John Knox

RaceTrace:
A real-time group tracking app

Computer Science Tripos, Part II

Churchill College

April 23, 2013

Proforma

Name:	**John Knox**
College:	**Churchill College**
Project Title:	**RaceTrace: A real-time group tracking app**
Examination:	**Computer Science Tripos, Part II (2013)**
Word Count:	**To do**[1]
Project Originator:	Dr John Fawcett
Supervisor:	Dr John Fawcett

Original Aims of the Project

To create an android application that allows multiple participants of some track based activity to continually track each other's progress. The display would be accurate enough to provide useful location information in such a situation. This translates to having accuracy and latency such that at all times the locations displayed must be either less than 5 seconds late or less than 10 metres from the true location. The communication was to be implemented in more than one way, so that an investigation could, and would be done into the relative effectiveness of at least two different protocols.

Work Completed

I have created an Android application that achieves this by first letting users connect to each other in one of two different ways, and then displaying a diagram which shows the relative positions and trails of the other players on the screen, along with that of the user. This diagram is continually updated with

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

the latest known locations of the other users. The inter-device communication has been implemented in two distinct ways, the first using a client-server model, and the second being peer-to-peer with various further protocol options for each model. The networking technologies used are Wi-Fi and operator provided mobile internet. I have also conducted a quantitative evaluation of the protocols implemented, and considered the security issues involved. The work completed fulfils the project requirements.

Special Difficulties

To do.

Declaration

I, John Knox of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

Acknowledgements

I would like to thank Dr John Fawcett for providing the original idea and his valuable guidance throughout the year.

Chapter 1

Introduction

1.1 Context and Motivation

With the sudden rise of smart phones in the past decade, new innovations are being discovered at a rapid pace. One niche within the applications market is the area of physical sports enhancement. There are now multiple widely used applications[1] on the market offering real-time as well as historic personal tracking and analysis for the duration of some physical activity, be it cycling, running, skiing or something else altogether. Typically these applications display the location of the user on screen as they advance around some course, and provide statistics along the way such as velocity, distance travelled and many others. They also allow the user to store records of their session, so that future performance can be compared and further analysis done. However, despite the many features on offer, none of these solutions acknowledge the fact that often the activities involved are group events, thus failing to capitalise on an entire dimension of available data. Namely, the performance of all of the other participants with whom the user may be competing with[2].

In the video game industry it has long been standard practice for racing games to display a simple map of the course on the screen (typically in one of the corners) showing the location of each competitor. This allows the user to continually evaluate their relative position, and stay informed throughout the duration of the race. While there are arguments for and against such a feature[3],

[1] Two popular examples at the time of writing are Endomondo and Sports Tracker

[2] The social networking "share" paradigm has been applied to such applications, allowing others to see user statistics after the event has taken place but, to my knowledge, there is no real-time sharing facility allowing others to compete at the same time

[3] While knowing the location of other participants can increase competitiveness and therefore improve performance, it can also result in degraded performance due to winners complacency

its unanimous inclusion in gaming is sufficient proof that it is, at the very least, sometimes desirable.

1.2 Initial Direction

Taking a step back for a moment and considering the alternatives to developing a smart phone application, there are a variety of ways such a system could be implemented. One approach to solving this problem would be to develop an application specific device for the job. If this approach was taken then a multitude of options would open up as to what hardware should be used. The networking aspect of the device would perhaps be the most interesting with the large variance between available options.

One reasonable option would be to use a private radio system. In a typical use case of the desired system, the distance between participants would not exceed the range of available portable license-free radio modules. This means the user could support their own network infrastructure without relying on external providers or being subject to external costs.

While designing a custom device would have some definite advantages, with the widespread adoption of smartphones - most of which have all of the necessary hardware for this task, there is really no need. In order to achieve widespread adoption, in comparison to the alternatives, the transaction cost should be minimised, so requiring additional hardware is an unnecessary disadvantage.

On top of that, current top end mobile phones nowadays have capabilities by far exceeding the requirements of such a project, making the design process easier and allowing for more complicated features to be added to the basic idea. For this reason I have opted to design a smartphone application. At the time of the decision, there were three main mobile application platforms, namely Apple iOS, Google Android, and Microsoft Windows Phone 7. The resources offered by each platform do not vary much, so it was a somewhat arbitrary decision of which to choose. The Android platform was chosen out of familiarity with the Java platform as well as having multiple devices at my disposal.

1.3 Scope of the Work

In order to achieve the goals laid out at the start there are four main features that are required. These are outlined below.

---

and losers frustration

1.3.1 User Interface

The app requires a main graphical interface for displaying the position of the other participants in relation to the user. It was not my intention to produce a production grade user interface though I recognise that this would be a useful addition to the project. The only requirement was that it be effective and reliable, this has been achieved.

1.3.2 Session Setup

In order for inter-device communication, there needs to be a method of initial synchronisation so that each device knows where to send data to. There are various possible approaches to this problem. I have implemented two alternate methods, each with their own merits.

1.3.3 Location Sensing

Clearly some method of determining the current location is required if this information is to be shared with other devices. This was achieved using a combination of data from the Global Positioning System (GPS) and the mobile network.

1.3.4 Inter-Device Communication

The way devices communicate with each other is crucial to the project. I have used the internet API provided by the Android operating system to do all main communication.

Chapter 2

Preparation

In this chapter I outline the theory behind the problem, and the work conducted before the start of the development stage.

2.1 Resources Available

In this section I shall go through the relevant capabilities of the devices at which I am aiming my application.

2.1.1 Networking

The ability to communicate between devices is vital to this project. Fortunately, there are multiple methods of communication on offer to mobile developers. Practically all mobile phones are now connected to the internet, many with high speed access thanks to 3G and 4G technologies, as well as having 802.11 WLAN and Bluetooth. Communication can be split into ad-hoc and infrastructure methods. Bluetooth falls under the Ad-hoc category, where there is no external networking infrastructure, along with Wi-Fi direct, a protocol available to the latest Android devices allowing Wi-Fi to be used in Ad-hoc mode without the need for an access point.[1] Wi-Fi is more commonly used in infrastructure mode, as a portal to the wider internet, as well as the mobile internet services such as 3G. The Android OS also allows devices to share an internet connection via Wi-Fi or Bluetooth.[2]

With the android devices available at the time, I had no way to use Wi-Fi direct since this is exclusive to higher versions of Android. The network data connection was the chosen method of communications, since it provides uniform access between pairs of devices whenever they have signal, as opposed to some ad-hoc method such as Bluetooth where opportunistic transfer would have to be relied on and application performance would vary widely depending on the

differing distributions of participants. For example chains of players where each is within range of the next would allow for a fully connected network, whereas any other arrangement would not, resulting in partial data showing on screen until the networks reconnect. In the worst case if no players were within ad-hoc range of any other then no data would be transferred rendering the application useless.

Adding ad-hoc communication as an additional feature would be advantageous, this is elaborated in Section 5.2, but developing an opportunistic network system would considerably increase the scope of the project so it has not been explored further.

2.1.2 Processor

The processing power of available Android devices varies widely, from top-end models having quad-core processors and dedicated graphics, to budget devices performing several orders of magnitude worse on CPU benchmarks.[3] Even the low end devices are sufficient for the computation required for this task. The requirements are:

- Send and receive multiple network packets per second
- Encrypt and decrypt multiple packets per second using some encryption scheme
- Scan through arrays spanning hundreds of data points, drawing each to the screen without delaying the user

2.1.3 Memory

In order to function properly, each device must record a trace of its history and that of the other devices. This has the potential to be a large amount of data so the requirement was estimated before proceeding with the project. A typical use scenario would not normally exceed 4 hours. Say each data point is represented by 12 bytes, 3 integers which hold an x-coordinate, a y-coordinate, and a time value. At one update per second per device the required memory to store the history would be:

$$\begin{aligned} size &= 1 \times 60 \times 60 \times 4 \times 12B \\ &= 172,800B \end{aligned} \tag{2.1}$$

That is, 169KB for each device. Clearly the requirement will scale linearly with the number of devices.

All Android devices provide each application with at least 16MB of allocatable memory. Bearing this in mind, the limit on number of users will lie somewhere between 50 and 100, far more than the number in a typical use case.

2.2 Networking

All popular mobile providers provide some wireless internet service, usually accessible anywhere where the conventional mobile telephone network is available. This allows for continuous connection while on the go, and allows applications to interact with servers located anywhere in the world.

2.2.1 Reliability of Mobile Networks

The portability of mobile phones brings some disadvantages along with it. One being that signal strength varies widely across land, depending on proximity of base stations as well as the characteristics of the surrounding landscape.

Typical uses of sports tracking applications include mountain biking and skiing, activities such as these raise concerns due to the nature of their location, often being within dense areas of forest and in remote areas that wouldn't usually be prioritised for by network providers. For these reasons, signal strength in these locations is typically weak, often cutting out entirely and though significant effort has been made to avoid it packet loss is still a problem.

This means an application relying on data coverage for communications would need to have some resilience to disconnection, and should use the available network throughput efficiently so that important data is prioritised.

Differing Characteristics

During some preparatory tests it was found that the characteristics of the two available 3G networks differed somewhat. While both the Three and Vodafone networks had similar bandwidth and latency, probing the networks showed that packets quickly queue up on the Vodafone network, perhaps as a penalty, whenever many packets are sent in quick succession. This results in effectively huge reception delays (up to 2 minutes). The way to avoid this is to bucket the data into larger packets to reduce the routing load on the network. On the Three network, no such throttling was apparent.

This showed that whatever communication implemented should try to minimise the unnecessary work for the network, and perhaps even have use current statistics to infer such network characteristics and behave accordingly.

2.2.2 Network Protocols

Client-Server vs Peer-to-Peer Systems

Given that the phone data connection is to be used as the communications "port", there are two ways the system could be realised.

One approach uses the support of servers in the cloud to serve client systems and act as intermediaries. The other approach is to use only client systems and no external processing other than the network infrastructure.

The main advantage to the peer-to-peer approach is scalability. If the application was to be widely used then a peer-to-peer system would be self sufficient and allow unbridled scale.

The client-server system on the other hand would require servers to be maintained either at the cost of some party. If the application package provided this service then more resources would have to be provided as the number of users increases. As well as this it would be a single point of failure whereas true peer-to-peer systems are only reliant on their peers.

There are advantages to the client-server model however, for one it avoids the NAT traversal problem outlined below in Section 2.2.3.

Another slightly more subtle advantage of using a server is an improvement in transmission success rate. This may be considered anti-intuitive since an extra hop is required for each packet, but the reason is explained below.

In the event of high packet loss (considered likely given the application), packets can be lost when transmitted from the phone to the base station and vice-versa. For peer-to-peer and client-server systems alike, the chance of a successful transmission with one attempt at sending is:

$$P_{success} = P \times P \tag{2.2}$$

However when repeat attempts are allowed from the intermediate server, the probability of success lies in favour of the client-server system:

$$P_{success} = P \times (P + \gamma) \tag{2.3}$$

where

$$\gamma = (1 - P)P + (1 - P)^2P + (1 - P)^3P + ... \tag{2.4}$$

The reason for the increase in probability of success, is that packets reaching the server but not the end device are not lost, since the server can retry from the "halfway" point. Hence even when repeat transmissions are enabled from

the initial sender (in this case the Android device), the client-server model will always have a probabilistic advantage where there is packet loss[1].

2.2.3 Network Address Translation

With most internet services still using IPv4, addresses now come at a premium with many ISPs using Network Address Translation (NAT) to expand their internal network address space. With the transient nature of mobile phone data connections, most if not all mobile ISPs fall into this category. One problem with this technology is that it can make peer-to-peer connections difficult or even impossible.

NAT occurs at the router between an internal network and some wider outside network. This router allocates addresses to all of the nodes in the internal network, but has it's own external network address. When an internal node sends a packet to some node in the external network, the router translates the external address to the corresponding internal address based on the flow port number and its current state.

The result is that many internal nodes can operate using a single external IP address, sidestepping the problem of exhausting the address space.

Problems arise when you try to connect to a node behind a NAT. If this is the first point of contact, then the router has no relevant state to determine which internal node to forward to, rendering it un-connectable unless it initiates the connection itself. With both clients behind different NATs, then none of them can connect to the other. UDP hole punching is a technique used to overcome this by getting both clients to initiate the flow, using the same port numbers. The NAT routers each see the outgoing packets and then set up the state so that the incoming packets will be directed to the right node.

Tests were conducted to find out whether peer-to-peer communication would be possible over the mobile data networks. During these tests, I had access to two 3G data networks, Three and Vodafone. Using UDP hole punching I was able to connect to an Android device over the 3G network provided by Three, which would ordinarily provide the means to achieve peer-to-peer communication. However in the case of Three, there are further complications meaning that despite packets being successfully sent from external networks, such as the Internet connection provided by Cambridge University, sending packets from within the Three network to other addresses in the Three network could not be done. From this I concluded that the network provider doesn't allow intra-network commu-

[1] Assuming that packet loss between the base station and server is negligible compared to loss between the client and base station.

nication[2] meaning true peer-to-peer protocols are not possible when limited to this network provider. Since external connections were successfully achieved, It is assumed that peer-to-peer communications would be possible when a mix of network providers is used.

Unlike Three, in the experiments conducted Vodafone always remapped the port numbers for outgoing messages, meaning the UDP hole punching method of NAT traversal would not work, disallowing peer-to-peer traffic between Three and Vodafone. In light of these discoveries, it was clear that a peer-to-peer system would not be universally accessible, but would work in theory when certain networks are used, for example between two networks operating using the system that Three uses.

The transition from IPv4 to IPv6 will massively increase the internet address space[4], and so should eliminate the need for NAT, making peer-to-peer mobile applications more practical and universally accessible.

For the above reasons, doing an implementation of the system using a peer-to-peer protocols would still have benefits, so I continued with the plan to make both a client-server implementation and a peer-to-peer one.

2.3 Location Data

2.3.1 Obtaining a Location

Most Android phones are able to use both GPS satellites and the mobile phone network to triangulate their location. If connected to Wi-Fi this can also be used to fetch location data. The android OS can provide location estimates from both sources when available, along with an estimate of their current accuracy. This allows applications to combine the different location providers and use the location data of the one that is most accurate at the time providing redundancy meaning long intervals without a location update are rare. The code snippet to combine the two location providers was taken from the Android APIs with minimal adjustment. The power consumption of the device increases as more location providers are used, with GPS being the most power hungry. For this reason, update intervals can be specified so that the providers will only update their location value at specific time intervals. Clearly the interval size depends on the application, with this particular application, aiming for real time updates and bearing in mind the competitive purpose of the application, relatively frequent

[2]Perhaps the reason is to stop attacks that originate within the network. This wasn't pursued further.

updates are desired.[7] In all tests conducted, the frequency of updates used was 1 second though this value is contained in the configuration file and can easily be altered in the application settings. It was observed that on the test devices used, battery consumption did not suffer severely even with this frequent update period.

2.3.2 Displaying Locations

When drawing the trail of a user to the screen, it is important for the image to appear in agreement with the path the user has experienced. Since we, as humans, don't notice the curvature of the earth in everyday activities, we perceive it as flat. This means for example when we walk the perimeter of what we think of as a square on the surface of the earth, we expect to see the drawn path appear as a square, when the path in reality is a 3 dimensional path.

The data provided by each of the location providers comes in the form of latitude, longitude, altitude, speed, and time values. Because the shape of the earth is closer to that of an ellipsoid than a sphere, simply plotting the latitude and longitude as y and x coordinates on the screen would cause the image to appear skewed. To account for the uneven form of the planet, we must apply a coordinate transform to map the data onto a geographic coordinate system. There are four main coordinate systems in use, latitude and longitude (LL), Universal Transverse Mercator (UTM), Universal Polar Stereographic (UPS), and Ordinance Survey Great Britain (OSGB). UPS is a system designed to cover the polar regions, which are not covered by UTM. The option of proceeding with UPS was discarded because for the purposes of this project, operating in the polar regions is not a concern though this decision may have to be revisited if further development takes place. OSGB covers only the area in and around Great Britain, immediately making it less desirable, but again for the purposes of this project, it would suffice, leaving UTM and OSGB as possible choices.

Both transforms are based on an ellipsoidal model of the earth, though UTM uses the more universal WGS84 model. Being a transverse Mercator projection, UTM has the property of being conformal, meaning that it preserves angles but distorts distance and area (over large distances across the earth), whereas OSGB is a Mercator projection meaning it is not conformal and does preserve distance and area but distorts angles.[5]

In order to tranform coordinates from LL to one of these systems, there is not just a single map transformation to apply. For example, in UTM the surface of the planet is divided into 60 geographical zones based on longitude and a different transverse mercator projection is used for each zone. This means calculation isn't

trivial, but fortunately there are various open source libraries available that offer conversion between the systems. Of these I chose JCoord because out of those available in Java with suitable licensing, it was the most lightweight and provides both OSGB and UTM coordinate systems.[6]

To decide which projection, if any would be needed, I used the location plotting function of the application (which was developed at the beginning of the project and is explained in Section 3.5.2) to trace my path when walking in certain shapes. Walking along the perimeter of an football pitch I was able to display the shape when plotted in the different coordinate systems to compare the shape and detect any severe distance scaling using the relative sizes of the boxes as well as any angle distortion from the well defined right angled corners of the pitch. Using a sports pitch also has the advantage that the surface is level and flat. When drawing the raw longitude and latitude as x and y coordinates, there was clear distortion in the angles of the trace displayed, causing the pitch to resemble a parallelogram rather than a rectangle. This deformity was also observed, to a slightly lesser extend when using the OSGB transform. However, when the UTM transform was used, there was a large improvement in the shape of the trace displayed. Upon careful inspection of the UTM trace the edges were found to be not perfectly perpendicular, but this error was not easily noticeable and certainly good enough for the desired application. There were also no observed differences between the areas of the pitch boxes.

With these results, I chose to use the UTM coordinate system as it displayed a much more intuitive representation of the path in the location where it was tested[3] than the other transforms.

[3]Churchill College Grounds, Cambridge

Chapter 3

Implementation

3.1 Software Structure

The client software is composed of the 8 packages briefly described below:

graphics Holds the classes responsible for all graphical manipulation other than the simple menus provided by the Android SDK.

gui Holds the Activity classes responsible for the Graphical User Interface.

location Contains the code for obtaining location data.

network Contains all code that does any network communication.

protocol Holds the ProtocolManager class and its subclasses which implement different protocols. Also holds other related classes.

session Holds all code directly relating to the current user session. This includes the Device, Session and Session construction classes.

settings Holds the configuration file for global storage of numerical values and options, as well as any miscellaneous configuration Enums.

store Holds all code for storage of location data and functionality for updating this and providing updates to other components.

A trimmed down dataflow diagram of the software is presented in Figure 3.1. The Setup Activity guides the user through the session setup process, interacting with the Session Setup class which does the work. This yields a Session object which is given to the Protocol Manager class and used to initialise the protocol.

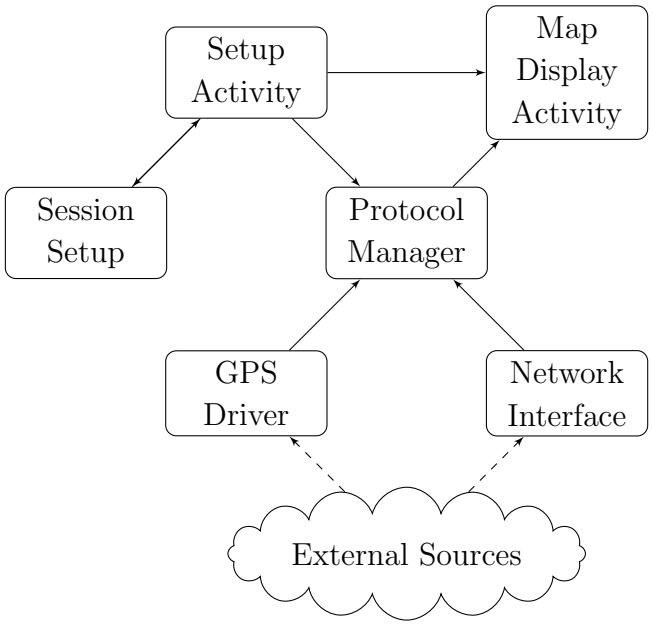


Figure 3.1: Diagram showing the flow of data between core system modules

At the same time, control is passed from the Setup activity to the Map Display Activity, which gets asynchronously updated as a result of the work of the Protocol Manager, which is to coordinate the incoming and outgoing data.

A Configuration class was used to store all of the numeric values and enumerations within the system - at all points in the project where a numeric value or Enum was required, this class is referenced. This makes the software entirely tune-able, with a single point at which changes need to be made.

3.2 Software Engineering approach

An iterative development process was used for the main construction of the application which is described below.

In Android development, decoupling the UI from the back end computation is simple, since each "screen" has its own Activity class which can interact with the system components through their public interfaces. This allowed the "bare-bones" structure of the GUI to be built without yet having any "behind the scenes" code.

With the bare GUI in place, the core framework of the system was built. This was a collection of classes that would hold the session state throughout the use of the application, and which the public methods of the various isolated components of the system would be called from. The core includes Classes representing De-

vices, Sessions, Protocols etc. Gaps with strictly defined interfaces that would be implemented by additional modules were left at this stage. This was aided largely by the following of Object-Oriented Programming (OOP) principles which were used throughout the entire project. Information Hiding is the process of defining standard interfaces for components and hiding the internal implementation so that internal changes may be made without affecting other components, this was invaluable when multiple alternative modules were implemented, such as the different session setup procedures.

Much care was taken at the start of the project to define the classes and structure of the software before construction began, so that each package and component would have a specific purpose (tight cohesion), and that they would be loosely coupled, meaning the points of interactions between them were minimal and well specified. To do this a UML class diagram was drawn up to provide a reference point for the interfaces and overall structure of the application. This was very helpful during the implementation stage since all structural and interface decisions were documented here as they were made, and the diagram was consulted often as the project advanced.

Iterative processes were used for the development of the networking modules as well, this is elaborated in 3.6.

With the core structure in place, the next step was to iteratively add components to the system until it was fully functional, and then to continue iterating, adding further improvements.

3.3 User Interface

Although this project was not concerned with the usability of the user interface, clearly some graphical interface was necessary, and to best represent the capabilities of the system, a reasonable attempt should be made. With this in mind a cognitive walkthrough was done to identify the aims of the users when using the app and the common decisions that would have to be made. The “menu screens” were wireframed by hand to quickly prototype an easy to use and intuitive design.

In the android ecosystem, the screens of the user interface are represented by Activities. Activity is a subclass of the Java Object class, allowing arbitrary functionality on top of the standard Android framework. Ordinarily each Activity corresponds to a view of the screen and has its own layout, which is usually specified in an xml file.

An instance of an activity may start another activity with the use of an Intent object, this instantiation can be given an arbitrary parameter so that the

original Activity can distinguish it from any other Activities it may have created. When the default behaviour is used, stopping a newer activity will result in it's creator being resumed. This naturally produces a "go back" effect and simplifies the process of creating a tree-like menu screen structure. When this occurs a return value, along with the arbitrary distinguishing parameter is provided to the original Activity by wall of a callback `onActivityResult()` method.

The menu structure of the application is illustrated in Figure 3.2. Each of the three activities accessible from the main menu is a subclass of SessionSetupActivity and can be used to initialise the a session. When a session ends and players exit the Map Display Screen, on return to one of the session setup activities, the `onActivityResult` method is overridden to immediately call `onBackPressed()`, forcing the user to return to the main menu.

As well as those in the diagram, there is a Preferences activity, which allows the user to modify some of the options, such as the protocol used, and their name to identify themselves on screen. There is also a Single User option.

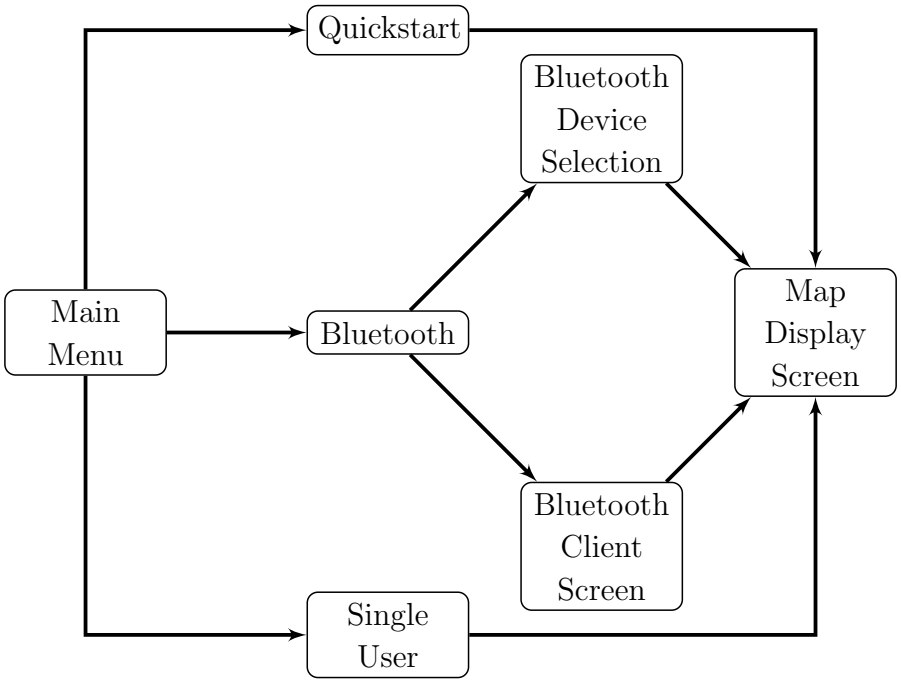


Figure 3.2: Flowchart showing the possible paths through the activities of the application

The Android Software Development Kit (SDK) offers assistance in creating graphical user interfaces, including a "What you see is what you get" (WYSIWYG) graphical editor. However, for the purposes of this project (simple menu and selection screens) this tool was unnecessary and it was decided that all GUI

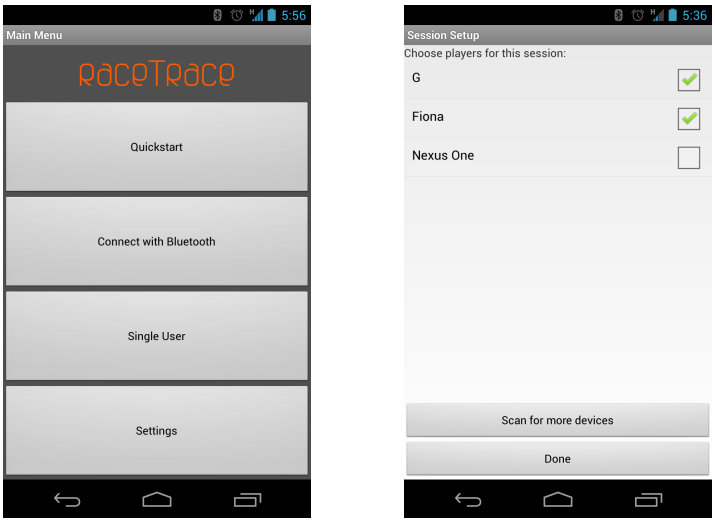


Figure 3.3: Screen shots of the Main Menu and Bluetooth User Selection screens

work would be done precisely by text entry and using relative positioning so no artefacts would be accidentally introduced by the automated tool. Such artefacts could be hidden during tests depending on screen size and resolution and lead to troublesome bugs. Despite this the tool was useful for visualising the changes made to the .xml layout files without having to re-install the application and test it each time.

3.4 Implementing the Session Setup function

3.4.1 Session Representation

As far as this project is concerned, a *session* is the recreational activity taking place from when the participants start to move until they all cease to move and the activity is considered finished.

A Session object, in this case, is a digital representation of such a *session*. It is implemented as a concrete Java class whose objects have both an ordered list of Device objects and a set of cryptographic keys for communication if encryption is used. The Session class follows the Singleton design pattern, ensuring that no more than one instance of it exists at any one time.

A Device object consists of a name, a DeviceHistory object which is where the historic location data is kept, a DeviceHandle which is an instantiation of a subclass of the Absract class DeviceHandle containing the address information for this Device, and a DevicePath which stores the drawable objects that are constructed by the MapDrawer class (revisited in detail in Section 3.5). In this

project, all communication was done via the internet, so DeviceHandleIP is the only implemented subclass of DeviceHandle, other than DeviceHandleSingleUser which doesn't store any information and is only used when there are no other participants.

3.4.2 Session Setup

The application structure first implemented left a gap for a session setup module. The reason for this is that there are numerous ways to setup the session and it was conceivable that there may not be an absolutely optimal method.

As further justification, consider the case of using the client-server model for main inter-device communications. With this topology in place, the server can act as an intermediary when setting up the initial session, since its address is always known by all participants. However, if peer to peer methods alone are used for communication, then there is no oracle that can introduce devices to each other, and some form of ad-hoc discovery method would be required.

At the initial stages of development, a simple single-user module was used to initialise the session, allowing testing of the data storage and visualisation components before further setup modules were implemented. As well as acting as a temporary placeholder, this module is activated in the final system whenever the user selects the "Single User" option at the main menu.

The first multi-participant session setup module was designed to be universal, since at the time developing further modules was an optional project extension so this may have been the only one. This meant that ad-hoc discovery methods had to be used since they don't rely on an external provider and can work with both server based and peer to peer systems.

There are several options for such ad-hoc device discovery.

All devices running Android version 2.2 and above that are Wi-Fi enabled are able to host personal Wi-Fi hotspots. One solution to the setup problem would be to have one device set up a hotspot and the other participants connect to that hotspot, then the host could identify and add all connecting devices to the session, and distribute the session information back to them over the connection.

A similar suggestion is to use Bluetooth as the medium. This would operate in a similar manner to using Wi-Fi. However for security, Android uses a primitive form of User Account Control (UAC) to ensure that some operations can only be done with explicit user permission. Pairing with a new Bluetooth device is one of these operations.[7] The principle of information hiding dictates that it is usually desirable for technical implementation details to be hidden from the user, but this method would violate that principle and complicate the process unless

all users had previously paired with this host, which is not uncommon though should not be relied on.

Near Field Communication (NFC) has recently been adopted by handset manufacturers, and allows communication by simply making contact between two NFC enabled devices.[7] QR codes allow communication via a visual channel and could be used by displaying a QR code on one screen while another devices directs their camera towards it. These two methods are similar in that they provide one-to-one communication and require one device to be physically oriented in respect to the other. This means coordinating a large group of devices could become a complicated process for the user, even with on screen instruction.

Bluetooth was chosen over Wi-Fi because it is a completely ad-hoc form of communication while Wi-Fi hotspots are not. Also for all users to connect to the hosts hotspot they would all need to know the password or it would be unencrypted, both are not ideal conditions of use for such an application. Wi-Fi direct would have been the best option but is not yet supported by most devices (including those available for this project).[1]

3.4.3 Bluetooth Session Setup

In order to initialise and distribute the session using Bluetooth, the following steps are taken:

- Master selects desired Bluetooth devices from a list on screen as shown in Figure 3.3.
- Meanwhile slave devices select connect and start listening for a Bluetooth connection.
- Master connects to each selected device in turn, retrieving its address information, as well as any necessary cryptographic information such as public keys.
- Master marshals session object from all of the collected information, including it's own.
- Master connects again to each device in turn and distributes the session object.

The available devices are retrieved from the OS using `getBondedDevices()` and the names from these are presented in the form of a ListView. ListView is provided in the Android API and is a subclass of the Android View class that

provides a way to display (optionally scrollable) lists of items on screen within an activity layout. When the choice mode is set to "multipleChoice" it lets the user select multiple items and provides the results in the form of a boolean array. This is done to let the user select which of its paired Bluetooth devices they wish to participate with. The resultant array is used to determine the desired devices and populate a list of them.

The master then goes through this list one by one and connects to the Bluetooth ServerSocket that each of those devices have opened, and uses these connections to exchange the required information.

3.4.4 Remote Server based Session Setup

While the Bluetooth based method worked well in testing, having to select multiple devices and then connect to them sequentially causes the process to take longer and require more effort than strictly necessary.

For this reason, another session setup procedure was built, using a remote server.

To connect, the user of each device presses the "Quickstart" button. This is the only user input necessary and is hence more convenient to use thsn Bluetooth.

When the function is activated the location of the device, obtained using a combination of GPS and Network provided data from the GPS Driver (See section 2.3), is sent to the server along with the session information of the device.

At the server, these requests are received and grouped by location into session groups. From these groups, session objects are marshalled and sent back to the devices involved.

The location information used for grouping comes in the form of 2D cartesian coordinates.

```
for device a: requests do
    a.wait();
    for device b: requests do
        if distance(a, b) ≤ x then
            a.add(b);
            requests.remove(b);
        end
    end
end
```

Algorithm 1: Naive grouping algorithm

```
for device a: requests do
    a.wait();
    for Block block: a.getBlock() + b.getNeighbours() do
        for Device d: block do
            if distance(a, d) ≤ x then
                a.add(d);
                requests.remove(d);
                block.remove(d)
            end
        end
    end
end
```

Algorithm 2: Improved grouping algorithm

If a naive grouping algorithm such as algorithm 1 was used the run time complexity would be $O(n^2)$ where n is the number of devices. This is avoided by noting that there is an upper limit on the distance between any two devices considered "close" to each other. This allows one to segment the 2D location space into a grid of disjoint areas, where comparisons need to be made only between points in adjacent segments, as is done in Algorithm 2. Note that both Algorithm 1 and Algorithm 2 require the list `requests` to be sorted in order of request time, this of course is done naturally when the list is populated. `a.wait()` simply sleeps until request `a` is old enough to be grouped. While still technically remaining $O(n^2)$ in the worst case (where all devices are close together), in practice there is an upper limit on the number of users that can be in the same place at the same time resulting in an effective O(n) runtime complexity to process all groups.

In practice when this method is available the speedup compared to the Bluetooth method depends on the length of time that the server waits before grouping requests, since they will not all arrive simultaneously. One important benefit is that it scales gracefully with multiple devices unlike the Bluetooth method which is serialised (see section 4.2.1). While being more convenient this comes at the cost of expressiveness since arbitrary participant selection is no longer possible. If desired, an option could be added however to allow selection of participants without much further work.

Clearly the Bluetooth method has the important advantage of not relying on an external provider.

3.5 Map Display and Data Structures

There are several ways the location data could be visualised on screen. The approach taken here is to draw a line to the screen for each user in the session, showing their entire path since the event began. The lines could be super-imposed onto existing map visualisation data such as that provided by Google[1] but with the desire to keep the interface simple and quickly interpretable this wasn't attempted. An example from the applications display is given in Figure 3.4.

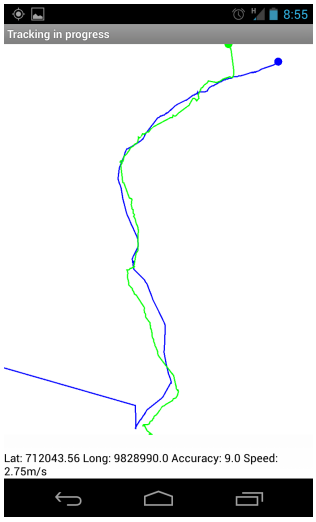


Figure 3.4: Screen shot of the Map Display Screen showing the visualisation of data during a field test

3.5.1 Storing Location History Data

For each device, the entire known location history, whilst in the current session, should be stored at every other device so that each device can potentially perform arbitrary computation and aggregation on it.

To do this, a data structure was required that would store a list of (time, location) data points.

The structure chosen was a linked list of array lists. The data-points are stored in fixed size arrays, where more arrays are allocated when needed. This means append, lookup and insert have runtime costs of $O(\lceil N/L \rceil)$ where $N =$ number of points and $L =$ array size.

For each device, this structure is indexed by an integer logical clock governed by the device the data originates from. This means the time between history

[1]Map data available at https://developers.google.com/maps/documentation/android/

updates is varied depending on the speed of travel resulting in constant spatial resolution. Its achieved simply by incrementing the logical clock every time it travels some fixed size distance. It also provides a simple way to check for missing data, since if there exists any empty index i such that $i < j$ where j is the latest known index, then the data for index i is absent and should be requested. Requests are explained further in 3.6.4. The alternative would be to index the lists by time, which would suffer from variable path resolution depending on speed, and several undesirable effects such as duplicate locations being stored whenever a device remains stationary.

Since the aim is to fully populate the list with data points up to some logical time l, allocating arrays is not wasteful because we always aim to fill all current gaps in the arrays. The trade off here comes with setting the length of the arrays, too large and more memory will be unused at the end of the array, too small and lookup times will increase because of the increased size of the linked list.

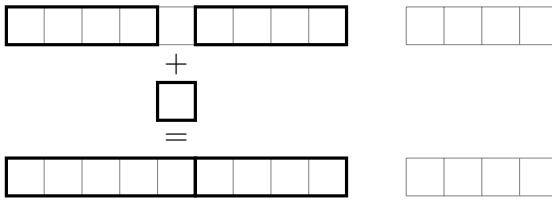
3.5.2 Drawing to the Screen

To present a visualisation to the user, the data points need to be mapped onto a 2D space the size of the device screen, and lines need to be drawn between them.

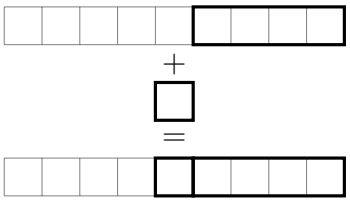
The android canvas API provides suitable methods for plotting points, lines and paths to the screen.[7] In this context, a Path object represents an append-only ordered list of points that can be pre-computed and drawn to the screen efficiently at different times, through the use of the canvas of a View object. Being append-only, the Path objects don't provide an immediate solution for drawing entire device histories to the screen for each frame. The reason for this is that at any point in time, there may be absent data points between two present data points. Since these absent points may later be obtained, the data structure should allow for insertion and not just append without having to reconstruct the entire Path object again.

A View object is a java object that can be embedded into the layout of an Activity. Whenever the activity is created, so are the embedded View objects. A custom view object was created, named a MapDrawer, which is a rectangular canvas that can be displayed on screen, and has access to the pathCache structure decribed below for drawing the traces.

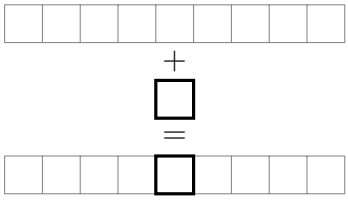
Figure 3.4 shows the MapDisplayScreen Activity, the main element of this screenshot which contains the different coloured lines is the MapDrawer View Object.



(a) Insertion into a single index gap



(b) Insertion at the end of a gap



(c) Insertion into the middle of a gap

Figure 3.5: Figure showing how insertion into the pathCache occurs in different contexts. Boxes of indices denote Complete Segments, sets of unboxed indices denote Gap Segments.

The PathCache

To solve the Path data structure problem, an object named a PathCache was designed and implemented. This is comprised of a Tree Set of Segment objects, where a Segment may be either a Complete Segment or a Gap Segment. Complete segments represent parts of the data point list that have no missing data. Gap segments are segments of the list that contain only missing points. Each CompleteSegment object has a standard Path object which contains its datapoints and can be drawn to the screen.

Each Complete segment contains a Path object that can be drawn to the screen.

This structure provides efficient insertion of data points into the lists as well as conserving the efficient append operation when using a single Path object. Figure 3.5 shows how datapoints are inserted into the pathCache. To append points, they are simply appended to the Path of the last CompleteSegment (the last segment will never be a GapSegment because gaps are defined as missing data points with index less than that of the largest index received).

When inserting points there are two possible cases.

In the first case the new point is at the start of a GapSegment, in which case it gets appended to the preceeding CompleteSegment, and the start of the GapSegment is moved along by one, or removed it it was of length 1.

In the second case the new point is not at the start of a GapSegment (meaning it is somewhere else in some GapSegment). In this case the GapSegment is shortened, a single point CompleteSegment added for the new point, and if the point wasn't at the end of the GapSegment, then an extra GapSegment is added to fill in the rest of the old GapSegment. The TreeSet provides mapping from indices to Segments (as well as fetching predecessor and successor Segments) in O(log(N)) time.

While not necessary for an understanding of the structure, it may be noted that GapSegment objects need not contain any information. The start of the any Segment (Complete or Gap) is defined by the integer index of it in the TreeSet and the end of it is defined as the integer index of its successor in the TreeSet, unless there is none - this is handled by keeping a single "largest index" integer in the pathCache. Therefore `null` would suffice in place of all GapSegment objects, instead for the sake of good OOP practice, a single instance of a GapSegment exists and may be referred to in the TreeSet several times[2].

To draw the paths to the screen each frame, every entry in the TreeSet is iterated through (in index order), with the Path object of every CompleteSegment found being drawn to the screen. For each GapSegment found, a straight line is drawn from the end of the previously drawn Path to the start of the next.

When scanning through the Path objects, the boundaries are kept track of, these are the maximum and minimum values occurring on each axis. They are used to calculate the scale factor required to map the life-size model to the scale of the screen. A matrix is constructed from this factor and applied to transform each of the paths, then they are shifted to be aligned at (0,0). This results in the paths being displayed always at the largest possible scale without distortion or exceeding the bounds of the screen.

3.6 Data Sharing

As mentioned earlier the networking parts of the system were sufficiently loosely coupled to enable modular swapping of different protocols. This was achieved using standard OOP conventions and the use of an abstract class ProtocolManager which was subclassed by the various different networking implementations.

The approach taken to send data was initially a simple bucket and send protocol. This means that as new points arrive from the location aware part of the

[2]This allows for "cleaner", more uniform segment checking using `instanceof(GapSegment)` rather than `== null`

system, they are put into a bucket and sending is delayed until either the bucket is full or the first item in the bucket is older than some fixed amount of time.

This prevents abusing the network by sending many smaller than necessary packets, as was seen to be a problem in 2.2.1, while avoiding excessive sending delays.

There are two ways new data can enter a client system: from the location aware part of the application (GPS or network provided data) or from the network of other devices. Depending on the case, one of the methods `insertOriginalDatapoint()` and `insertNetworkDatapoint()`, which are static methods of the ProtocolManager class, is called.

These call the necessary methods to add the new points to the PositionStore. In the case of `insertOriginalDatapoint` it also adds the new data to the outgoing queue to be broadcast to other devices.

3.6.1 Asynchronous data notification

An interface `PositionStoreSubscriber` is defined that lets implementing classes subscribe to updates from the PositionStore. Whenever new points are added to the PositionStore, it alerts the subscribers of the update (by calling `notifyOfUpdate()`) so that they can be notified and process the new data asynchronously.

The objects that subscribe to updates are the MapDrawer in the client application, and in the server application where there is no MapDrawer component the singleton ServerState instance subscribes. The ServerState is responsible for managing the server-specific state and dispatching messages to clients.

3.6.2 Iterative construction of a simple server

For the first implementation, the client-server model was chosen. This means that all communications between devices would be via a central server, effectively forming a star topology. This simplifies the client side code since clients only ever need to send and receive messages to and from a single address. First the sending only aspect of the client side code was written, and a prototype java server application was made that would print details of received data so that the client code could be debugged.

After this, the core framework of the client side app was added to the server, including the PositionStore and message formatting code.

The server acts like a client, in that it has the same data structures for storage, and receives packets containing data points and consequently updates it's state

to keep track. Differences arise where instead of a MapDrawer object being subscribed to new data updates, the part of the application that coordinates the outgoing messages, ServerState, is subscribed.

3.6.3 Peer-to-peer version

For the peer-to-peer implementation of the system, each device has a DeviceConnection object for each other device. The protocol as usual is flexible and left to be implemented by the ProtocolManager subclass. In this case what has been done (in ProtocolManagerP2P) is when a device generates new points, it sends out the payload message to all other devices, by iterating over the DeviceConnections. Other approaches were identified during the preperation stage, such as forming a ring network or pairing up devices based on spatial locality, but with the time constraints these were not implemented. These approaches are explained in Appendix C.

In ProtocolManagerP2P, requests are always sent to the generators of the requested points and responses are sent to either the requester or all other devices, as specified in the configuration file.

3.6.4 Requests

Since the User Datagram Protocol (UDP), an unreliable transport protocol, is used to transfer history data between devices, some method of retransmission is required if a complete knowledge of path history is desired at each device.

Two message types have been defined for inter-device communication once the main protocol is underway. The first is the payload type which carries historic location data about possibly several devices, and the second is the request type which lists the datapoints that are absent from the PositionStore of a device. The formats of each are given in Appendix E.

A request system was implemented which allows each device to determine the data it is missing, and request it from the network by way of a request message.

The way this is done is by maintaining a list of the currently absent data points. When a new point arrives, if it has a larger index than any present datapoint, all indices from the current latest one up to that index are added to the absent list. If the point is not the new latest point, it is simply removed from the list.

With this absent list, requests can be made to the other devices and/or the server for them. Because of the monotonic indexing system for each device and

the way the absent list is maintained, the presence of an index in the absent list implies that it *is* present at at least one other device.

The way that requests are made and responded to has been purposefully made flexible. The implementation details are left to the subclass of ProtocolManager that is being used, though for this project the only request procedure used is to send out a single request message listing the entirety of the absentList periodically whenever any data is missing.

Several response decision policies have been implemented which govern how devices respond to requests. An Enum in the config file specifies which policy should be used. The implemented options are:

- Always respond if this device has any of the requested data
- Respond with probability p, if this device has any of the requested data
- Respond if this device has $> n\%$ of the requested data
- Respond with probability p where $p =$ the fraction of requested data that this device has
- Respond if any of the requested data was originally generated by this device
- Respond if, of the requested data, this device generated the largest proportion of it
- Never respond

When the client-server implementation is in use, devices only ever communicate directly with the server, so can only make requests to the server. When this occurs, since the server too stores history data, it looks up the requested data in its own PositionStore. The requested data that it finds are termed the *hits* and the remaining requested indices are the *misses*. The server replies to the requester with the *hits* in the form of a payload message. It then groups the list of *misses* by the device that originally generated them, and to each device it issues a new request containing the group from that device. There is no value in requesting the data from any device other than the generator as no other device can possibly have it since all messages must have gone via the server.

3.7 Extra functionality for development only

During the project there were certain stages where testing of currently implemented features was difficult before completion of the next piece of work.

One of these cases was the MapDrawer object, which was developed prior to the location-aware part of the application meaning there was no available information feed into the module for testing purposes. In this case, random data was generated and used in place of real location information. A screen shot of the application using such a method is shown in Figure 3.6. This allowed for testing the display before the location code was built, but also made testing more convenient when using Android virtual devices running on a PC which doesn't have GPS.

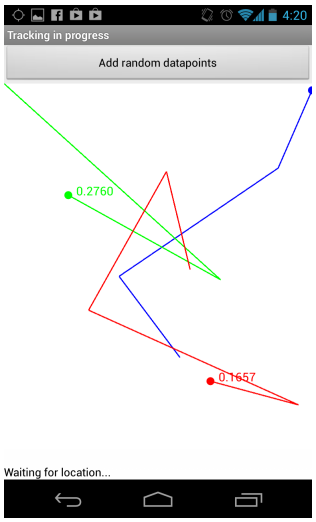


Figure 3.6: Map Display Screen showing a session running with randomly generated location data

At the start of the project, only one physical device was at hand for testing. To test the message processing code at this stage, a "duplicate" option was added to the server that caused it to duplicate every recieved point, change the device ID and translate the location by 10m. This extra point when inserted, simulated the interaction of a second device and would would show up on screen as having a position right next to that of the real device.

A Logger class was also built which subscribes to the PositionStore update facility and keeps a record of all notable events[3], this record is written to disk when the session ends, and was used to do a quantitative evaluation of the system in chapter 4.

[3]This includes events such as receiving a packet and issuing a request, along with information such as the size of the received packet etc.

3.8 Concurrency

One of the main challenges in the project was dealing with concurrent threads. This includes spawning threads when needed, synchronizing threads and ensuring thread safety, and doing object de-allocation - a problem usually considered solved in sequential Java which brings new challenges when multiple threads have references to the same objects. In Android, each application runs as its own process, but freedom is given to use as many threads as desired. The development guidelines offer advice on using threads, an example being to never construct new data structures on the UI thread to prevent unresponsive delays in the user experience. All of the official guidelines were followed throughout development, and there were many points in the project where concurrency was used.[7]

Whenever concurrency was present, use of data structures and state was carefully tracked and the `synchronized` keyword was used to enforce mutual exclusion between conflicting methods.

In addition to this, `destroy()` methods were defined for all classes with static state and these were called in a cascading fashion whenever de-allocation was due, such as the user exiting an ongoing session back to the main menu. Also a global synchronized boolean variable `alive` was defined in `ProtocolManager` which is checked for in all situations where execution could otherwise continue after the user has exit (including at the wake up of any sleeping threads). When the user does exit, this is set to false and then all orphan threads quietly terminate.

Chapter 4

Evaluation

In this chapter I will describe how I verified the success of my project and provide qualitative and quantitative measurements of the performance of certain aspects of the system.

4.1 Field Tests

Arguably the most sensible way to test such a system is to trial it in the environment for which it was designed. This means installing the application onto real android devices and having several people navigate a circuit while operating them.

Concern: does this count as a user study? needing all the ethics approval? - if so can just do it with single user.

4.2 Comparison of Session Setup procedures

4.2.1 Timing

To make a quantitative comparison of the two setup procedures implemented would be difficult because the user experience is hard to quantify. Despite this, timing measurements were made to give an idea of the difference in time taken between them.

In the experiments carried out, to setup a two-device session, the Bluetooth procedure took on average 4.8 seconds[1] and the quickstart method took on aver-

[1]This increased to 16.0 seconds when Wi-Fi was enabled on the devices tested, this is explained by the fact that Bluetooth and Wi-Fi share part of the same spectrum and intrfere somewhat.[8]

age a very similar 4.7 seconds. However these measurements are of the minimum setup time, meaning the time for selection of participants or grouping is set at zero. In reality, this would take additional time depending on the situation. It is not clear which of the Bluetooth user selection and the server waiting time would be quickest but it is clear they are within the same ballpark.

What is significant is that the quickstart method is highly parallelized since the bottleneck for each device is the network latency to the server, and these bottlenecks are independent, resulting in constant time setup[2]. For the Bluetooth method however, the most time consuming operation is the connection initiation between devices which must be serialised so the time taken increases linearly with the number of devices.

This makes the quickstart method faster in all cases[3], but especially so for larger groups.

4.3 Security and Privacy Issues

Since the application involves the transmission of location data along with the names of users, there is potential for breaches of privacy by leaking information to those outside the session, or by false locations being provided to those within.

For the purposes of this evaluation, I will define the term *secure* as ensuring authenticity of the sender of all received messages, and integrity and confidentiality of the content of those messages.

Both session setup procedures provide the ability to distribute cryptographic keys for use in the main communication phase. This presents a multitude of end-to-end encryption options for this phase, one of which would be to use a public key system with the following setup:

Each client has their own public/private key pair, and those public keys are distributed with the session at the start. Confidentiality is achieved by encrypting messages with the public key of the recipient, requiring the private key to decrypt them. Authenticity and Integrity are ensured by digitally signing each message with the private key.

An alternate option if authentication between members of the session is not required, would be to use a shared key scheme, where on session creation a single public/private key pair is generated and shared between all clients.

[2]Counting the server grouping operations as negligible in comparison with the network latency.

[3]Provided network latency is similar to those at the time of the experiment, and assuming the time taken to select participants is equal to the server grouping timeout

The above schemes do not require the use of any Public Key Infrastructure (PKI), instead the keys are distributed along with the session object at the start. Therefore the security of the entire scheme relies on the security of the session setup protocol.

4.3.1 Security of Bluetooth setup procedure

When security is concerned, the physical proximity requirement of Bluetooth transmission aids its security beacuase an attacker would need to be physically present. All transmissions using Bluetooth are encrypted using the information established at the time of pairing, and Fast Frequency-Hopping Spread Spectrum is used for the radio frequency making eavesdropping practically impossible without the key.[9] However it has been shown that with the ability to eavesdrop the pairing process, the key can be feasibly deduced by brute-force, providing access to the later established communication channel.[10][11] For these reasons the Bluetooth method is only *secure* provided an attacker is not present at the time of pairing. However since the pairing of Bluetooth devices is an abstraction provided by the operating system, and not restricted to, nor within the control of this application, it is reasonable to assume all existent pairings are trustworthy. The application itself, is *secure*[4].

4.3.2 Security of the QuickStart method

Simply put the QuickStart procedure is not *secure*. Since the peer grouping relies solely on location data provided by the client and this data is not validated, an attacker can submit an arbitrary location and be grouped with participants at that location, gaining access to their location for the duration of the session without even costing their own privacy since they can spoof their location in the main phase too. Thus confidentiality is not provided. In the same way an attacker can spoof their identifier, posing as any arbitrary participant. Thus authenticity is also lost. Integrity remains.

This applies to the procedure as implemented, but it is possible to make it *secure* with some, quite significant alterations. One partial fix would be to add a user selection screen after the initial grouping as mentioned in 3.4.4. This would provide confidentiality, since data would only be sent to (and readable by) those selected, but not authenticity as identities are still ambiguous.

[4]As always in the field of security, the security of the entire system should be taken into account - it is not sufficient to blindly rely on the application itself.

A PKI could be used to provide the desired security features as follows. Each device, on installation of the application (in collaboration with the server) generates a lifelong public/private key pair and unique identity, such as a username. When requesting to the server to join a session, the server responds with a challenge that the client must respond to and sign it with its private key. This authenticates the client to the server, the server then groups the requests as usual and replies with the session object encrypted to each recipient using their public key. Server authenticity is also required to prevent MITM attacks between the server and client, so all server bound messages will be encrypted using its known public key, and it will sign and encrypt all of its outgoing messages.

Now when the user selection occurs, all identities will be unique so selection is no longer ambiguous and authenticity is provided for the duration of the session, as long as all further messages are signed and encrypted using the same keys.

The addition of a PKI would be a significant addition to the project but would provide the QuickStart method with the desired properties.

4.4 Packet loss and Comparison of Network Architectures

To compare the performance of the different protocols, a considerable amount of time was spent working out what information to log, and implementing this logging functionality so that the results could be compared against each other and interpreted easily.

As described in section 3.7 the logger outputs a .csv file to disk at the end of each session. Files containing Comma Seperated Values (CSV) were chosen as the format because they can be easily imported into spreadsheet software to find manual trends, as well as almost all other plotting software. Microsoft Excel was used for manual examination of the graphs produced. A truncated version of one such output file is given in Appendix D.1.

In excel, a spreadsheet was set up that aggregated the data of all log files from the same session to provide more insightful data such as the transit time of packets to different devices.

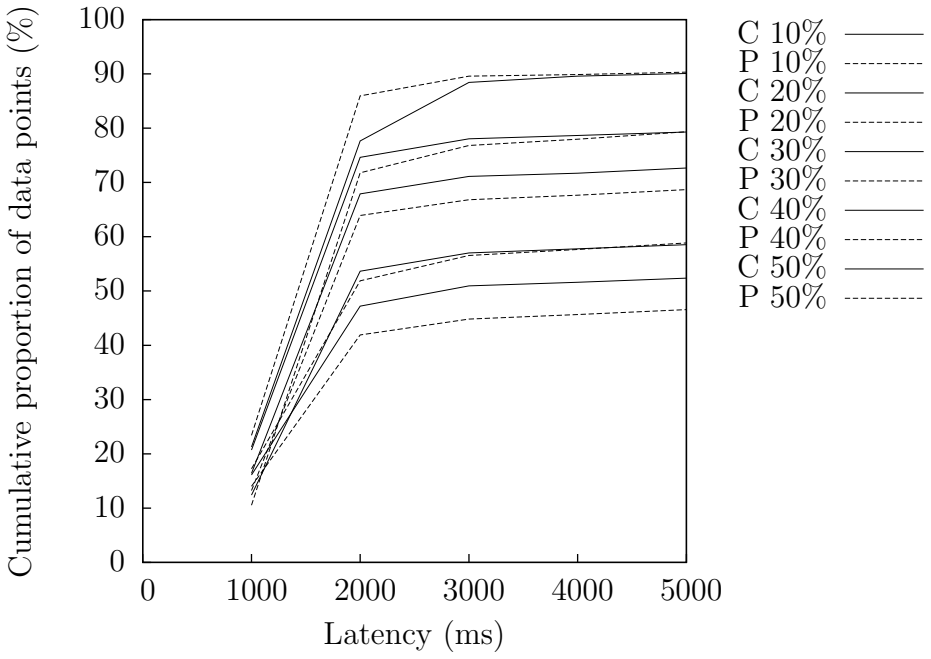


Figure 4.1: Cumulative frequency graph of data point latency. Lines are labelled 'C X' or 'P X' with 'C' representing Client-server, 'P' representing Peer-to-peer and 'X' being the drop rate.

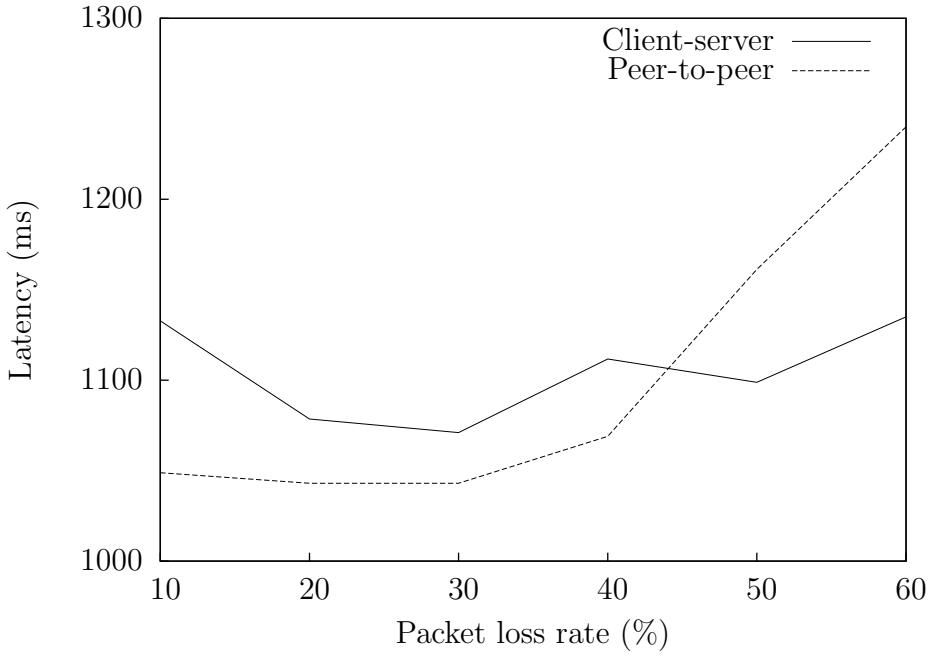


Figure 4.2: Average latency of different topologies with increasing packet loss

To gather the packet loss data shown in Figure 4.1, code was added to the client to randomly discard incoming or outgoing packets depending on the variables in the configuration file. This is to simulate the loss of network packets due to wireless interference / signal issues. For these tests, the system was run on a 3-device session for 10 minutes on each trial. Three trials were done for each 10% increment from 10% to 50% drop rates and median values were used.

Figure 4.1 shows that when packet drop rates are low ($< 10\%$), at least 80% of datapoints are successfully delivered within 2 seconds. This number decreases as expected when drop rates increase, with 40% of datapoints taking under 2 seconds for drop rates of 50%.

Figure 4.2 shows that for drop rates of $< 40\%$, the average latency of the peer-to-peer system was less than that of the client-server system but increased more quickly with increasing packet loss. Above about 40% the client-server system has the advantage, supporting the theory explained in section 2.2.2.

4.5 How the application scales

As described in the proposal, the server application was modified to simulate an arbitrary number of devices, interacting with a real device running the client application. This allowed scale tests to be conducted with relative ease. Figure 4.4 shows how the average latency and sustained success rate varies for different session sizes.

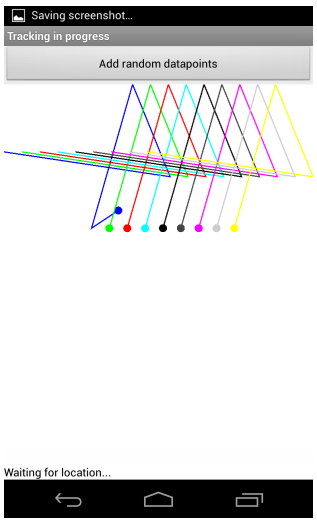


Figure 4.3: Map Display Screen during a simulated 9-device scale test on random location data

In maximum load tests the performance benefits of the client-server model were observed. With the server doing all of the coordination and routing, the client applications are spared much computation. This leads to tangible differences in the scalability of the system.

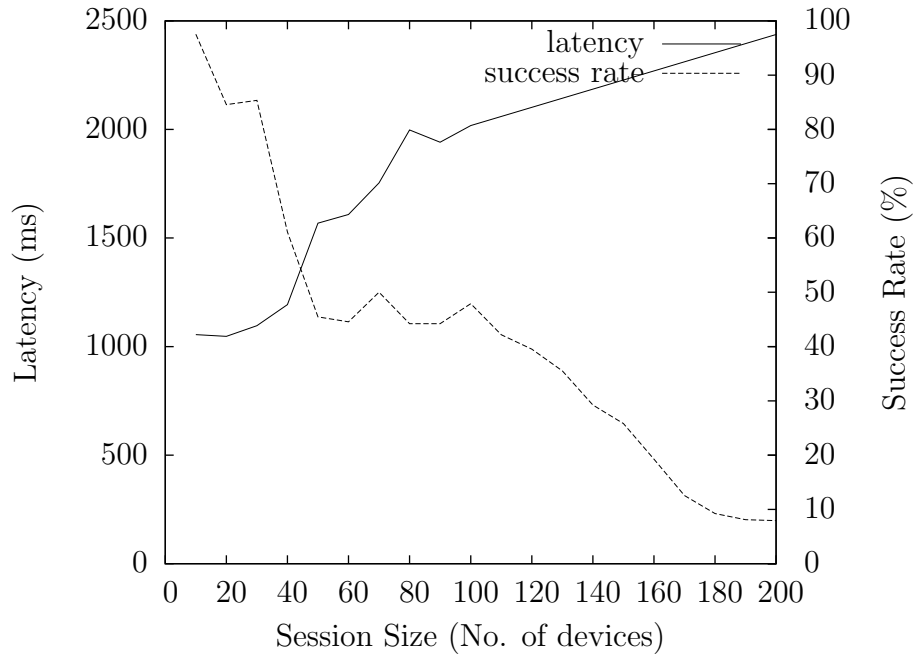


Figure 4.4: Graph showing how average latency and sustained throughput is affected by increasing session size

With each device producing one data point per second the Peer-To-Peer system was rendered useless[5] at 1024 devices per session. At this point, the networking requirements of the application exceeded the devices capabilities[6]. This could be improved by sharing data in a more structured arrangement (See appendix C) so that each device doesn't need to maintain as many connections. In the same conditions, the client-server implementation ceased to be of any use with 2048 devices. Both of these maximum values are far greater than the session size the application was designed for. With session sizes of fewer than 10 devices, the success rate is consistently 100%. Nevertheless, with more intelligent grouping of information and more fine-grained locking between threads, it is believed that these limits could be increased if desired.

It is worth noting that in this case, scalability refers to the number of devices in some given session, and not the number of sessions. When the number of sessions

[5]The point at which all incoming updates cannot be processed in time

[6]Device used for this test was a Google Nexus 1

is considered, the Bluetooth method will excel in session setup performance, since all setup work is distributed.

4.6 Priority of sending

Since the distribution of new data is always done in one thread, and the requests and re-distribution in another, priority is given to the distibution of new data because there is less of it. This prioritising was an original aim in the protocol and has been observed in the results of the tests conducted.

Figure 4.1 shows that the system was able to deliver the majority of data points within 2 seconds whenever the packet drop rate is 40% or less, and in fact figure 4.2 shows that in every test (up to 60% drop rate) the average latency did not exceed 2 seconds.

As can be seen in Figure 4.4 of Section 4.5, even with packet success rates of less than 20% the average latency[7] was less than three seconds. Therefore on average, the points that were drawn ($< 20\%$), were relatively recent (at < 3s) so the desired prioritising of recent points has been achieved.

This shows that even when the system is struggling, it still provides useful rather than out of date data.

4.7 Meeting the original goals

The initial success criteria outlined in the proposal were:

- App displays an on-screen visualisation of data from the local GPS
- App plots positions of up to at least four participants (including itself) in near real-time on the map
- At least one server based method and one peer to peer method of communication has been implemented, with comparisons of their performance made

All above criteria have been satisfied. Another informal goal in the proposal was to evaluate the security and privacy issues of the system. This was done in Section 4.3.

[7] Time between generation of a point and it being drawn to the screen of another device

Chapter 5

Conclusion

As explained in section 4 all original goals have been met and the implementation and evaluation is regarded successful. During the evaluation I was able to appreciate the difficulties of quantitatively evaluating real life systems.

I am certain that the application would be useful in addition to any physical mounting equipment required for use. All practical field tests yielded positive results and the quantitative evaluation shows that the system is theoretically useful.

I plan to continue the development of the system by adding some of the features of section 5.2 and eventually publish it.

5.1 What I would have done differently in hindsight

5.2 Future Work

There are many ways this work could be extended to provide useful and/or interesting features.

Communication

One aspect that was considered at the start of the project is to make use of multi-channel communication. Currently all communication during the main phase is done through one medium, either Wi-Fi or mobile internet. Further network interfaces could be added, for example Bluetooth or Wi-Fi direct and used in an opportunistic way to transfer information between devices whenever possible. The whole set of channels could be used simultaneously providing redundancy

and application flexibility. The flexibilty would come in that not all devices would be required to have, say a mobile internet service, they could instead leech off of the other devices[1].

In addition, multipath TCP (MPTCP) or similar protocols could be used to provide seamless connection while moving through different networks, say from 3G to Wi-Fi as you proceed around the track. MPTCP effectively allows the simulataneous use of a single TCP connection over several channels.

Different ways of structuring the device space were explored at the beginning of the project but not adopted because of timing constraints, these are elaborated in Appendix C. Such grouping of, or affinity between devices based on proximity or other metrics could be used to find more intelligent continuous distribution routes. Using the various wireless adhoc technologies available this could potentially be vary advantageous.

As previously mentioned in 2.2.3 IPv6 is increasingly being adopted, this would eliminate the NAT problems and allow the peer-to-peer implementation built here to operate properly on 3G (and beyond) data networks.

Another desirable feature of the system would be some reflective aspect that alters the protocol parameters based on current statistics. Characteristics of different (even just mobile) networks vary from one to another (See section 2.2.1), as well as in time, so an efficient protocol should make changes according to the current network state.

Application features

Another way to continue the work would be to improve the application and what it does with the data.

A desirable feature would be to have a real time leaderboard that shows the current ranking of participants during the race. It could be augmented with estimates of the time or track distance between the user and other participants to give a numerical value rather than relying on only the visualisation. This would require some method of determining which of two participants is ahead of the other in the race, a decision that may not be trivial when complicated circuits are used.

As mentioned earlier, the user interface including the track visualiser could also be improved, while still keeping the underlying program the same.

[1]The system as implemented does already have such a leeching feature. The tethering capabilities of devices are used to share their connection while participating, whenever an non-internet enabled device is within Wi-Fi range it automatically connects and synchronises its history data.

Bibliography

[1] *Wi-Fi Direct*,
Wi-Fi Alliance, 2013
http://www.wi-fi.org/discover-and-learn/wi-fi-direct

[2] *Android 2.2 Platform Highlights*,
http://developer.android.com/about/versions/android-2.2-highlights.html

[3] *Android Devices - CPUMark Rating*,
PassMark Software, 2013
http://www.androidbenchmark.net/cpumark_chart.html

[4] *RFC 2460 - Internet Protocol Version 6 (IPv6)*,
Internet Engineering Task Force, 2013
http://tools.ietf.org/html/rfc2460

[5] *Universal Transverse Mercator System*
Steven Dutch, 2013
http://www.uwgb.edu/dutchs/fieldmethods/utmsystem.htm

[6] *JCoord*
Jonathan Stott, 2013
http://www.jstott.me.uk/jcoord/

[7] *Android API Guides*,
http://developer.android.com/guide/components/index.html

[8] *Wi-Fi and Bluetooth - Interference issues*,
Hewlett Packard, 2013
http://www.hp.com/rnd/library/pdf/WiFi_Bluetooth_coexistance.pdf

[9] *Security of Bluetooth: An overview of Bluetooth Security*
Marjaana Trskbck, 2013
http://www.cs.hut.fi/Opinnot/Tik-86.174/Bluetooth_Security.pdf

[10] *Sniffing the Bluetooth Pairing*
Alberto Moreno Tablado, 2013
http://www.seguridadmobile.com/bluetooth/bluetooth-security/sniffing-the-Bluetooth-pairing.html

[11] *BTCrack Source Code*
Thierry Zoller, 2013
http://blog.zoller.lu/2012/07/btcrack-101-updated-release.html

Appendix A

Project Proposal

Appendix B

UML Class Diagram

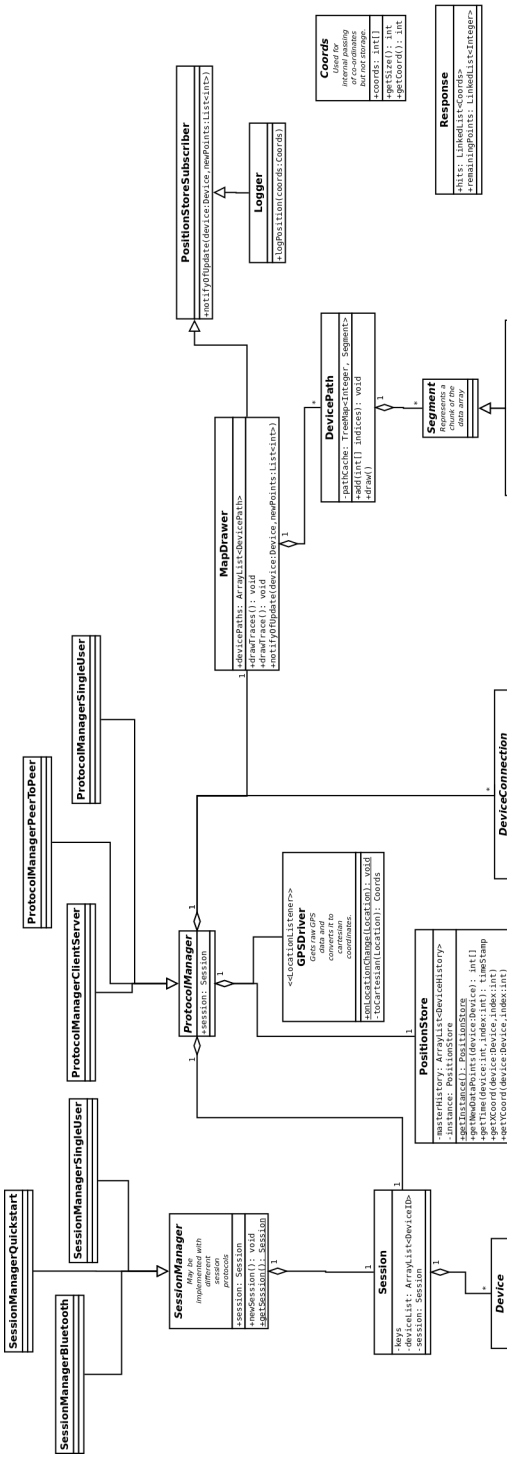


Figure B.1: UML Class Diagram showing the main classes that make up the system.

Appendix C

Structuring the Device Space

In this project, a flat device space is used, meaning that there is no particular association between any two devices, they all have uniform connections with the others.

This

Appendix D

Example log file

```
To do.
```

Figure D.1: A truncated .csv file output by the logger function at the end of a session

Appendix E

Format of Messages