth, 2015), and thus Bluetooth is not an option in our project where we want an arbitrary number of peers and a decent detection range of at least 100 meters.

That leaves us with Wi-Fi as the only option to implement a peer-to-peer network and to wirelessly discover peers. Each device would implement a protocol based on TCP/IP to craft special wireless beacon packets to announce that it is running AMT software. Wi-Fi P2P would then allow the AMT applications to connect to other nearby AMT applications without needing to connect to a network/hotspot or the Internet. Wi-Fi P2P utilizes the Wi-Fi Direct technology that allows Android version 4.0 or later devices with the appropriate hardware to connect directly to each other via Wi-Fi without an intermediate access point (Android Developers – Creating P2P Connections with Wi-Fi; Android Developers – Wi-Fi Peer-to-Peer, 2015). It can operate at up to 100m and it can transfer data at the speed of 2.5 to 3 Mbps which is fairly decent. Thus, the minimum requirements for an Android device to be able to run AMT would be version 4.0 of the OS and Wi-Fi Direct hardware capabilities. As of 2015, and according to Google, devices that have versions of the OS older than 4.0 are less than

7% of all the devices that have checked in to Google's servers (Android Developers – Dashboards, 2015). Although, not every smartphone supports Wi-Fi Direct, there are thousands that do according to the Wi-Fi Alliance's website (Wi-Fi Alliance, 2015). Most modern smartphones support it and that makes it a good candidate for the implementation of the AMT.

Unfortunately, further research showed that the Wi-Fi adapter cannot be used in monitor mode to analyse beacon packets and wirelessly discover peers. In order to put the Wi-Fi Network Interface Card (NIC) into promiscuous monitoring mode, an application requires root access to the system (i.e. administrator access), provided that the Wi-Fi adapter's driver supports monitor mode (Stack Overflow, 2011). Rooting a device is not an easy task for the average user and it would void the warranty of her device (e.g. smartphone, tablet etc.). We would want to monitor for devices that broadcast beacons over Wi-Fi to announce that they are running AMT and then connect to them rather than connecting to all the devices that have Wi-Fi direct enabled. Furthermore, Wi-Fi Direct is limited in the number of devices that it can pair and for "one to many" connections to be possible a device has to become the group owner (Stack Overflow 2012; Wi-Fi Alliance 2015). Thus, Wi-Fi Direct is also not a viable option.

With no other option for peer discovery and peer-to-peer communication, there's a significant deviation from the original plan and we'll have to reside to a classic Client-Server architecture to achieve the same results.

Once again there are two options. To either use a mobile device (e.g. a smartphone) as the server or a setup a server on a desktop computer. The former option would put a lot of the burden of the network's traffic on a single device which would also need to have its 3G or Wi-Fi enabled to be able to communicate with the rest of the devices over the Internet and to also have GPS enabled in order to constantly receive its coordinates. All of that would drain the device's battery very fast, and thus it's not a viable option.

The proposed solution is to setup a webserver and configure the AMT to send its data to that server.

The webserver will be able to receive GPS data and store it in a database and retrieve it whenever a device requests it. Each AMT client will constantly receive GPS coordinates from satellites and send them to the server which will create a new database entry in its table or update an existing row. Finally, each AMT client will be able to periodically query the server which will return a JSON array of devices stored in the database along with their coordinates. AMT will use those coordinates and those of its own to calculate their distance from the user's device and display them on a radar GUI.

2.4. Design documents

2.4.1. Use case diagrams

For the needs of the analysis we've prepared UML use case diagrams. In **Appendix A – Figure 1** and **Appendix A – Figure 2** we illustrate the use case diagram for the client and server respectively.

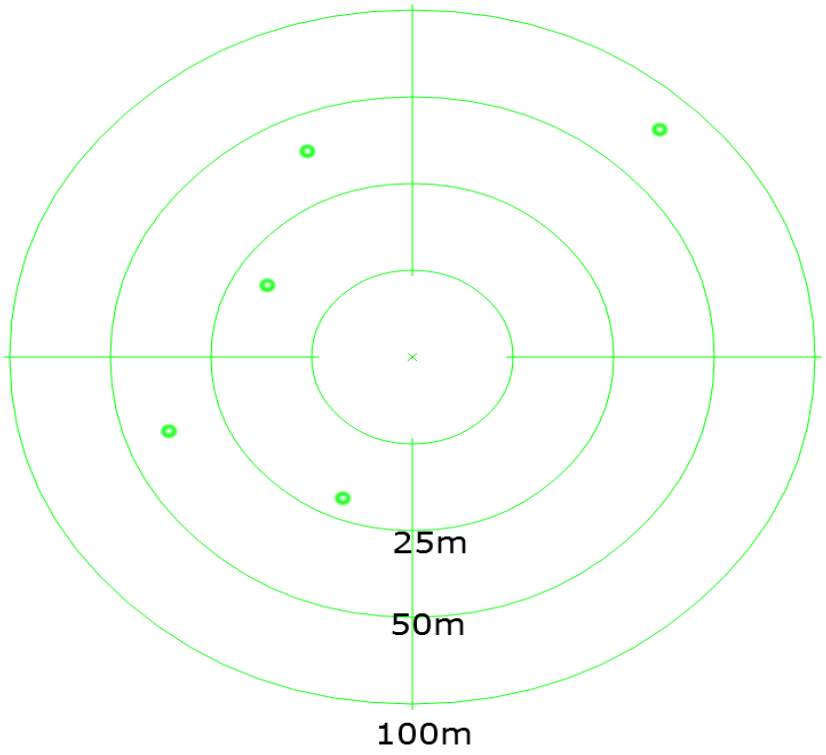
The server should be able to accept asynchronous requests from the client to either store its device's information or to return a list of devices from the database (it has two separate web services for that purpose). When a client sends its data to the server in a POST request message, the server queries the database to check if the device is already there and if it is it updates it, otherwise it creates a new row in the database to store the new device. If a client doesn't send updates (i.e. POST request messages) within a set time frame, then the server removes the device from the database. Whenever a client requests a list of devices in range, the server checks with the request's payload for a provided **radius** and coordinates, performs calculations and queries the database for devices that are between certain longitude and latitude values (according to calculated minimum and maximum values).

The client should be able to receive GPS coordinates and send them to the server (along with device-specific data such as device name) in a POST request message in order to get registered in the database, and it should be able to send POST requests to the server in order to receive information about devices in range. The desired range may be configured by the user; otherwise the default value will be sent to the server. The data that is received from the server can be used in two ways. First, they can be used to combine the received coordinates with the client's coordinates and then calculate the distance of the detected devices from the client's device, in order to display their position on two-dimensional radar. Secondly, they can be used to display a list of detected devices along with information such as their device names and their distances in kilometres.

The users of our system are administrators of the webserver and ordinary users of the client such as smartphone owners.

2.5. Storyboard

Below is a rough storyboard of how the AMT radar GUI should look like. The point at the center represents the user's device and the green dots represent the current position of other AMT-powered devices that have been registered to the server and that the user's AMT has retrieved from it since the last request that it made. Each circle represents a circular field around the user's device and the radius of each circle the distance between the border of that field and the device. By splitting the maximum tracking range (set to 100 meters for this example) into circular areas, the user can better estimate her distance from each one of the discovered devices.



3. Implementation

3.1. Description

The Server side was written in PHP and MySQL and the AMT Client was coded in the Java programming language, an object-oriented language ideal for the implementation of our system, since it is the official language for Android development and the one promoted by Google, the company that develops the Android OS.

Hardware/Software Requirements:

For this project I'll be using the **Android Studio IDE** which is a free, open source Java IDE that can run on all popular operating systems. It is developed by Google and it is the official Integrated Development Environment (IDE) for the development of Android applications.

Android Studio was also chosen for the development of this project as it offers a well-structured and friendly user interface with various automations for faster and cleaner coding, efficient project management and authoring of bug-free code. To name some of its features, it supports syntax highlighting, code formatting plus removal of trailing spaces, automatic import of libraries and code tooltips with Java documentation (JavaDocs).

In order to program in Android Studio, it was required to have installed the Java SDK from Oracle, the Android SDK tools, and certain APIs for Android in order to develop the application, design the Graphical User Interface, and make use of device sensors and adapters.

In order to build the project in Android Studio or in another IDE, it is required to have installed at least Android 4.0.3 (**API 15)** "SDK Platform" and "Sources for Android SDK" through the Android SDK Manager.

The final system can work with at least a pair of Android devices (i.e. two or more devices are needed) and they have to be equipped with a GPS sensor, and have access to the Internet via 3G or Wi-Fi in order to communicate with the server.

- Target platforms for development and demonstration of finished product -

For the development:

Development Environment:	Android Studio IDE 1.2.1.1 (Build #AI-141.1903250)
Java:	1.7.0_45-b18; Java HotSpot(TM) 64-Bit Server VM 24.45-b08
System:	Windows 7 version 6.1 running on amd64

For the testing and demonstration:

Devices:	1. Sony Xperia S smartphone with Android v4.1.2 (Jelly Bean) 2. Samsung Galaxy S T805 tablet with Android v4.4 (KitKat) 3. LG g2 d802 smartphone with Android v4.4.2 (KitKat)

The client can run on any device with an Android Operating System of version 4.0 and higher that has a GPS sensor, a Wi-Fi adapter and/or 3G Internet connection. Of course two or more devices are required in order to demonstrate how a device displays the motion of the other devices.

3.2. Software development

The server program is installed on a webserver that supports PHP such as the Apache webserver. We will upload there all the PHP scripts that are required by the AMT to discover other devices in range. Those will be mainly two web services, one for registering the user's device and/or update its coordinates and another one that returns an array of the devices in range upon a request from the client.

The communication between the server and the AMT client is asynchronous and stateless; meaning that there won't be an open socket connection between them and each request made to the server will be unrelated to any previous request. The client will constantly send POST request messages and the server will reply with response messages and treat each request as an independent transaction. In other words, their communication will consist of independent pairs of requests and responses.

The AMT client has a timer according to which it constantly reads its coordinates from the GPS receiver (at least whenever the device is moved). Otherwise, no action is required. It also has a timer to constantly send requests to the server in which it posts any new coordinates that it acquired and requests from it to update its database with the new coordinates.

That said, in the following sections we will separately look into the implementation of the Server and Client program.

3.2.1. The client

The Client features a Graphical User Interface which was built around Google's Radar library and more specifically the package "`com.google.android.radar`". The repository can be found at:

https://code.google.com/p/apps-for-android/source/browse/Radar

The Google Radar (Google Code, 2008) is a sample application which offers the tools to build a fully featured Radar application by embedding it in the project (Stack Overflow – Embedding a Radar from Google, 2012). It offers three classes. Those are the RadarActivity, RadarView, and GeoUtils. They can be used together to create a simple Radar GUI and to perform latitude and longitude calculations out of the box such as computing the distance in kilometers between two points on Earth.

The entry point in our app is in the RadarActivity class where the main method creates a new instance of the RadarView class, which is essentially our GUI.

The AMT client requires to be granted access to certain device features so before carrying on with the implementation we have to declare the required permissions in our app's manifest file, the "AndroidManifest.xml". The manifest file is important for every Android project and every application must have it in its root directory, because it provides the system with the necessary information to run the application's code. Among other information, it names the Java package which serves as a unique identifier for the application and it points to the main Activity where the code is initiated (Android Developers – App Manifest; Android Developers – Manifest.permission, 2015).

We set the following permissions:

ACCESS_COARSE_LOCATION: Needed by the LocationManager to access approximate location derived from network locations such as cell towers, and Wi-Fi. The difference from the fine location is that the coarse location permission doesn't allow access to the GPS or to any other passive location providers (i.e. without a connection to the Internet).

ACCESS_FINE_LOCATION: Needed by the LocationManager to access precise location from location sources such as GPS, cell towers, and Wi-Fi.

FINE_LOCATION: Deprecated, but kept just in case. ACCESS_FINE_LOCATION should be enough.

ACCESS_LOCATION_EXTRA_COMMANDS: Allows our application to access extra location provider commands.

ACCESS_WIFI_STATE: Needed for determining whether the Wi-Fi is enabled and to access information about Wi-Fi networks.

ACCESS_GPS: Deprecated, but kept just in case. ACCESS_COARSE_LOCATION should be enough.

INTERNET: Allows the application to open network sockets and to communicate over the Internet. In short, it lets our application to use the Internet. It has to be combined with the ACCESS_NETWORK_STATE permission in order to check and validate that there is a connection to the Internet before attempting to use it.

ACCESS_NETWORK_STATE: Needed for determining whether the Internet is enabled and whether the device is connected to a network.

WRITE_SETTINGS: Needed for reading system settings.

WRITE_EXTERNAL_STORAGE: Needed for storing data to external storage.

READ_PHONE_STATE: Allows read only access to phone state. It is needed for reading certain information about the device, such as its manufacturer, its brand and its model.

3.2.1.1. Class RadarActivity

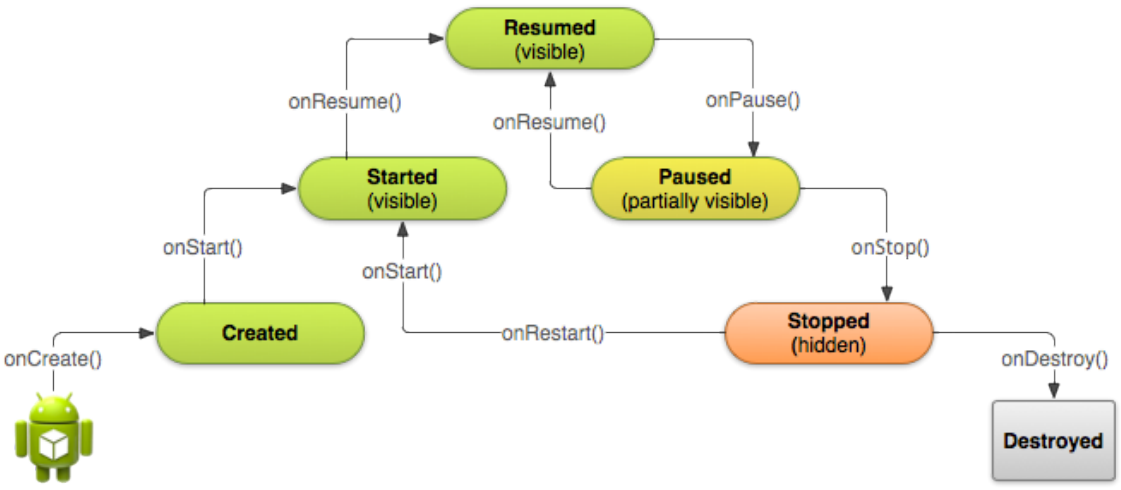
The Android system always initiates the code in an Activity instance, in other words in a class that extends the Activity class (`android.app.Activity`). The child class is also called an activity. In our application, we have a single activity class which is also where the starting point in our code is.

That is the **RadarActivity** class which is responsible for creating and updating the RadarView GUI, and for creating a LocationManager that monitors for changes in the position/location of our device. An activity is like a page in our GUI, and a single, focused thing that the user can do. Our system has a single activity, because that is where the entire radar interface is created - in a single page. In order to use our activity class with *Context.startActivity()* we have added an <activity> declaration for RadarActivity in the manifest file.

The activity class does not have a main() method. Instead, the Android system has specific callback methods for starting and ending an activity and they correspond to specific stages of its lifecycle (Android Developers – Activity; Android Developers – Starting an Activity, 2015). These callback methods are the following seven: *onCreate()*, *onStart()*, *onResume()*, *onRestart()*, *onPause()*, *onStop()* and *onDestroy()*.

During the life of the activity and depending on the user's actions, the system calls any of these methods. An activity class does not have a classic constructor that is named after the class. Instead, a new instance is created by the *onCreate()* method which is called and run just once to create and initialize the activity, and ultimately the last callback is the *onDestroy()* method that completely removes the activity instance from the system's memory. The *onDestroy()* method does not need to be implemented since the system automatically calls it when the user closes the application or kills its process.

The lifecycle methods reflect the different states that an activity can have, as depicted in the figure below that shows the changes that can happen to an activity's state as the user interacts with it.



Source: (http://developer.android.com/training/basics/activity-lifecycle/starting.html)

The above illustration of an activity's lifecycle shows six possible states, although we usually regard the "started" and "resumed" states as one and refer to it as the "running" state. An activity has the "created" state when it is initialized by the *onCreate()* method and the "destroyed" state when the system makes a callback to the *onDestroy()* method to remove it from its memory.

The "created" and "started" states are only transient, because the system quickly moves from them to the next state by calling the next lifecycle callback method. That is, after the system calls *onCreate()*, it quickly calls *onStart()*, which is quickly followed by *onResume()*.

From all the six states, only three of them are static, meaning that an activity can exist in only three of those states at any time, for an extended period of time. The three static states are the "resumed", "paused", and "stopped" states.

If an activity is in the foreground of the screen and the user can interact with it, then the activity is active and running. It has been created and started and maybe also – if it was previously paused on stopped - resumed.

If it has lost focus and goes in the background or it is partially obscured by another activity but still visible (in case that the other activity that is in the foreground doesn't cover the entire screen or is semi-transparent), then the activity is paused and it cannot receive any user input or execute any code, but it is still in the system's memory.

Finally, an activity is stopped when it is in the background; completely hidden and not visible to the user. For example, it is paused when the user hits the back button or minimizes it. It cannot receive any user input or execute any code. An activity always goes through the "paused" state and the *onPause()* method is always executed before going to the "stopped" state. When in the "stopped" state it also maintains its state in the memory, however, the system can terminate its process in cases of low memory.

Method: onCreate()

The first thing to do in this method is to call the *onCreate()* method of the parent Activity class which is essential according to the official documentation (our code will not compile otherwise) in order to have a Context assigned to our activity and in order to avoid any *SuperNotCalledException* exceptions from being thrown (Stack Overflow, 2013). The rest of the code in this method will be run in addition to the existing code in the parent's method.

Next thing to do is to define a layout resource with *setContentView()* for the UI of our activity and to get the context of the current state of the application/object. We use the "radar.xml" layout where we've styled our activity's UI (e.g. set a black background color etc.). We get the context for our newly created activity, which will allow it access to application-specific resources and classes (Android Developers – Context, 2015), and we transfer control to the new instance of Activity.

Now that the activity is properly set up, we make a couple of calls to methods of the GlobalHelpClass which are explained later on in detail in the corresponding section. The first call initializes an object that will communicate with a SQLite database where we can store the data that is received from the webserver. The second call returns a

unique ID for our device (e.g. “ffffffff-bbbf-57cb-0000-000023339ea2”) which is intended to be sent to the webserver.

We proceed to initialize the RadarView object, draw it on the layout (“radar.xml”) and pass it the context. Next, we initialize a SensorManager object which will allow our activity access to the device’s sensors if needed (Android Developers – SensorManager; Android Developers – Sensor, 2015), and a LocationManager object which is responsible for monitoring and handling location changes. The LocationManager class provides us access to the system’s location services and it can detect location changes using either the device’s GPS receiver or an Internet connection (assisted GPS). Next, we call the *setTarget()* method of the RadarView instance to update the radar GUI. Finally, we create the sidebar that has the list of detected devices and fill it with a ListView list that is composed of ArrayAdapter table elements. The ListView is defined in the “radar.xml” layout file.

Method: onResume()

This method is called right after the activity has been created for the first time and also whenever the application is restarted after being paused and stopped. Since we unregister the SensorManager listener whenever the lifecycle callback to *onPause()* is made, we reregister it in this method and we also start/restart the animation of the radar screen by calling the *startSweep()* method of the RadarView instance.

Then, we perform a couple of checks in try-catch blocks to determine whether the GPS and Internet are enabled.

If neither the GPS nor the Internet (network connection) is enabled, then we alert the user with a popup dialog to let her know that the GPS is not enabled and therefore it is required to be enabled. We set two buttons on the dialog; one that transfers the user to her device’s location settings where she can enabled location/GPS and another one to cancel and close the dialog.

If according to the LocationManager the Internet (network connection) is enabled on the device, either the 3G or the Wi-Fi or both of them, then we call the *GlobalHelpCalass.isOnline()* static method to perform a check with the ConnecctivityManager that can tell us whether the device actually has access to the Internet and is connected to a network (Stack Overflow – ConnectivityManager to verify internet connection, 2011). If it is not connected, then we alert the user with a popup dialog to let her know that the device doesn’t have an Internet connection and that she should resolve it. We set two buttons on the dialog; one that transfers the user to her device’s Internet settings and another one to cancel and close the dialog.

Method: onPause()

It stops the SensorManager and LocationManager by removing their listeners and thus prevents the LocationManager from monitoring for location changes and receiving location updates. Then we call the *stopSweep()* method of the RadarView instance to stop animating the radar screen. Finally, we change the state of the activity to “Stopped” which is a required transition after going to the “Paused” state.

Method: onCreateOptionsMenu()

This method creates a menu for the application. The menu's layout is created in the "menu_main.xml" file. We add just one item to the menu, which is also defined in the "menu_main.xml", and assign it a dropdown menu so when the item is pressed it will show the user a group of options to choose from. That item is the "detection radius". The user can select a radius value from the item's dropdown menu and then switch to using that one.

Method: onOptionsItemSelected()

This method gets the id of the aforementioned value that was selected in the "detection radius" dropdown menu. Then it carries out a series of if statements (as many as the options in the dropdown menu) to determine which one of the options was pressed. The options represent values for the "detection radius" such as 10 kilometres, 20 kilometres etc. Finally, it makes a call to the *GlobalHelpClass.setRadius()* static method, to set a new value for the radius and it updates the radar's GUI. This radius is sent periodically to a web service at the webserver along with the devices latitude and longitude. The server uses all three variables to return a reduced list of devices that it has stored in its database – it returns only those that are in the specified range around the device.

Method: receiveTaskCompleted()

This method is called when communication with the server has finished, after every HTTP response message. Since every response message contains a JSON array of devices and their coordinates, the method calls the RadarView to be updated with the new coordinates. Then, in a try-catch block, we update the sidebar's list (the ListView) with the new information. The *GlobalHelpClass.retrievePointsAsString()* is called for that purpose, as it parses the aforementioned JSON array so that it can be stored in the ListView's table (the ArrayAdapter). Finally, we display the new table in the list and effectively remove the older one.

3.2.1.2. Class RadarView

This is the main library of the Google Radar (`com.google.android.radar`) and it is mainly responsible for building and updating the radar's GUI.

3.2.1.3. Class AsyncResponce

This is an interface class that has the signatures of two methods that should be implemented by any class that implements the interface. The AsyncResponse is implemented by the RadarActivity class which uses those methods that are designed to be called when the asynchronous communication with the webserver has finished. In

other words, they are called after every HTTP response, hence the name of the interface “AsyncResponse”.

3.2.1.4. Class GeoUtils

This class is used solely by RadarView instances to perform calculations. It is provided as is by the Google Radar library (`com.google.android.radar`) to perform some useful latitude/longitude math. It has two static methods.

Method: distanceKm()

It computes the distance in kilometres between two points on Earth using the spherical law of cosines (Movable Type Scripts, 2015). It takes four parameters. Those are the latitude and longitude of the first point and the latitude and longitude of the second point, which will actually be the coordinates of our device and those of another device. The method returns the distance between those two points, in other words, the distance between the devices. The mathematical formula that is used to calculate the distance is the following:

$$d = \operatorname{acos}(\sin\varphi_1 \cdot \sin\varphi_2 + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \cos\Delta\lambda) \cdot R$$

, where φ is latitude, λ is longitude, and R is Earth’s radius (mean radius = 6,371 km).

Method: bearing()

It computes the bearing in degrees between two points on Earth. The idea here is that the meridians and parallels of Earth are not straight lines. So if we were to begin from a specific point on a meridian or a parallel to go to another point on the same meridian or parallel, the final heading would differ from our initial heading if we were to follow a straight line and that’s because of the Earth’s spherical-like shape. This method is using a mathematical formula called “forward azimuth” to determine the direction/heading (north, south etc.) in degrees where another point is located, in respect to the point where the user’s device is. The mathematical formula that is used to calculate the bearing between the two points is the following:

$$\theta = \operatorname{atan2}(\sin\Delta\lambda \cdot \cos\varphi_2 , \cos\varphi_1 \cdot \sin\varphi_2 - \sin\varphi_1 \cdot \cos\varphi_2 \cdot \cos\Delta\lambda)$$

, where φ is latitude and λ is longitude.

Method: radToBearing()

Converts an angle from radians to degrees.

3.2.1.5. Class GlobalHelpClass

This is a utility class that is used by the RadarActivity and the RadarView and it contains global functions that are static and belong to the class. It is mainly used to create and manage the SQLite database, although it can carry out a few other tasks such as checking for an established Internet connection.

Method: initDatabase()

This method does initialization of a HelpDatabase object which will be responsible for communication with the SQLite local database, and initialization of a DatabaseActions object which will be responsible for querying the database.

Method: retrievePoints()

Simply calls the *retrievePoints()* method of the DatabaseActions object. It is explained in detail in the corresponding section of the DatabaseActions class.

Method: retrievePointsAsString()

Simply calls the *retrievePointsAsStringArray()* method of the DatabaseActions object. It is explained in detail in the corresponding section of the DatabaseActions class.

Method: insertDevices()

This method is solely used by the ReceiveDevicesLocationsTask class which is responsible for contacting the webserver and retrieving a JSON array with information about the detected devices. The JSON array is passed as a parameter to this method which deletes the previous "devices" table in the local SQLite database and creates an updated one as follows:

1) It creates a ContentValues array that can store 5 values, since the webserver returns 5 values about each device: **device_id**, **lat**, **lng**, **D**, and **name**. The goal is to store those in the 5 columns of the local database respectively: **device_id**, **lat**, **lng**, **distance**, and **name**.
2) For every JSON string in the JSON array it extracts the 5 values and puts them in the ContentValues array.

3) Inserts the ContentValues array in the devices table as a row. The *insertOrThrow()* method is designed to take a ContentValues array and insert each of its values in a separate column in a row in an SQLite database.
4) The ContentValues array replaces its values in every iteration with those of the next JSON string and the process repeats until the entire JSON array has been inserted into the database.

Method: getDeviceUniqueID()

This method accesses the preferences that the AMT has stored in the device's storage in order to get a set of **device details** and more specifically to retrieve its unique **device_id**. The **WRITE_EXTERNAL_STORAGE** permission was required to be set in the Android manifest file, in order to use this capability.

If the device_id has not been stored yet it proceeds to create one by calling the getPsuedoUniqueID() method which produces it and returns it in a String. Finally, it stores that String in the device's storage as its "device_id".

Method: getPsuedoUniqueID()

Here I used a well-known technique around the Web for creating a unique ID for an Android device (GitHub – Create Pseudo Unique ID for Android Device, 2014).

This method creates two strings. One that is a sum that is derived from the character length of the following strings put together: **BOARD**, **BRAND**, **CPU_ABI**, **DEVICE**, **MANUFACTURER**, **MODEL**, and **PRODUCT**. Those are certain details of the user's device such as the name of its model and manufacturer. The second string is the device's serial number.

Finally, the two strings are digested into hash values and then combined together to form the unique ID.

Method: getBaseUrl()

This is a getter/accessor method for getting the value of the "baseUrl" attribute that belongs to this class. The "baseUrl" contains the domain of the webserver with which the AMT communicates.

Method: getMaxRadius()

This is a getter/accessor method for getting the value of the "maxRadius" attribute that belongs to this class. The "maxRadius" is the maximum detection radius that we've set for the AMT.

Method: getRadius()

This is a getter/accessor method for getting the value of the "currentRadius" attribute that belongs to this class. The "currentRadius" is the current detection radius that is configured by the user.

Method: setRadius()

This is a setter/mutator method for setting the value of the "currentRadius" attribute that belongs to this class.

Method: isOnline()

This method creates a ConnectivityManager object to check whether the device is connected to the Internet. It returns "true" if it is and "false" if it isn't.

3.2.1.6. Class DatabaseActions

This class is used solely by the GlobalHelpClass class to query the local database that the GlobalHelpClass creates via an instance of the HelpDatabase class. More specifically, there are two methods, *retrievePoints()* and *retrievePointsAsStringArray()*, with which we retrieve all the devices (and their associated information) that we stored in the database the last time.

Method: DatabaseActions()

This is the constructor. It is passed a SQLiteDatabase object as a parameter with which it initializes a field that belongs to the DatabaseAction object that is created by the GlobalHelpClass.

Method: retrievePoints()

This method is used to construct an SQL query string that when executed returns all rows from the "devices" table (of the local "motionTracker" SQLite database) in descending order according to the "distance" column. We store **the results of the raw SQL query** into a Cursor object which is thread-safe and does not need to be in a synchronized block in order to be used by multiple threads (Android Developers – Cursor, 2015). The main reason, however, for using the Cursor object is that it allows us random read-write access to the result set that was returned by the database. For instance, we can extract only the device names from the result or only the distances.

Method: retrievePointsAsStringArray()

This method does the same as the above one except that it returns an array of Strings instead of a Cursor object. The array is constructed as follows:

1. We get the number of rows that is contained in the Cursor's results.
2. We move the Cursor's pointer to the first row (i.e. the first result).
3. We iterate through all the rows (i.e. results) that contain the values from all the five columns of the "devices" table of the database and from each row we only extract the **name** and the **distance**. With those two pieces of information we construct a String in every iteration, one String for every device, using its name and distance. For example, a format like this:

   "device name: Louis / distance: 4.700889296058902 km"
4. We add all the strings (as many as the devices) in the array of Strings.

The final "deviceArr" array that is returned is used to populate the sidebar of the radar GUI, and thus we call this method whenever we need to update it with the new information from the database.

3.2.1.7. Class HelpDatabase

This class is used solely by the GlobalHelpClass class to create a SQLite database called "motionTracker" on the user's device in order to store the data that it receives from the webserver along with its own information in a single table that is called "devices".

Method: HelpDatabase()

This is the constructor. It initializes a new database called "motionTracker".

Method: onCreate()

This method accepts a SQLiteDatabase database as a parameter and executes an SQL query to create the "devices" table that has five columns: **id** (row numbering), **device_id** (a precomputed id for each device), **lat** (device latitude), **lng** (device longitude), **name** (device name), and **distance** (distance from the user's device). For the user's device the distance would be 0.

Method: onUpgrade()

This method accepts a SQLiteDatabase database as a parameter in order to upgrade the current database by executing an SQL query to drop the old "devices" table and importing the new one.

3.2.1.8. Class ReceiveDevicesLocationsTask

This class is solely used by the RadarView class. It is responsible for the asynchronous communication with the "**find_device.php**" Web Service. It extends the **AsyncTask** class which allows performing short operations in the background separately from the main thread of the program. In short, it allows the child class to run its code in a thread (Android Developers – AsyncTask, 2015).

Method: ReceiveDevicesLocationsTask()

This is the constructor of the class. The RadarView class creates an instance of this class and passes to this constructor a Location object that contains the latitude and longitude of the user's device. The purpose of doing that is because we want to send those coordinates to the "**find_device.php**" Web Service.

Method: onPreExecute()

Before the thread starts running we can perform some actions in this method. We chose to leave it empty.

Method: doInBackground()

This method is executed when the thread starts running in the background. It returns an Integer that will be passed to the *onPostExecute()* when the thread is completed.

The main purpose of this method is to construct a POST request message to send to the server:

1. First, we set the URL of the web service that we will contact.
2. Secondly, we prepare an HTTP connection and we set the HTTP method to "POST".
3. We add an HTTP header to our message, "**Accept: application/json**", in order to tell to the server that we accept data in that specific format, with that specific MIME type.
4. We create the message payload that consists of an ArrayList that contains our device's latitude, longitude, and the detection radius.
5. We send the POST request, expecting a "200 OK" HTTP response code.

Method: onCancelled()

This method is called if the thread is stopped before it finishes its execution.

Method: getQuery()

This method parses the JSON array that is returned by the "**find_device.php**" Web Service.

Method: onPostExecute()

This method is finally called when the thread is completed or stops running and execution returns to the main thread. Depending on the Integer that was returned by the *doInBackground()* method, we throw a suitable error message if needed.

3.2.1.9. Class SendDeviceLocationTask

This class is solely used by the RadarView class. It is responsible for the asynchronous communication with the "**device_position.php**" Web Service. It extends the **AsyncTask** class which allows performing short operations in the background separately from the main thread of the program. In short, it allows the child class to run its code in a thread.

Method: SendDeviceLocationTask()

This is the constructor of the class. The RadarView class creates an instance of this class and passes to this constructor a Location object that contains the latitude and longitude of the user's device. The purpose of doing that is because we want to send those coordinates to the "**device_position.php**" Web Service.

Method: onPreExecute()

Before the thread starts running we can perform some actions in this method. We chose to leave it empty.

Method: doInBackground()

This method is executed when the thread starts running in the background. It returns an Integer that will be passed to the *onPostExecute()* when the thread is completed.

The main purpose of this method is to construct a POST request message to send to the server:

1. First, we set the URL of the web service that we will contact.
2. Secondly, we prepare an HTTP connection and we set the HTTP method to "POST".
3. We add an HTTP header to our message, "**Accept: application/json**", in order to tell to the server that we accept data in that specific format, with that specific MIME type.
4. We create the message payload that consists of an ArrayList that contains our device's latitude, longitude, name, and unique id.
5. We send the POST request, expecting a "200 OK" HTTP response code.

Method: onCancelled ()

This method is called if the thread is stopped before it finishes its execution.

Method: getQuery()

This method parses the JSON array that is returned by the "**device_position.php**" Web Service, which contains either a failure or success message.

Method: onPostExecute()

This method is finally called when the thread is completed or stops running and execution returns to the main thread. Depending on the Integer that was returned by the *doInBackground()* method, we throw a suitable error message if needed.

3.2.1.10. Class UpdaterTaskClass

This class is used solely by the RadarView class to perform periodical updates to its GUI. It is responsible for repeating the communication with the server (ReceiveDevicesLocationsTask) every 10 seconds.

Method: UpdaterTaskClass()

Creates an UIUpdater object, that can be used to perform UIUpdates on a specified time interval. There are two constructors; one that defines the time interval itself and another one that takes it as a parameter.

Method: startUpdates()

This method starts the periodical update routine (mStatusChecker adds the callback to the handler).

Method: stopUpdates()

This method stops the periodical update routine from running, by removing the callback to the handler.

3.2.2. The server

In this section we will analyse the database and the web services. The server consists of three PHP scripts and a MySQL database. One of the scripts named "**mysqli_connect.php**" is included by the other two in order to connect to the database. It contains the database access information; the database **host**, the database **name**, a database **user**, and the user's **password**.

The other two scripts named "**device_position.php**" and "**find_device.php**" are two web services that are contacted by the AMT client.

- The Database -

Table structure for table "devices":

```
CREATE TABLE IF NOT EXISTS `devices` (
  `id` int(10) unsigned NOT NULL,
  `device_id` varchar(64) NOT NULL,
  `lat` varchar(15) NOT NULL,
  `lng` varchar(15) NOT NULL,
  `name` varchar(50) NOT NULL,
  `updated_at` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP
) ENGINE=InnoDB AUTO_INCREMENT=12 DEFAULT CHARSET=utf8;
```

Indexes structure for table "devices":

```
ALTER TABLE `devices`
  ADD PRIMARY KEY (`id`),
```

```
  ADD UNIQUE KEY `device_id` (`device_id`);
--
ALTER TABLE `devices`
MODIFY `id` int(10) unsigned NOT NULL AUTO_INCREMENT,AUTO_INCREMENT=12;
```

id: The primary key of the table “devices”, AUTO_INCREMENT

device_id: A unique id for every device.

lat: latitude

lng: longitude

name: The device’s name.

- The Web Services -

1. Device Position:

The “device_position.php” script receives POST requests that contain the **latitude** of the mobile device, the **longitude**, the **device unique id**, and the **device name**. It checks if all four values were sent in the POST message and then queries the database for the specific device id to determine whether the device already exists in the “devices” table or not. The purpose of this web service is to register new devices in the system.

If there’s already a record of the unique device id (i.e. device_id = device), that means that the device already exists in the table, and so it simply carries out an update of the lat and lng coordinates for the specific device in the table.

If there’s not one, then it adds a new row in the table for a new device and inserts all the values that were sent with the POST message.

Worth noting, in both cases we insert an “updated_at” timestamp for each device to be able to determine when its row was last modified.

Finally, it sends a message in JSON format to the client to let it know whether the operation succeeded or failed.

2. Find Device:

The “find_device.php” script receives POST requests that contain the **latitude** of the mobile device, the **longitude**, and the detection **radius**. The radius value that is sent by the AMT client is the maximum distance from the point of the user’s devices. The web service should return from the database only those devices that are in the specific range.

It does that by using the provided radius and the latitude to calculate **a maximum and minimum latitude**, and by using the provided radius and the longitude to calculate **a maximum and minimum longitude**. Then it only selects those devices in the “devices” table that have a latitude between the minimum and maximum latitude and a longitude between the minimum and maximum longitude. Additionally, it filters those

that haven't been updated for over an hour (i.e. those that have a 1 hour old timestamp) so that they will eventually disappear from the radar.

The purpose of this web service is to return a JSON array of devices that are within the specified range "rad" of the device that sent the POST request.

3.4. UML Class Diagrams

The AMT client class diagram is shown in Appendix A – Figure 3.

3.5. Testing

I uploaded all the PHP scripts to a webserver and I also set up a database and configured the "baseUrl" of the AMT to point to a domain that resolves to the webserver's IP address.

I've tested the server program by connecting to it from a couple of Android smartphones and tablets and I also used a plugin for the Chrome browser called "**Postman – REST Client**" that allowed me to craft and send POST requests to the webserver and add dummy devices for testing. Using the Postman I was able to send a device id, a device name, and latitude and longitude coordinates (found on Google Maps) to the "device_position.php" web service and that way I was able to successfully register new devices. Then, I would use the Postman to send the latitude and longitude coordinates of a device, either a real or a dummy device, as well as a value for my preferred detection radius to the "find_device.php" web service and that way I was able to successfully receive a reply from the server containing an JSON array of "devices in range" that it retrieved from the database. such as mIRC, Icechat and ChatZilla and also from the IRC Client that I personally developed. I tried out various values for the coordinates and the detection radius to ensure that the server program works correctly. The testing not only performed on a single webserver. I also set up five other webservers hosted on different machines (although they were all running distributions of the Linux OS such as CentOS and Debian) at different domains, to verify that the server works properly in various environments.

I've tested the AMT client with six different webservers. I installed the application's package on a couple of smartphones and tablets and tested the AMT with different combinations of connectivity settings. I tested it with only the GPS enabled, only the Wi-Fi, only 3G and the combinations of all these such as GPS + Wi-Fi. The client could work fine in connection with fully-functional servers, and it was able to successfully send POST request messages and receive POST responses from the server.

For the testing I used these devices:

Devices:	1. Sony Xperia S smartphone with Android v4.1.2 (Jelly Bean) 2. Samsung Galaxy S T805 tablet with Android v4.4 (KitKat) 3. Samsung Galaxy S T805 tablet with Android v4.4 (KitKat) 4. LG g2 d802 smartphone with Android v4.4.2 (KitKat)

I changed the setting for the detection radius to lower and higher values in the AMT and successfully received a list that contained only the dummy devices that were in the specified range. For instance, I set the radius to 200 kilometres and then to 100 kilometres and observed that the devices that were further than 100 kilometres away would disappear as the server wouldn't include them in the update messages that it regularly sends to the client. The devices would also appear correctly in the list of detected with their names and distances displayed.

Finally, I changed the locations of the connected smartphones and tablets (i.e. moved them around) to verify that the GUI of the radar would be updated correctly with their new coordinates and successfully track their motion.

4. Evaluation and Conclusions

4.1. Objectives evaluation – Reflection

This was a very hard project for me to complete. Although I am very satisfied with my progress and learning in Java programming for Android, and relieved that I can present an application that works and that is effectively a motion tracker for the Android OS as it was initially planned, I am slightly disappointed by the final look and feel of my work as I failed to implement many of the program features that I came up with during the implementation stage.

The primary reason for that was that I was aiming for a peer-to-peer architecture, rather than classic client-server architecture. My research showed that the Wi-Fi adapter of a device cannot be used in monitor mode to analyse beacon packets and to wirelessly discover peers. In order to put the Wi-Fi Network Interface Card (NIC) into promiscuous monitoring mode, an application requires root access to the system (i.e. administrator access), provided that the Wi-Fi adapter's driver and the adapter itself support monitor mode.

Thus, I had no other option than to reside to a classic Client-Server architecture to achieve the same results. With no other option for peer discovery and peer-to-peer communication, there was a deviation from the original plan. As a result, I did not manage to fulfil the advanced objectives, although the application met at least the core objectives.

I lost valuable time researching about solutions and studying Android programming in order to overcome all the issues that I had in its implementation, exceeding by far the initial estimated time plan for research and implementation.

On the bright side, I'm very pleased that I've managed to implement a functional application for the Android OS and complete my project.

4.2. Future Improvements - Hints

In future versions I was thinking to work more on the radar's accuracy and improve the menus and options, the sidebar and the general look and feel.

One feature that I would like to add is a compass in order to be able to tell whether the other devices are to the North, South, East or West. Another one that I would like to add is the ability to chat with other devices where I could embed code from an IRC client and server that I had coded in the past.

The system could be improved as far as security is concerned, by preventing packet floods and by implementing a mechanism to filter POST messages that don't originate from AMT clients.

5. References

- [Wikipedia – Motion capture, 2015] http://en.wikipedia.org/wiki/Motion_capture , accessed on 29/5/2015
- [Google – Patents, 1998] https://www.google.com/patents/US5744953 , accessed on 29/5/2015
- [Android Developers – Motion Sensors, 2015] http://developer.android.com/guide/topics/sensors/sensors_motion.html , accessed on 29/5/2015
- [Android Developers – Position Sensors, 2015] http://developer.android.com/guide/topics/sensors/sensors_position.html , accessed on 29/5/2015
- [Wikipedia – Sonar, 2015] http://en.wikipedia.org/wiki/Sonar , accessed on 29/5/2015
- [PhysLink, n.d.] http://www.physlink.com/Education/AskExperts/ae456.cfm , accessed on 29/5/2015
- [SlashGear, 2015] http://www.slashgear.com/nasas-new-radar-detects-heartbeats-saving-4-lives-in-nepal-06382451/ , accessed on 29/5/2015
- [OpenSignal, 2013] http://opensignal.com/blog/2013/07/30/40-developer-tips-for-android-optimization/ , accessed on 29/5/2015
- [Vogella, 2015] http://www.vogella.com/tutorials/Android/article.html#androidstudio_starter , accessed on 29/5/2015
- [Brad Wiegmann, n.d.] http://www.bradwiegmann.com/sonar-and-gps/sonar-gps-structure-scan-and-software-mapping-articles/515-cost-of-sonar-and-gps-units.html , accessed on 29/5/2015
- [Android Developers – Bluetooth, 2015] http://developer.android.com/guide/topics/connectivity/bluetooth.html , accessed on 29/5/2015

- [InfoWorld, 2014] http://www.infoworld.com/article/2608498/mobile-apps/what-you-need-to-know-about-using-bluetooth-beacons.html , accessed on 29/5/2015

- [Security Laboratory, 2007] http://www.sans.edu/research/security-laboratory/article/bluetooth , accessed on 29/5/2015

- [CNET, 2012] http://www.cnet.com/news/the-power-of-bluetooth-4-0-itll-change-your-life/ , accessed on 29/5/2015

- [Wikipedia – Bluetooth, 2015] http://en.wikipedia.org/wiki/Bluetooth , accessed on 29/5/2015

- [Android Developers – Creating P2P Connections with Wi-Fi, 2015] http://developer.android.com/training/connect-devices-wirelessly/wifi-direct.html , accessed on 29/5/2015

- [Android Developers – Wi-Fi Peer-to-Peer, 2015] http://developer.android.com/guide/topics/connectivity/wifip2p.html , accessed on 29/5/2015

- [Android Developers – Dashboards, 2015] https://developer.android.com/about/dashboards/index.html , accessed on 29/5/2015

- [Wi-Fi Alliance, 2015] http://www.wi-fi.org/product-finder-results?sort_by=default&sort_order=desc&categories=4&capabilities=3&certifications=34 , accessed on 29/5/2015

- [Stack Overflow, 2011] http://stackoverflow.com/questions/2334244/is-there-any-way-to-put-android-wifi-droid-handset-into-promiscuous-monitoring , accessed on 29/5/2015

- [Stack Overflow, 2012] http://stackoverflow.com/questions/11251610/wifi-direct-android-4-0-with-multiple-3-devices , accessed on 29/5/2015

- [Wi-Fi Alliance – How many devices can connect?, 2015] http://www.wi-fi.org/knowledge-center/faq/how-many-devices-can-connect , accessed on 29/5/2015

- [Wikipedia – Stateless protocol, 2015] http://en.wikipedia.org/wiki/Stateless_protocol , accessed on 29/5/2015

- [Google Code, 2008] https://code.google.com/p/apps-for-android/source/browse/Radar , accessed on 29/5/2015
- [Stack Oveflow – Embedding a Radar from Google, 2012] http://stackoverflow.com/questions/10568994/embedding-a-radar-from-google-in-app , accessed on 29/5/2015
- [Android Developers – App Manifest, 2015] http://developer.android.com/guide/topics/manifest/manifest-intro.html , accessed on 29/5/2015
- [Android Developers – Manifest.permission, 2015] http://developer.android.com/reference/android/Manifest.permission.html , accessed on 29/5/2015
- [Android Developers – Activity, 2015] http://developer.android.com/reference/android/app/Activity.html , accessed on 29/5/2015
- [Android Developers – Starting an Activity, 2015] http://developer.android.com/training/basics/activity-lifecycle/starting.html , accessed on 29/5/2015
- [Stack Overflow, 2013] http://stackoverflow.com/questions/14671897/super-oncreatesavedinstancestate , accessed on 29/5/2015
- [Android Developers – SensorManager, 2015] http://developer.android.com/reference/android/hardware/SensorManager.html , accessed on 29/5/2015
- [Android Developers – Sensor, 2015] http://developer.android.com/reference/android/hardware/Sensor.html , accessed on 29/5/2015
- [Stack Overflow – ConnectivityManager to verify internet connection, 2011] http://stackoverflow.com/questions/7071578/connectivitymanager-to-verify-internet-connection , accessed on 29/5/2015
- [Movable Type Scripts, 2015] http://www.movable-type.co.uk/scripts/latlong.html , accessed on 29/5/2015
- [Android Developers – Cursor, 2015] http://developer.android.com/reference/android/database/Cursor.html , accessed on 29/5/2015

- [GitHub – Create Pseudo Unique ID for Android Device, 2014] https://gist.github.com/pedja1/fe69e8a80ed505500caa , accessed on 29/5/2015

- [Android Developers – AsyncTask, 2015] http://developer.android.com/reference/android/os/AsyncTask.html , accessed on 29/5/2015

- [Movable Type Scripts – Selecting points from a database by latitude/longitude within a bounding circle, 2015] http://www.movable-type.co.uk/scripts/latlong-db.html , accessed on 29/5/2015

6. Bibliography

1. Murphy, M. (2015) “The Busy Coder's Guide to Android Development”

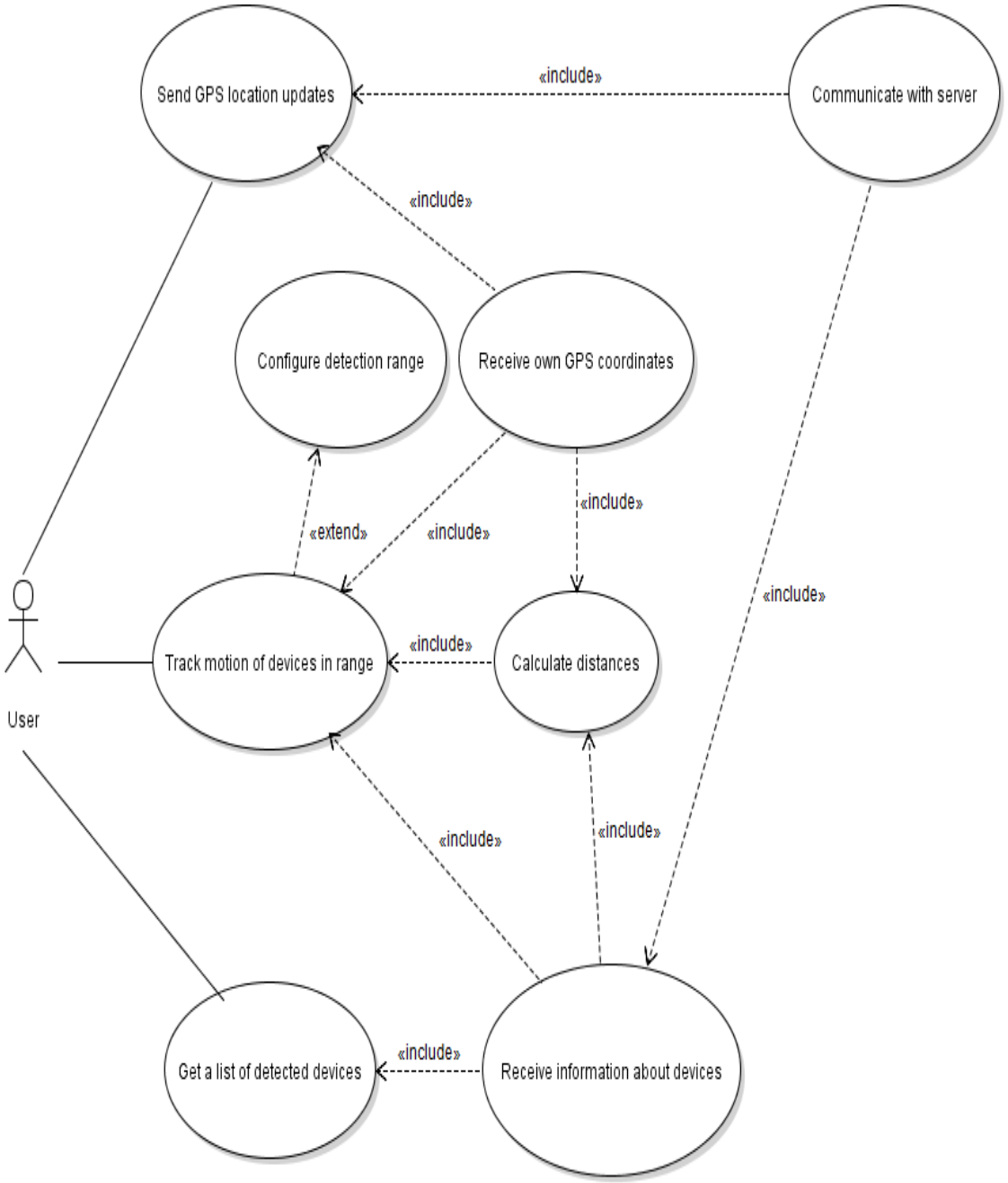
2. Liskov, B. & Guttag J. (2000) “Program Development in Java: Abstraction, Specification, and Object-Oriented Design”

3. Panayiotopoulos, J. (2003) “Internet Programming with Java”

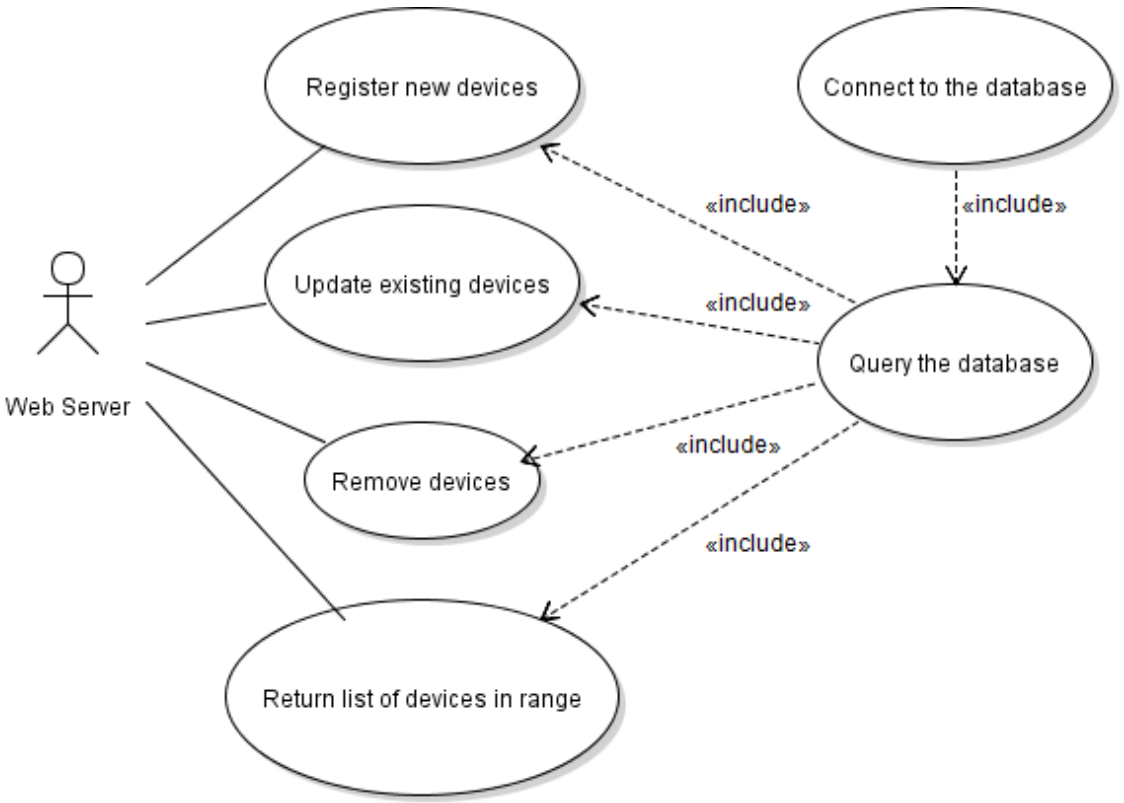
7. Appendices

All diagrams were created using the free “Violet UML Editor”.

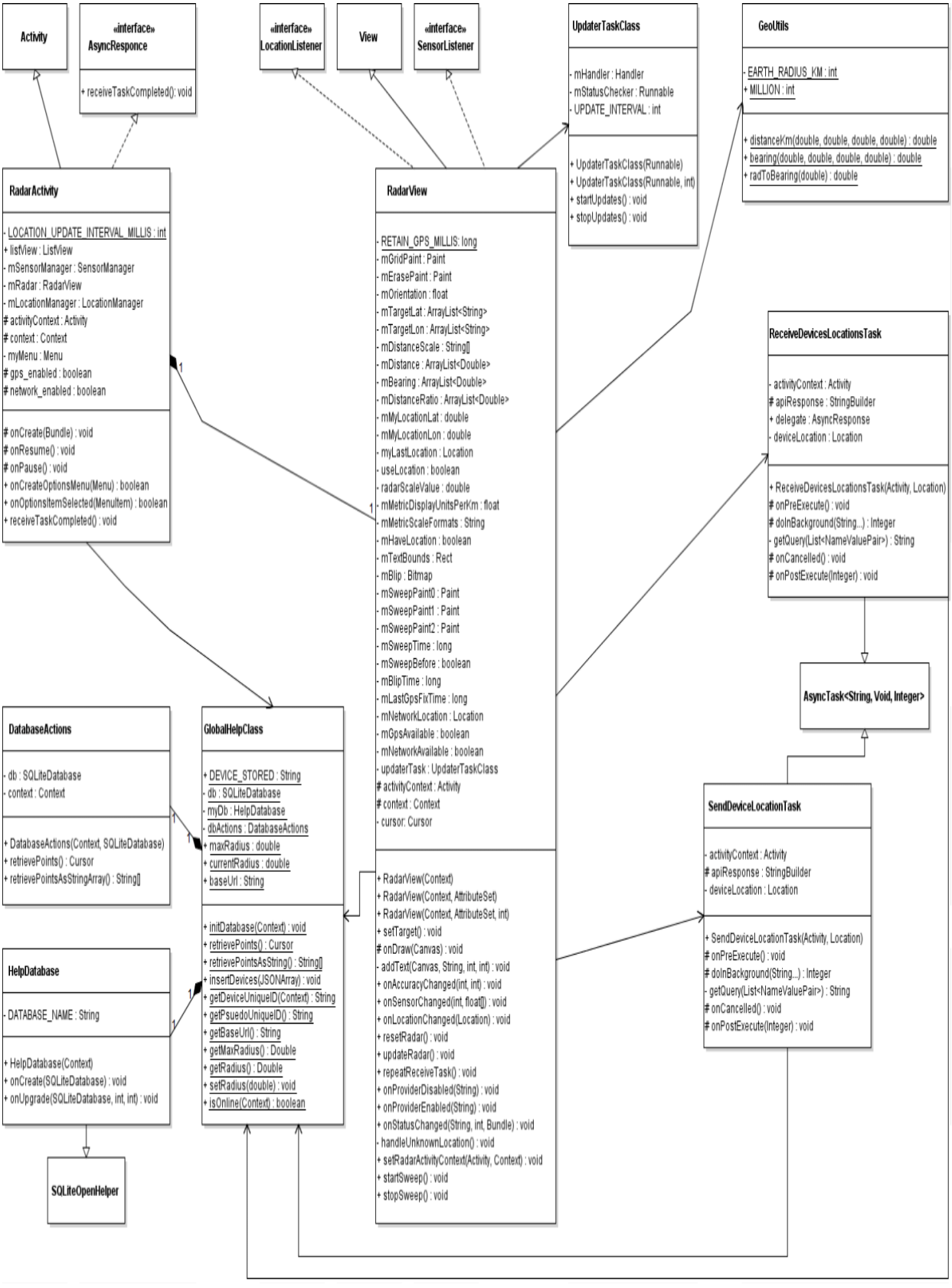
Appendix A. UML Diagrams



Appendix A - Figure 1: AMT use case diagram

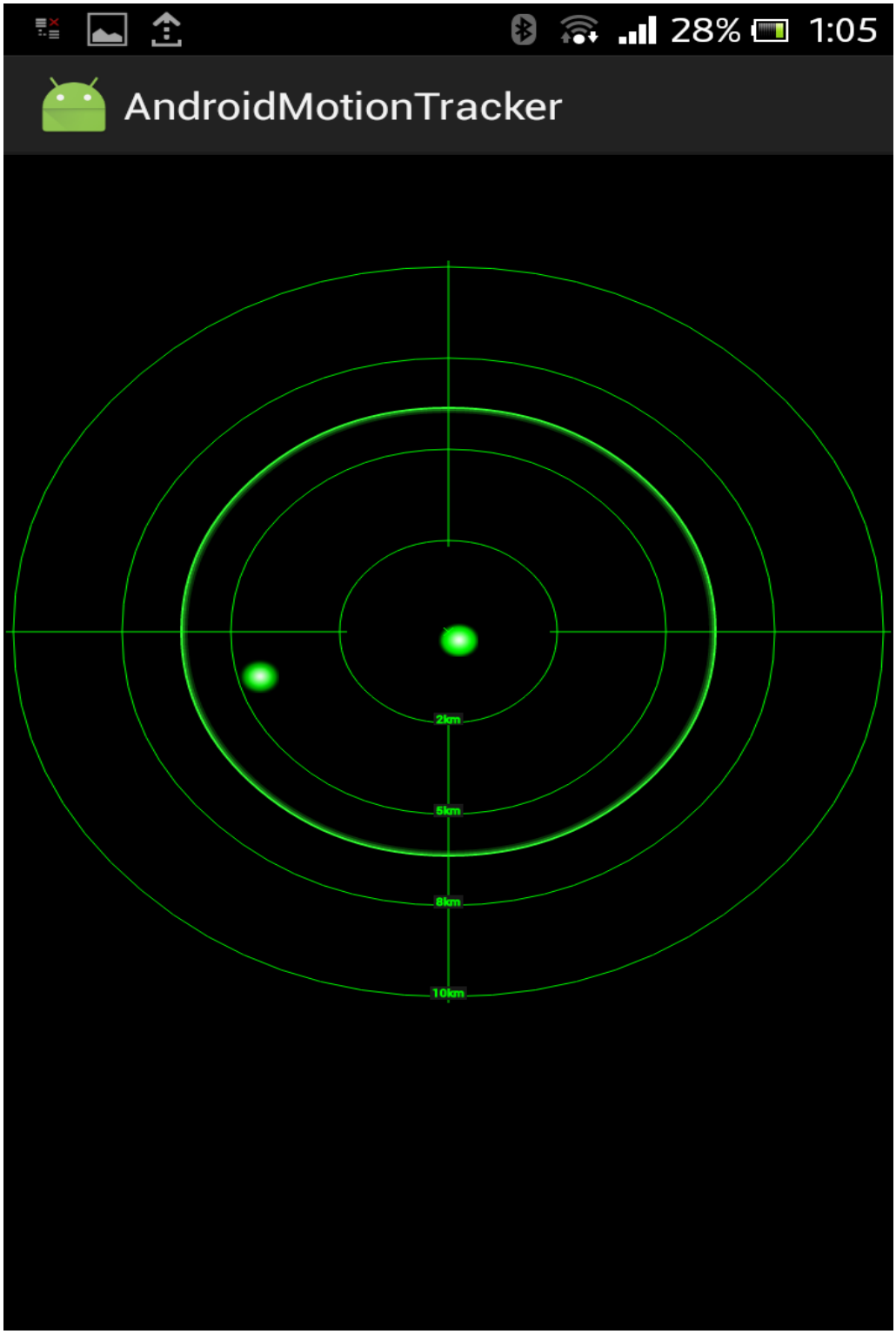


Appendix A - Figure 2: Server use case diagram

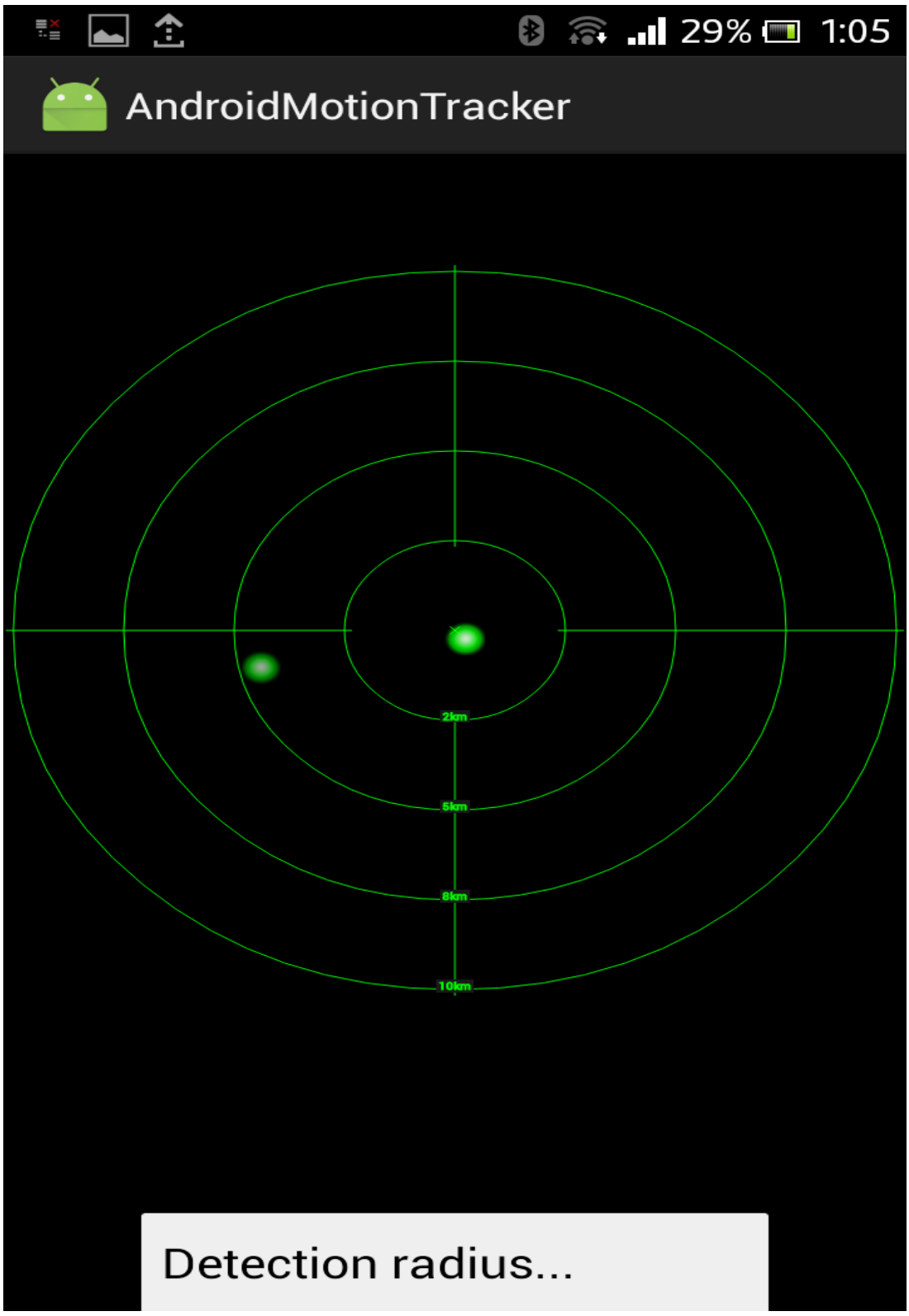


Appendix A - Figure 3: Class diagram

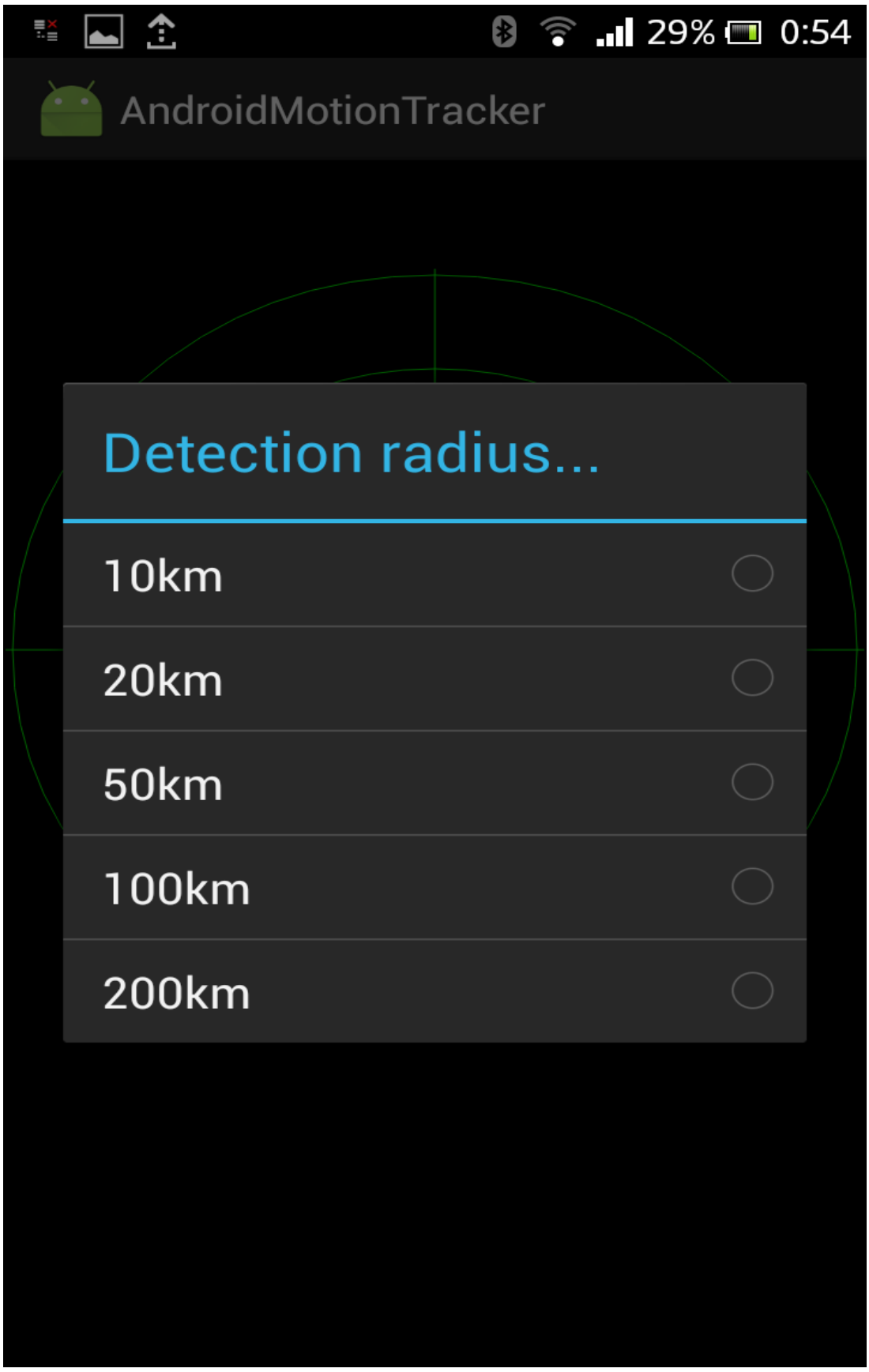
Appendix B. Front-end screen shots



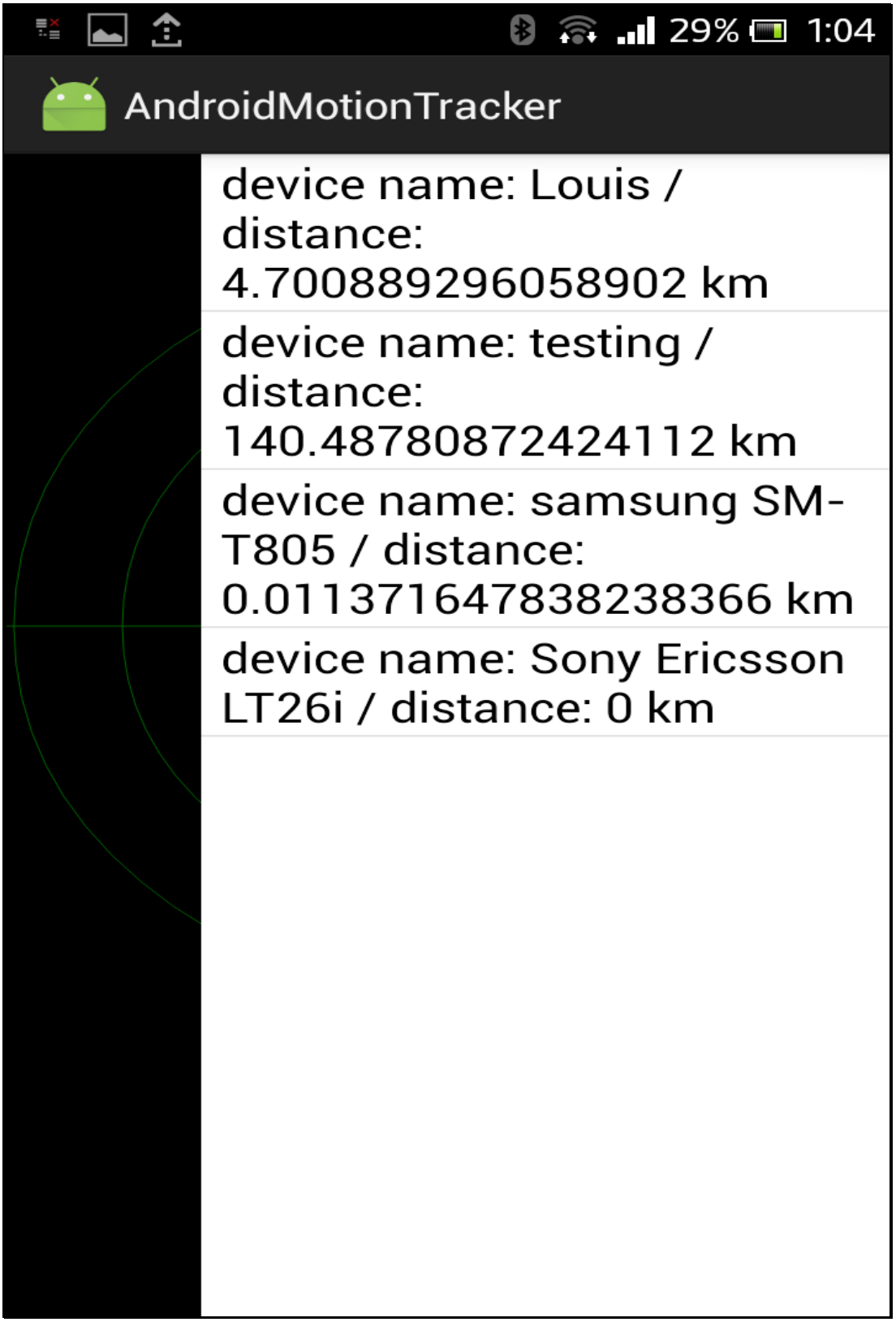
Appendix B - Figure 1: “Radar” Screen



Appendix B - Figure 2: “Detection radius menu” Screen



Appendix B - Figure 3: “Detection radius configuration dialog” Screen



Appendix B - Figure 4: “Detected devices list” Screen

Appendix C. Installation and setup

C.1. CD contents

Into the CD the following folders are located:

- Appendices Folder: contains the UML diagrams, screenshots and storyboard
- Application Folder: contains the Android application package in .apk format
- Source Code Folder: contains all the project source code
- FPR_GeromitsosLoudovikos.doc : the present document

C.2. Using the System

1. The PHP scripts have to be uploaded to a webserver and it is essential to properly configure the **baseUrl** attribute in **GlobalHelpClass** to point to the correct domain and directory where the scripts were uploaded. For example:

   baseUrl = "**http://mydomain.com/radar/**";

2. A database has to be configured in **mysqli_connect.php** and have **devices.sql** imported via phpMyAdmin.

3. Users must enable the "unknown sources" or "mock locations" setting at their device's settings, which can be found in the "Developer options". Having done that, they can proceed to install the **amt.apk** and run the application.