Imperial College London

Computing

MEng Project Report

Lokey:

Location Awareness Revisited

Author: Krishan Patel

Supervisor: Naranker Dulay

Second Marker: Michael Huth

June 18, 2013

Abstract

The proliferation of smartphones continues to create new markets for handheld applications. Smartphone apps can now be tailored to truly understand their users and their surroundings. This opportunity remains untapped by a majority of the development community. With every update, dominant mobile operating systems attempt to make it easier for developers to integrate location services. However, this has only had a small effect until now; the majority of apps that do use location features are apps that focus solely on location services. We believe that the ability to integrate more complex location features into apps that do not rely on location alone will increase the benefit for users.

Conversely, users are often wary of apps that use their location information. Even if an app is location aware and provides a great service through location analysis, some users feel this is an invasion of privacy or are worried that their information may be used maliciously.

Lokey is designed to solve both these problems. By giving developers a more feature-rich location framework, Lokey allows them to spend less time trying to implement location services and more time on the rest of their application. Users also benefit as Lokey allows a fine-grained allocation of permissions, rather than the current absolute 'allow' or 'disallow' option. This investigation measures the effectiveness of the framework by producing a location dependent app using only Lokey for all the location features.

Acknowledgements

I would first like to thank my supervisor, Dr. Naranker Dulay, for his invaluable assistance throughout the course of this investigation. My thanks also go out to my second marker, Professor Michael Huth, for his vital inputs after my interim report. I would also like to thank my friends and family who have always believed in me and supported my work. Their real-world testing was a key factor to the completion of this investigation.

Contents

1 | Introduction

In recent years, location aware mobile applications have become increasingly popular. By understanding the user's location, an application can become tailored to the user's surroundings. Mobile applications such as friend finders and location aware search engines are starting to become very popular in mobile marketplace.

However, the current mobile app stores comprise of either applications that are location dependent (i.e. they only provide location based services) or applications that do not use location services at all. The reason for this is the relative complexity of integrating *useful* location features in apps whose main feature set is not location dependent.

For example, a deals application, like Groupon [11], could potentially provide a richer user experience if it knew what stores the user was likely to visit. By using this information, the application can become adapted to only show deals that are relevant to the user. Another example is a taxi application. If the app had access to the users most common locations, it could calculate the cost for a cab to these locations without the user having to explicitly insert their information. This would provide a far smoother experience for users.

The complexity of developing a solution for problems like this lies in the fact that developers are only given access to the users current location (discussed further in section 3.1.1). To acquire the kind of data mentioned in the example above, developers would have to create their own location tracking service, which is not the easiest of tasks.

On the other hand, privacy remains a concern for users. With new companies emerging as information sources everyday, users are growing increasingly cautious of what information to share. This is particularly a problem for applications that use location services, as users often fail to see the need for an app to know where they are. Location awareness would allow user to have an enhanced experience and developers to have higher user numbers and/or higher incomes. One of the main reasons for this concern is that users are generally not made aware of what data is being saved about them, or why applications need this data in the first place.

The aim of this investigation is to combine these two problems and produce a framework that satisfies the needs of both developers and users. Our framework will provide apps with access to richer information about the user's location. For example, we will give applications the ability to know where users are commonly located at certain times of the day. To do this, we will create a background service that runs on the users device and continuously tracks their location. We will allow users to view any data that is being saved about them, and deny apps from using them if necessary.

A key component of the success of this service will be the extent to which it minimises resource usage. Though preliminary investigations, we have found that constantly tracking location through GPS can drain the battery as quickly as 9% per hour. By using network signals instead, battery usage can be significantly reduced to 1% per hour. However, network based locations are not as accurate as GPS. We investigate techniques to reduce the uncertainty of future readings by tracking the users position continuously. By using techniques such as dead reckoning and the Kalman filter (discussed in section 2.3.1), we are able to significantly improve the precision of location estimates.

Using a variety of computing techniques, we will produce an API that uses the gathered data to provide useful information. We will employ a range of techniques from machine learning, to cluster locations relevant to the user, to using decision theory to accurately predict a users movement patterns. By conducting these calculations in a central background service, we reduce the amount of computation done on the device as other apps will not have to carry out the same calculations.

The framework is designed to be both broad and deep, to ensure that developers can find a variety a features, with as much precision as required. Relevant features have been determined by consulting current mobile developers. We have asked them why they would use this framework, what they would require, and what they would use it for. By asking existing developers, we gathered a deeper understanding of what is currently lacking, and what would be the most helpful.

For the users, we have designed the framework to be completely transparent. By allowing users to access and even modify the data stored about them, we believe users will be able to have more trust in this framework. This works as an advantage for the users, as well as an incentive for developers to use this framework instead of rolling out their own solution. Users are also be able to see what information applications are requesting and are able to deny those requests if they wish.

1.1 Contribution

The main contribution of this project is a mobile framework, Lokey, built to gather accurate data about the location of users. This data is converted into useful information that is made available for use by developers to facilitate easy development of location aware applications. The framework has a user interface component from which end users can see and manipulate what data has been gathered. Users are also able to see which applications are accessing this data, and what parts of the data they are accessing. An in-app permissions system allows users to allow or deny each of these applications access to individual components of their location data.

The framework collects a number of interesting features about its users; for example it gathers the users frequently visited locations and the users common journeys. These features are further discussed in Chapters 4 and 5. The framework also provides applications with ‘wake-ups’ when certain location based actions occur. This allows applications to register their interest in events such as when the user reaches a certain area or when the user starts moving. Using more complex techniques, the framework is able to accurately extract predicted information about the user. For example Markov models are used to

predict where the user may be going when they leave a location.

The framework is judged, in part, by its accuracy and resource usage. Our framework consumes almost no battery when the user is idle. We found that, when travelling, battery usage is approximately 1% per hour. These impressive results are due to the *adaptive updates* technique we discuss in Chapter 5. This technique allows us to maintain high levels of accuracy even with reduced battery consumption.

The ease of use of the system is judged by development effort required to build a sample application. We found that development of an application that would have required a significant amount of effort was made simple to build by using our framework. We also conducted interviews with a number of developers to ascertain whether they would be willing to use a system like this, and the initial response was very positive.

2 | Background

This section presents any work carried out by others which is relevant to this investigation. We will go through any foundational work or ideas that we have built upon. We will also provide results of any preliminary work we have carried out.

2.1 Location Properties

In order to explain more complex features of location systems, we will first go through the basic building blocks. This section provides formulae and ideas used by the majority of location systems implemented today, and will be used by our system.

2.1.1 Geographic Coordinate System

A geographic coordinate system allows every location on the Earth to be specified as a coordinate. The most commonly used coordinate system is the latitude-longitude system, where:

- **Latitude** (ϕ) is defined as the angle that a straight line between the location and the centre of the earth subtends with respect to the equatorial plane (the plane the forms the equator where it cuts through the surface of the Earth).
- **Longitude** (λ) is defined as the angle between the meridian a location lies on and the prime meridian (which goes through the Royal Observatory in Greenwich).

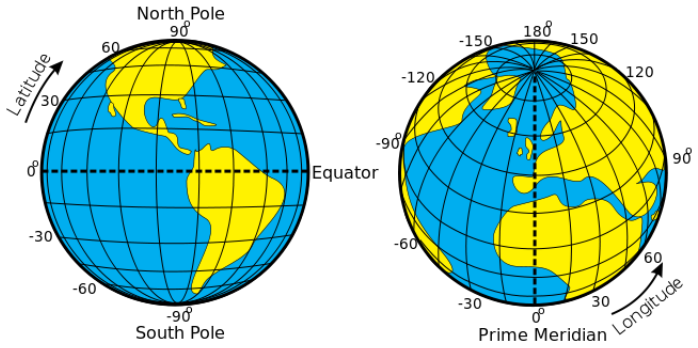

Figure 2.1: The direction of latitude and longitude angles on the surface of the Earth.

2.1.2 Distance

The distance between two locations cannot be calculated using Pythagoras' theorem due to the curvature of the Earth. The Haversine formula uses spherical trigonometry to enable the calculation of the distance between points on the surface of a sphere. The following formula can be used to calculate the distance between a pair of latitude-longitude values:

$$\begin{aligned} a &= \sin^2(\frac{\Delta\phi}{2}) + \cos(\phi_1).\cos(\phi_2).\sin^2(\frac{\Delta\lambda}{2}) \\ c &= 2.atan2(\sqrt{a}, \sqrt{1-a}) \\ distance &= R.c \end{aligned} \tag{2.1}$$

where R is the radius of the earth $\approx$ 6,371km.

2.1.3 Bearing

The bearing between two locations gives the angle between the 'forward' direction and the direction from one location to the other. The following formula can be used to calculate the bearing between a pair of latitude-longitude values:

$$\Theta = atan2(\sin(\Delta\lambda).\cos(\phi_2), \cos(\phi_1).\sin(\phi_2) - \sin(\phi_1).\cos(\phi_2).\cos(\Delta\lambda)) \tag{2.2}$$

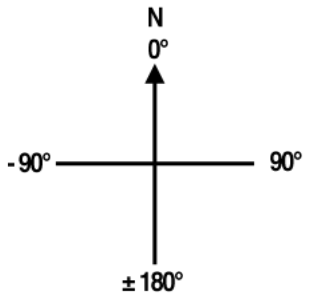

Figure 2.2: Bearings as calculated by Equation 2.2

A key property of bearings that can be seen from Figure 2.2 is that they cannot simply be negated to get the opposite direction.

2.1.4 Storage and Queries

Mobile devices generally provide an implementation of SQLite, which allows developers to store persistent information in databases. Although powerful, SQL is not designed to process locations out of the box. There has been a lot of research in the area, with a popular tool being PostGIS. This is an open-source program that adds geographic support to PostgreSQL databases. However, it would be incredibly tedious to translate a system like this to work on mobile devices. Using techniques discussed in the research by Egenhofer et al. into a new spatial query language [4], we will build a thin query system that allows us to query location based information.

2.2 Location Options

There are two options for gathering a users location on a mobile device; Global Positioning System (GPS) and network based location.

GPS requires the device to be in view of the sky so it can receive satellite signals. This results in poor performance while indoors. This being said, it is very accurate when the device is outdoors. It is typically able to pinpoint users to within 50 meters. However, depending on the users location it can take up to 15 minutes to receive the time to first fix (TTFF). The major drawback of GPS is the fact that if it is left on, it can quickly drain the battery. Preliminary experiments have shown us that tracking using GPS can use up to 9% of the battery per hour, even when the user was constantly in their home.

Cell tower triangulation uses the network cell towers the phone is communicating with, and determines the devices location by using measured response times. Because the device may only be connected to one or two towers at a time, the estimate can get very inaccurate. Android enhances these estimates using Wi-Fi based information. By keeping a list of routers and their approximate position, Google makes note of the Wi-Fi routers around you and uses their location to enhance their response [20]. The negative side of cell tower triangulation is that it requires a *persistent internet connection* to translate the cell tower response times into a latitude and longitude (even though it uses very little data).

Figure 2.3 compares the battery usage of GPS and network locations. This experiment was carried out on a 'average' (i.e. not top of the range) Android device. The device was not used for any other purpose than location tracking, and did not optimise location tracking in any way.

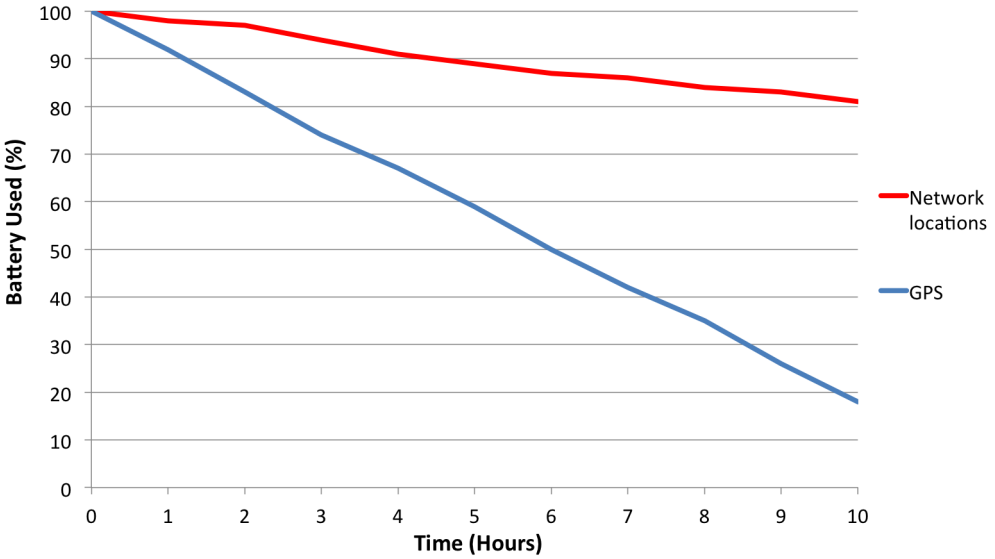

Figure 2.3: Battery usage per hour for Network Locations and GPS on a mid-range Android device.

In general, it is highly unlikely that a user leaves their GPS tracker on while they go about their daily business. In contrast, users usually have some connection to the internet, either

through 3G, 4G or WiFi. For this reason, as well as the reasons given above, our plan is to use GPS only as a 'back up', and allow the app to function as well as possible using only network location services.

2.3 Improving Location Estimates

2.3.1 Network Based Accuracy

As mentioned, network based location estimates are more uncertain than GPS based estimates. To calculate how much uncertainty there is when the mobile tries to determine location, we created a simple Android application. The application would log the estimated location every 90 seconds, and the range of uncertainty. We kept the application running for approximately 6 days and observed the following results.

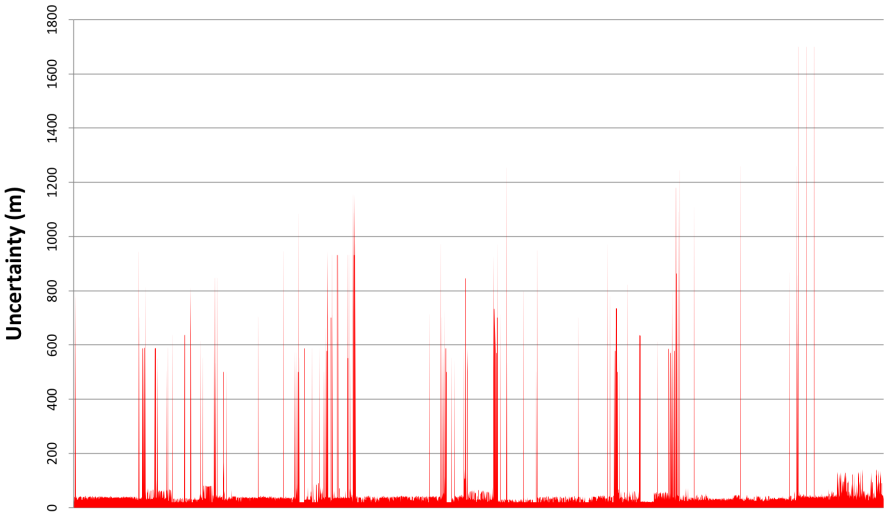

Figure 2.4: Uncertainty of location estimates gathered using network signals

As shown in Figure 4.1, the estimated location fluctuates a lot. During times when the user is stationary, such as when they are sleeping (e.g. between points 1 - 396) the estimation is relatively accurate. However, when they started moving the estimation became very erratic, with the uncertainty radius often reaching over 500 meters.

2.3.2 Basic Filtering

The Android documentation encourages developers to filter location themselves in this manner:

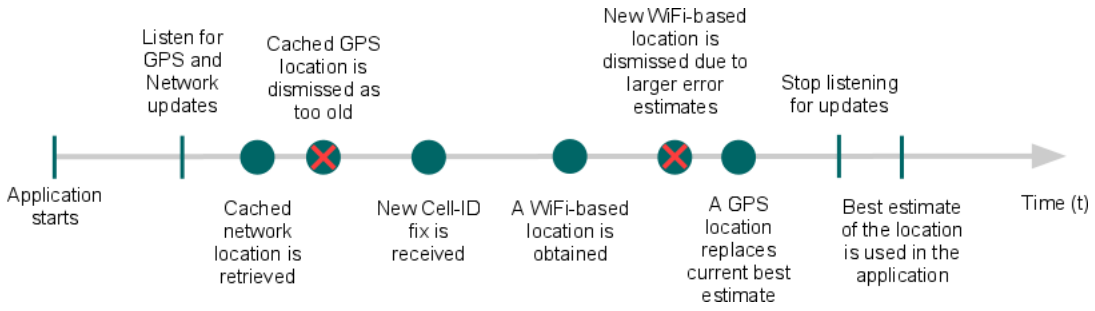

Figure 2.5: Android Location Filtering Advice [7]

This shows that a significant amount of work is required just to get a relatively accurate estimate for the users current location. This represents a small part of the large hurdle developers have to overcome to make their apps location aware. The proposed framework will incorporate techniques like this within their calls so developers do not need to worry about these details.

2.3.3 Dead Reckoning

Dead reckoning is the process of calculating a users approximate location by using a previously known position and the users speed and direction at the time. By using dead reckoning, we believe we can significantly reduce the uncertainty of a users location. This relies on the fact that users under normal circumstances are unlikely to drastically change speeds in very short periods of time (e.g. a user starting at rest is not likely to have travelled 500m in under 30 seconds).

We will be investigating whether a simple dead reckoning model can be used to reduce the uncertainty. To allow this, calculated locations will kept so that when a new location estimate is received, the last two confirmed locations will be used to calculate the average speed and direction. By using dead reckoning, an estimate of the users current location can be calculated, and compared to the actual measured location. If the uncertainty of the measured location is high, the calculated position can be used to refine the measurement.

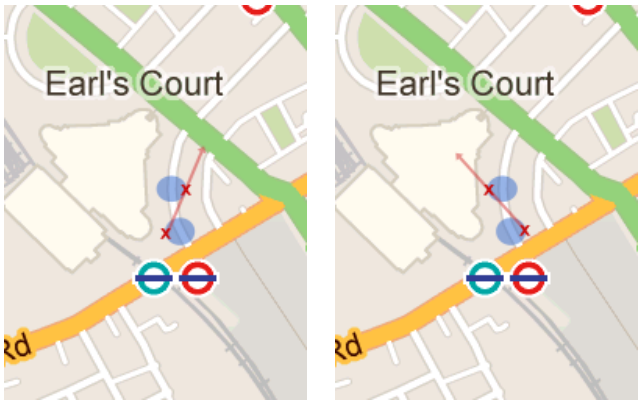

Figure 2.6: Dead reckoning used with two previously measured locations, showing how uncertainty effects direction estimates

If we look at Figure 2.6, we can see that dead reckoning can also get very complicated due to even the smallest amount of uncertainty in a reading. We can see above that the user must be following the curve of the road, but it would be hard to produce a system that can accurately evaluate this situation and give the correct position using dead reckoning alone.

2.3.4 Kalman Filter

A more advanced approach to reducing the uncertainty of the users' location estimate involves using a Kalman filter. The Kalman filter is a recursive filter that reduces the uncertainty associated with noisy data (such as our mobile location readings). By combining the dead reckoning estimate for the users location and the mobile data location reading, the Kalman filter should effectively be able to reduce the amount of uncertainty in the reading, especially reducing the huge jumps to above 500m as shown in Figure 4.1.

The Kalman filter works as follows:

$$P(S_t|S_{t-1}, \mu_t, M_t) \tag{2.3}$$

A predication for the next state S_t is calculated using the previous state S_{t-1}, the dead reckoning estimate of how much we have moved μ_t, and the mobile location reading M_t. The state is considered to consist of a position, a velocity and an acceleration. We will be using a simplified form of the Kalman filter to reduce the computational load between location updates.

We first calculate a prediction for our next state using our estimated movement and our previous state:

$$\overline{S_t} = A.S_{t-1} + B.\mu_t \tag{2.4}$$

We then estimate the variance in our state by using the covariance in the prior state (P_{t-1}), and our expected variance in the measurement (Q):

$$\overline{P_t} = A.P_{t-1}.A^T + Q \tag{2.5}$$

By using a simplified form of the Kalman filter with any readings that come through, the user's estimated position can be made a lot more accurate without causing too much of a computational hindrance.

2.4 Improving Battery Life

2.4.1 Substitution

Current mobile operating systems allow applications to chose whether they want to use GPS or network based locations. However, this decision has to be made once and then cannot be changed. Applications cannot dynamically switch location providers unless they cancel the previous one and register a new listener.

Jeongyeup Paek et al. have carried out extensive work to create a "rate-adaptive positioning system for smartphone applications" [16]. Instead of only using network locations or GPS, this system turns on the GPS receiver when it requires a more accurate reading. They say "It is based on the observation that GPS is generally less accurate in urban areas, so it suffices to turn on GPS only as often as necessary to achieve this accuracy."

Similar work was carried out by Zhenyun Zhuang et al [22]. Their work improves the efficiency of location sensing applications by (again) adapting the sensors as required. They developed a technique they call *substitution*. The difference between this work and Paek's is in the way they determine when to turn on GPS. Their system learns 'environmental characteristics' such as the availability and accuracy of providers. "The profiler monitors and stores relevant information, including current locations, visit frequency, and sensing characteristics (e.g., availability, positioning accuracy) of location providers."

2.4.2 Journeys

Another interesting technique developed by Zhuang et al., referred to as *suppression*, is to track whether the user is static or moving. They used alternate sensors, such as the accelerometer, to detect when the user is stationary. At these times, the location receiver was turned off to save battery. When the user started moving, the sensors were turned back on. This is particularly useful for apps that continuously track location, as the majority of users will spend most of their day in a single place (e.g. at home or work). By turning off receivers at these times, the efficiency of these applications can be greatly improved.

2.5 Frequent Locations

The work done by Daniel Ashbrook and Thad Starner [1] describes a system which learns a user's frequent locations given a large number of pre-recorded location readings for that user. They cluster common locations together to form areas the user is most likely to visit later e.g. their home or workplace.

They also highlight a process refined from the work by Bhattacharya et al. [3] on predicting the next location based on the current location. The process they describe uses Markov Models created for each of the users frequent locations to determine which of the other frequent locations a user may be headed towards.

While their work was done primarily on pre-recorded values (for learning), we feel this work can be adapted to work with 'live' readings too. The techniques we will be using are highlighted below.

2.5.1 Clustering

Clustering is the process of grouping entities together based on the *distance measure*. A simple way to identify which cluster a location belongs to would be to assign it to the cluster that is closest. However, the difficulty here will come in deciding:

- When to create a new cluster
- When two clusters should be merged together
- When a cluster should be split into smaller clusters, and how the original cluster should be split

2.5.2 Markov Models

A Markov model is a tool for representing probability distributions over a sequence of observations. The assumption made by these models is that a future state can be determined from the current state alone i.e. it is memoryless (known as the *Markov property*).

We will create a Markov model for each of the frequent locations we find to gain probabilistic insights into where the user may be headed. Our work will differ from the work done by Ashbrook, as they predicted the next location based on the current location alone. We will be incorporating other elements of the current state into our model, including the current time and day, and the bearing the user has travelled on so far.

2.6 Using Other Sensors

Mobile devices have the added advantage that they often come equipped with a number of other sensors. These allow developers to access a range of information about the state of the device and the environment around it. Sensor fusion is the combination of readings from a number of different sensors (each of which may have some inherent noise), and can allow for more accurate calculations. We will use these sensors in combination with our location estimates to gain a more accurate picture of what the user is doing.

2.6.1 Accelerometer

An accelerometer is a device that measures acceleration. Most smartphones come equipped with internal accelerometers, capable of measuring changes in the orientation and any

tilting motions. We felt that while location services will give us an accurate picture of the users activity, we can use other sensors such as the accelerometer to confirm and even suggest alterations to location readings as shown in [15]. For example, if the accelerometer says the user is walking very fast (or indeed running), we can adjust our estimates of their speed to reflect this. This is very important when the user initially sets off as the first readings normally do not convey the users speed very accurately.

The Android operating system provides a simple API for accessing values calculated by the internal triaxial accelerometer (which gives separate x, y and z readings). Although calculating these values requires no extra battery (as the Android system is continuously updating them), registering for change alerts on accelerometer values can cause the application to drain the battery very fast.

Step Detection

A key component to the success of this technique will be the module which aims to recognise the activity the user is carrying out. One of this simplest forms of activity recognition is to recognise when a user takes a step; a field which has been researched pretty extensively. By analysing the step rate, we can get a fairly accurate measurement of the users speed.

The method we will conduct to detect steps is an amalgamation of techniques discussed in [13] and [21]:

1. Calculate the magnitude of accelerometer reading values as
$$r = \sqrt{x^2 + y^2 + z^2} - 9.8 \tag{2.6}$$
where 9.8 is subtracted to remove the effects of gravity.

2. Pass this value through a low pass filter, which would ideally result in a graph where every local maximum corresponds to a footfall. We use a low pass filter as it reduces the impact of high-frequency signals as steps would cause peaks in the low frequencies not the high frequencies (which are basically noise). A possible solution to this would be to use a FIR filter. However, this would not be very useful for detecting multiple activities as it requires a 'cut-off' frequency, and we cannot define a frequency that will work for both running (which needs a high 'cut-off') and walking (which needs a lower 'cut-off'). We have chosen a simple low pass filter which is more practical for mobile phones as it is real-time and not very computationally expensive:
$$rc = 0.5 \quad dt = \frac{1}{25} \quad \alpha = \frac{dt}{rc + dt}$$
$$x' = \alpha * x + (1 - \alpha) * x' \tag{2.7}$$
where x' represents the smoothed value and x is the raw input.

3. Generate a template which represents a basic step, and use template matching with the results to determine.

4. Use a pseudo-derivative to measure exactly when a step takes place:
$$y(n) = \frac{1}{8}[2x(n) + x(n-1) + x(n-3) + 2x(n-4)] \tag{2.8}$$

Note: we can effectively remove the leading fraction because we will not be using these values themselves, just looking for whether they are positive or negative.

After the above, we should have a set of values where the 0-crossings represent footfalls.

Activity Detection

Activity detection is the process of determining what kind of activity the user is carrying out. Allowing developers to be aware of the users activity fits into the bigger picture of context awareness. An investigation by Ravi et al [18] attempted to solve recognising user activity as a classification problem. Their experiment aimed to classify a wide range of activities from walking to sit ups and the results were very positive. Due to the different requirements of this investigation, we will only be analysing walking and running. While their experiment analysed the mean, standard deviation, energy and correlation of moving windows of readings, we will be focusing on the first two. Their experiment also tested a number of methods for classification. We will be focusing on a k-nearest neighbour algorithm which was proved to work well in their experiment.

2.6.2 WiFi

One of the main sources for data on mobile phones is WiFi, as often users will have limits on how much 3G data they are allowed to consume. For this reason, a mobile device's WiFi receiver is usually kept on for most of the day. A big area of research recently has been trying to use WiFi information to determine location (primarily indoors). Whilst a lot of researchers have found success in this field, we felt it would be overkill to implement a whole WiFi based location system for this investigation. However, to completely disregard WiFi information would be wrong. Inspired by the work done by Rekimoto et al. [19], we plan on using WiFi information as a small subsidiary component of our investigation. This is further discussed in section 4.

3 | API Design

This chapter details the decisions made with regard to the API available to developers. The first decision to be made will be the choice of operating system we develop the framework for. We will choose the platform we feel would provide the best measure of 'initial response'. If this response proves positive, we can extend the system to other platforms in the future. We will analyse current deficiencies and consult developers to gather an awareness of what features would be most beneficial to developers.

3.1 Comparison of Platforms

One the first decisions to be made is the choice of operating system for our solution. We will analyse the most popular systems to understand what is currently available to developers. For this investigation, we concentrated on the two most popular mobile platforms: Google's Android and Apple's iOS.

3.1.1 Android

Android currently has approximately 75% of the market share for smartphones [17]. This is because there is a large range of Android devices, allowing many poorer countries access to cheaper models. This results in the major difficulty of developing for Android; the fragmented nature of the Android market means there are many corner cases which are difficult to accurately test for. However, Google is very active in the developer community, meaning there are lots of resources available to Android developers.

Location Services

Android provides the LocationManager class to give developers access to location services. Taken directly from the Android developer guides [6], the current location services allow developers to:

- Query for the list of all LocationProviders for the last known user location.
- Register/unregister for periodic updates of the user's current location from a location provider (specified either by criteria or name).
- Register/unregister for a given Intent to be fired if the device comes within a given proximity (specified by radius in meters) of a given lat/long.

Point 3 above is a powerful and popular feature called *geofencing*. However, developers have been very unhappy with its performance due to heavy battery use and inaccuracy, as expressed by Greg Milette and Adam Stroud in their book [14]. They explain "Although Android's default proximity alert implementation may be simple to use, it can be costly in terms of battery life. [...] Android sets up a LocationListener for every proximity alert that is set. This means that (each) proximity alert will consume large amounts of battery power because the device will receive location updates frequently."

As described, the proximity alert system uses a very naïvealgorithm. It simply checks through the entire list of points of interest for every location update. This results in very poor battery performance and is the reason a lot of developers have chosen to implement their own systems for this kind feature.

Permissions

Android's permission system involves apps declaring their permissions when submitting to the Google Play Store. Users can view permissions required by the app when installing. However, there is no way for users to accept some permissions and deny others; they can either install the app or not. With regards to location specifically, users also have the option to allow all apps to use location or allow no apps to use location, as highlighted by the Figure 3.1.

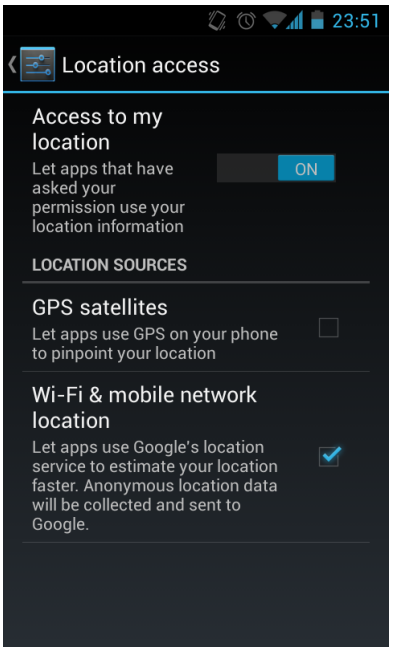

Figure 3.1: Android option for users to allow or disallow all apps to use location services

3.1.2 iOS

Although Apple's iOS has a substantially lower market share than Android, it is still the platform of choice for a majority of developers. iOS is seen as a 'safer' route to generate revenue [12].

Location Services

Taken directly from Apple's location awareness programming guide [5], iOS provides the following functionality for developers:

- The significant-change location service provides a low-power way to get the current location and be notified of changes to that location. (iOS 4.0 and later).
- The standard location service offers a more configurable way to get the current location.
- Region monitoring lets you monitor boundary crossings for a defined area. (iOS 4.0 and later on devices that support region monitoring).

Permissions

In contrast to Android, iOS does not have the same concept of permissions. Instead, Apple manually looks through each application upon submission to judge whether user information is being misused. Once approved, an application can do anything it likes using Apple's documented APIs. With regards to location, however, iOS provides the user with a little more. Users are permitted to allow or disallow *individual applications* from using location services, as shown in Figure 3.2.

Figure 3.2: iOS option for users to allow or disallow specific apps from using location services

3.2 Android

After the analysis of the current state of the two platforms, we decided to target Android. This decision was made for a number of reasons:

- Android higher market share and weaker permissions system (for location services) provides a larger scope for improvement.
- Android's developer community have been pretty vocal about their dissatisfaction with the current API, so getting developers to use the framework produced should be easier.
- Android is open-source [8], making it easier to work with as we can always go into the source code and see why things work the way they do.
- Finally, in comparison to other operating systems, such as Apple's iOS, Android allows more freedom when it comes to features such as background services and inter-app communication.

3.2.1 Services

The framework will be designed to allow any number of clients to access the gathered location information. One of the key benefits this system will provide is that by using a centralised model, location calculations can be done by just one app rather than each of the clients doing the same work. Unfortunately, this is not provided by Android as an out-of-the-box solution, meaning a custom solution will have to be implemented. Android does allow applications to share data, as long as both applications agree.

Our plan for this project is to come up with a custom solution which takes advantage of Android's *services*. An Android service is an application component that is designed to perform long running operations (in the background) [9]. Services can be used as continually running processes on the device to perform indefinite tasks, e.g. tracking location. One problem which will need to be addressed is that the Android system can stop a service at any time if it requires memory to be freed. The service will be stopped without any signal or sign foretelling the developer what is about to happen. This is unavoidable, but there is an API which informs the system that the service should be restarted as soon as possible. For this reason, the development style of services needs to change to ensure that at no point is any information held which has not been committed to memory. This way, if the service is killed, the newly created service can 'resume' form the state saved by the previous service.

Services can be bound to by Android activities, the foreground components which display the user interface. By binding to the service, an activity can interact with and query the service. For our framework, we would like a service that does not belong to any single app, and so can be bound to by an activity of any client. As mentioned, this is not permitted in Android, and so a proprietary solution will need to be developed to ensure to same service can be accessed securely by any client authorised to use it.

3.3 Developer Research

To decide on the feature set and to deduce what developers would be looking for in a framework like ours, we consulted a sample group of Android developers. The participants had a range of experience in developing location aware applications; from simple applications that only require the current location to one application which tracked users to learn their common underground stations.

3.3.1 Reasons for use

When questioned as to whether they would be willing to use a third party library in their application, all the developers responded that they already use third party libraries. When asked whether they would develop an application that communicates with another application, a few of them expressed concern. The primary concern was, understandably, what they would do if this application wasn't installed. We eventually agreed that in effect they could implement their apps functionality (for developers who came with ideas) and use Lokey for extra functionality that would enhance the user experience rather than embody it.

Another interesting piece of feedback we got was that developers liked that the user would be able to control the clients' access to information. More than half to the participants used Android devices themselves, but only one said he rigorously checked applications permissions before installing. The others simply said they wouldn't install an app that looked 'dodgy'. Of the participants, a majority stated that they had received negative feedback on the Play Store due to a misunderstanding about what the app was doing with user's data. They commented that there was no official way to provide reasons for requesting certain resources. The de facto practice has now become to define these reasons in the description when the app is uploaded. However, most users do not read this description and are quick to complain when they feel any distrust. They concluded that it would be helpful to allow users to dynamically control of what permissions are given.

3.3.2 Requested features

Due to the time constraints of the project, a small number of features had to be selected for implementation. We asked participants for any ideas they were willing to share, and then suggested a few features we had previously considered. We also asked them for any suggestions as to what they would do with the feature if it was implemented.

Geofencing

By far the most requested feature was geofencing; that is setting up virtual perimeters and alerting an app when the user enters or exits this area. The majority of developers declared that they would be interesting in adopting a library that efficiently implemented geofencing. They said they could easily think of numerous applications for this kind of

technology, with examples including a 'friend alerter', a 'profile changer' and an alternate system for workers to 'clock in' to work places.

We asked the developers why they felt this was a difficult task and why they did not implement it using Android's current solution. It was interesting to note that more than half of the participants did not know this feature even existed. Out of the ones who had heard of it, none had used it due to the "terrible battery problems" that came with it. One developer had implemented his own solution for geofencing, which exactly did what the current Android implementation did, except that it only used one Location provider for all geofences. He expressed that this significantly improved the battery life compared to the current Android solution. When probed further, the developer said that since he had quite a few geofences, the app would constantly cycle through them checking if any had been entered. This led to his app getting very poor reviews and the feature was subsequently removed. A number of developers felt they could have developed a system like this, but would be worried about the battery usage. The rest of the participants felt they would not know how to accurately determine if the user had entered a small area using network based location (which they had all used at some point in the past).

Start/Stop Moving

Another commonly requested feature is to be alerted when the user sets off from a location. This was suggested for applications that can perform tasks such as: check the train stations around the user for delays or warn the user about traffic near them (when they leave a location).

Frequent Locations

A few of the participants suggested they could see use for being able to access a list of the users frequent locations. Suggestions for actual uses include an app that can tell you where the cheapest petrol station is around any of your frequent locations.

3.4 API

A key component to the success of this framework will be how appealing it is to developers. Given the results in section 3.3, we have gathered that there is a range of functionality developers would require in a product like Lokey. We have chosen the features we felt were most practical and proposed the biggest challenges in terms of gathering information. This section describes an overview of the features we have decided to implement, and the API developers will use to access these features.

Tracking Location

One of the core components of the framework will be to receive the location updates and filter them to maintain an accurate estimate of the users location. We will make these

filtered values available to developers in the same way as Android; they can ask for the last known location and they can request continuous updates.

getLastKnownLocation() Returns the last known location of the user. If the user is currently on a journey, it returns the last waypoint. If the user is stationary, it returns the filtered stationary location. The locations returned here will be Lokey's filtered values, so will be more precise than the raw network readings.

startTrackingLocation(listener) Registers the client to be alerted about any future changes of the users location. The client will be informed as long as they are 'awake' i.e. they haven't been sent to the background by the user.

stopTrackingLocation(listener) Unregisters a client registered using the above call.

Tracking Journeys

Tracking journeys has a number of advantages. Not only will we understand more about the user and their pattern of movements, we will be able to change certain aspects of the system based on whether the user is currently on a journey or not. For example, we will be able to reduce the rate of location updates while the user is not on a journey so that battery can be conserved.

isCurrentlyOnJourney() Returns a value indicating whether the user is currently travelling to a new location.

getCurrentJourneyDetails() Returns details about the current journey, or null if the user is currently not travelling. This can be used by clients to quickly gain knowledge about the users current state, i.e. a client can immediately establish an awareness of where the user is going as soon as it is opened.

registerJourneyListener(listener) Registers the listener passed in as a parameter to be notified whenever the user sets off from a stationary location and whenever the users current journey comes to an end. In the latter case, the user will also be provided with a summary of the journey.

unregisterJourneyListener(listener) Unregisters a listener registered using the above call.

getCommonRoutes(latitude, longitude, radius) Returns a list of routes the user has frequently taken. A location parameter will need to be provided to stop any single call from accessing too much data at the same time. This is a problem as it may block Lokey from serving requests from other clients.

The framework will be designed to be object oriented, so calls that return journey details will return an instance of the following class:

Journey Details	
start latitude	double
start longitude	double
end latitude	double
end longitude	double
times taken	integer
average time	long
average speed	float
last time	long
waypoints	List

Figure 3.3: The class used to encapsulate a common journey.

Tracking Frequent Locations

Users are likely to have certain locations they frequently visit, such as their home, their workplace or their favourite restaurant. Since the framework is built to constantly track the users location, it would be useful to understand where the user spends their time. This would allow developers to use these locations to pre-compute factors of their experience so users have the information waiting for them. The calculated frequent locations will be available to developers using the following call:

getFrequentLocations(latitude, longitude, radius) Returns a list of frequent locations the user has visited. To curb the computational impact of malicious certain calls, a location parameter will need to be provided, and only frequent locations near this location will be returned. The range of frequent locations will also be passed as a parameter, but will have a maximum value of 20km.

This call will return a list of objects of the following type:

FrequentLocation	
latitude	double
longitude	double
radius	float
last visit	long
total time spent	long
number of visits	integer

Figure 3.4: The class used to encapsulate a common journey.

Predicting Destinations

As we are tracking the users frequent journeys, we believe this can be built into a solution for predicting the destination a user is headed towards when they leave a location. We

will use the techniques discussed in section 2.5 to build a Markov model that represents the probabilities of destinations. Developers will have access to this using the following call:

getPredictedDestinations() Returns an ordered list of destinations Lokey has analysed the user to be travelling towards. The locations returned will be locations frequently visited by the user, and will only be included in the list if there is a *strong* probability that they are the intended destination.

A key design decision to be made was how many destinations to return. We decided we will return a list of predictions up to some limit. This means any destination that has a probability above a certain threshold will be returned. This threshold will be determined based on a case-by-case basis due to the fact that the probabilities may all be very low or very high. Returned destinations will be an instantiation of the FrequentLocation object described above.

Activity Detection

As discussed in section 2.6.1, using the accelerometer to track user movement provides another measure of speed. These readings will be used to help confirm when a journey starts/stops. However, since they will be implemented, we feel it would be a shame not to allow developers to make use of these modules as well. We will expose the following function to developers:

getCurrentActivity(duration) Analyses the user activity using the accelerometer for a time period passed as a parameter. The minimum value for the duration is 5 seconds. The caller will then receive a callback containing the activity detected by Lokey.

The returned value will be a member of an enumerated type. Possible values will include: *walking, running* and *driving.*

Geofencing

Due to the particular point being made by a lot of developers that the one more advanced location feature in Android doesn't work, we decided to attempt to improve this feature. We believe that by leveraging the information gathered form the features mentioned above, we will be able to implement a geofencing solution that is far more battery conscious than Android's solution. We will allow developers to register points of interest in the same way as the current implementation to enable a smooth transition to our system:

registerPointOfInterest(id, latitude, longitude, radius) Registers a point of interest, which will cause the client to be alerted whenever the user enters or exits an area. The location and circular radius must be provided as parameters, as well as a unique identifier to allow clients to know which point of interest has been triggered.

unregisterPointOfInterest(id) Unregisters a point of interest (identified by the id passed as a parameter) registered using the above call.

3.5 Summary

We have chosen to develop our framework for the Android mobile platform. While both Android and iOS provide very few options to users in terms of controlling permissions, iOS is slightly ahead of Android in that it allows users to block an individual application from using location services. Thus, by developing the framework for Android, we can make greater improvements on the currently available system.

To choose the API exposed to developers, we analysed the current features available in Android. We also consulted developers on their opinions. We settled on six main features that we feel will allow developers to access sufficiently detailed information about the users location without having to do too much work themselves.

4 | System Architecture

This chapter discusses the architecture of the proposed framework. We will cover the components involved, as well as any design decision made during the structuring process. We will also detail how clients will access Lokey's API, as this is not a common feature in android.

4.1 Accessing Lokey's API

A key architecture decision to be made is how to address the problem highlighted in section 3.2.1; that is, how to allow any number of clients to have access to Lokey's functionality. Essentially we want a single service to be running on the end users device, which carries out all the location calculations. This service should then be open to queries by any client application. As mentioned, this is not standard Android behaviour and so are required to build a custom solution. The two options we have identified are:

1. Each client keep its own copy of a 'LokeyService'. At any time only one client should have started their version of this service, and all other clients on the device should be able to communicate with this.
2. Have a standalone Lokey app which maintains its own running copy of the LokeyService. Clients will communicate with their own version of a provided 'LokeyClientService', which will in turn communicate with the single instance of the LokeyService on the device.

Option 1 would work as follows. When a client is started for the first time, it will check if any another client on this device has started its own version of this service. If another client has, the new client will bind to the existing service instead of starting its own.

There are a number of disadvantages associated with this option:

- Control of the service is handled by one of the clients. This means if the user kills the client, the LokeyService which everybody relies on will also go down.
- If the client in control of the service is deleted, the other applications relying on the service will have to start one service amongst themselves, using a form of distributed leader election.The killed service would also have to pass over any information it has gathered when it is removed from the device.
- Clients would all have to be allowed to communicate with each other, which meant forcing each one to specify this in the manifest.

Option 2 is more appealing as it will provide a standalone application, form which users can change permissions, view their data etc. It also means that control of the running service will be handled by the Lokey application rather than any one of the clients. This way, a user will know the repercussions of their actions if they were to kill Lokey's service.

This implementation does have the disadvantage that users will have to ensure Lokey is installed on their device before they can use any client. We will try to mitigate this by providing developers with easy checks to see if Lokey is installed and a dialog to show users that will explain why and how to install Lokey.

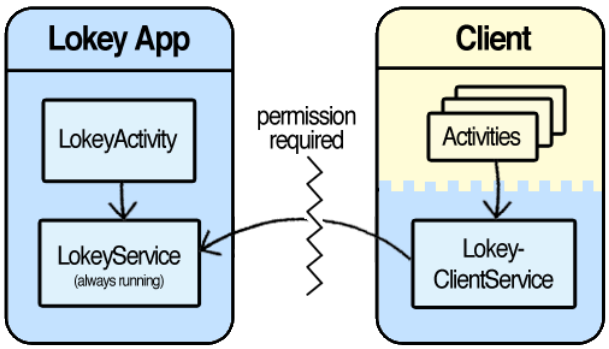

Figure 4.1: Overview of main app and client components

4.2 Lokey Service

The LokeyService (hereby referred to as the *main service*) is where the majority of this project will be focussed. This service is where the location data will be calculated and saved, and where clients queries will eventually be logged, evaluated and satisfied. The main things we will take into consideration when implementing this component are:

- Using as little battery as possible.
- Ensuring the service is always running, i.e. it starts up when the phone is turned on or rebooted, and restarts as soon as possible if it is killed or crashes.
- Ensuring the service stays running for as long as possible. As mentioned, the environment may choose to kill a service at any time to reclaim memory. The chances of getting killed are significantly reduced by ensuring the service uses as little memory as possible.
- Ensuring the service never looses any information. It will be necessary to make sure the service never holds anything in local memory that has not been committed to a persistent store.

4.2.1 Modules

Due to the nature of the features to be implemented, we have chosen to implement our system using *modular programming*. This means responsibility for each distinct part of

the functionality will be managed by a single component. The key reason for using this modular approach is extensibility. There will be a few core modules which are used by other 'feature modules'. The design of this system should allow for new features (and therefore new modules) to be easily linked to existing functionality.

The **location tracker** module is the main focus of this investigation. It will be responsible for filtering location updates to maintain a constant awareness of the users location. This class will therefore also manage the lifecycle of journeys. It will use the *movement tracker* and the *wifi tracker* to help confirm when a journey has started and when it has ended. Once completed, journeys will be passed to the *journey saver*. This module will also inform all interested clients about journey start / stops.

The **journey saver** module will efficiently save journey information. It will use the *frequent location saver* to save the start and end points, and will save the waypoints with these references into the journeys table.

The **frequent location saver** module will group together and save locations that the user frequently visits. This component will respond to alerts made by *location tracker*, so will not need to be continuously running. This module will make use of the *wifi tracker* to make it easier to identify locations that essentially represent the same place (even if the latitude/longitude values are different). By saving the SSID of any wifi connected to at the location, we will be able to identify similar locations more easily.

The **destination predictor** module uses previous journeys to make heuristic predictions about the destination the user is trying to reach. It is the job of this module to maintain the Markov model created for predicting the users destination. When a journey ends, the destination predictor will analyse the journey to make decisions about what should be updated. This module will only respond to queries, so does not need to be constantly running.

The **wifi tracker** module will monitor changes in the state of the device's wifi connection. The module will save the time at which connection changes occur. We chose not to save this information in a database as it is transient and will not be important once the connection has changed. However, it is saved to another form of persistent memory, the user preferences, to stop any information being lost in case the service is killed.

The **movement tracker** is used to estimate the current activity of the user. By analysing the changes in accelerometer readings, the movement tracker will estimate the users current activity and (if applicable) the current speed. As mentioned in section 2.6.1, constantly monitoring the accelerometer can quickly deplete the battery. Therefore, this component will have to explicitly be turned on for a certain period of time (to be specified) by any component which wishes to know the current activity.

The **geofencing tracker** module manages any points of interest registered by clients. It will use location updates by the *location tracker* to calculate whether a point of interest has been entered/exited. A key focus of this investigation will be to use this module as efficiently as possible. Instead of loading all points of interest (from the database) and checking through each one at every update, we will implement a system that only loads points that look relevant. For this reason, this module will use the previous routes and the predicted destinations to understand which points of interest to load.

Figure 4.2 shows an overview of when the modules communicate.

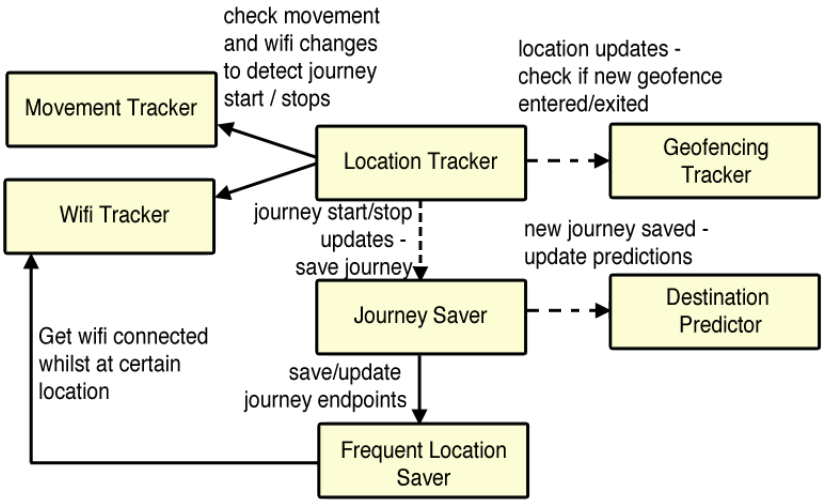

Figure 4.2: Communication between modules. Dotted lines indicate asynchronous updates while black lines indicate direct function calls.

4.2.2 Data Storage

As the service stays on for increasing amounts of time, we anticipate certain components to have produced large amounts of data. This means there will be a significant amount of data (up to megabytes) saved by the modules mentioned above. Android exposes an API to allow applications to create and query SQLite databases.

The database maintained by the main service is shown in Figure 4.3. A summary of the purpose of each table is given below:

Client Table This will hold an entry for every client that accesses Lokey, along with the permissions that have been granted. As SQLite supports bitwise operators, we use a single field to store all the permissions. When applications register for the first time, they are able to provide reasons for requiring certain features. We save these as an array of strings in the client table.

Log Table This will store all the calls made by a certain client, identified by its package. The database will create a new entry for every new call, so will have to be 'reduced' at certain time intervals in an effort to summarise results over long periods of time.

Point Of Interest Table This will hold any points of interest that have been registered by clients. Points of interest will be uniquely identified by the package of the client and an id provided by the client.

Frequent Location Table This database will store frequent locations along with an indication of the frequency of visit and time spent in each location. The database should be structured in such a way that queries return a list locations that are both spatially and temporally accurate. The locations are also stored with an associated router SSID. We chose to use SSID instead of MAC address because in many cases

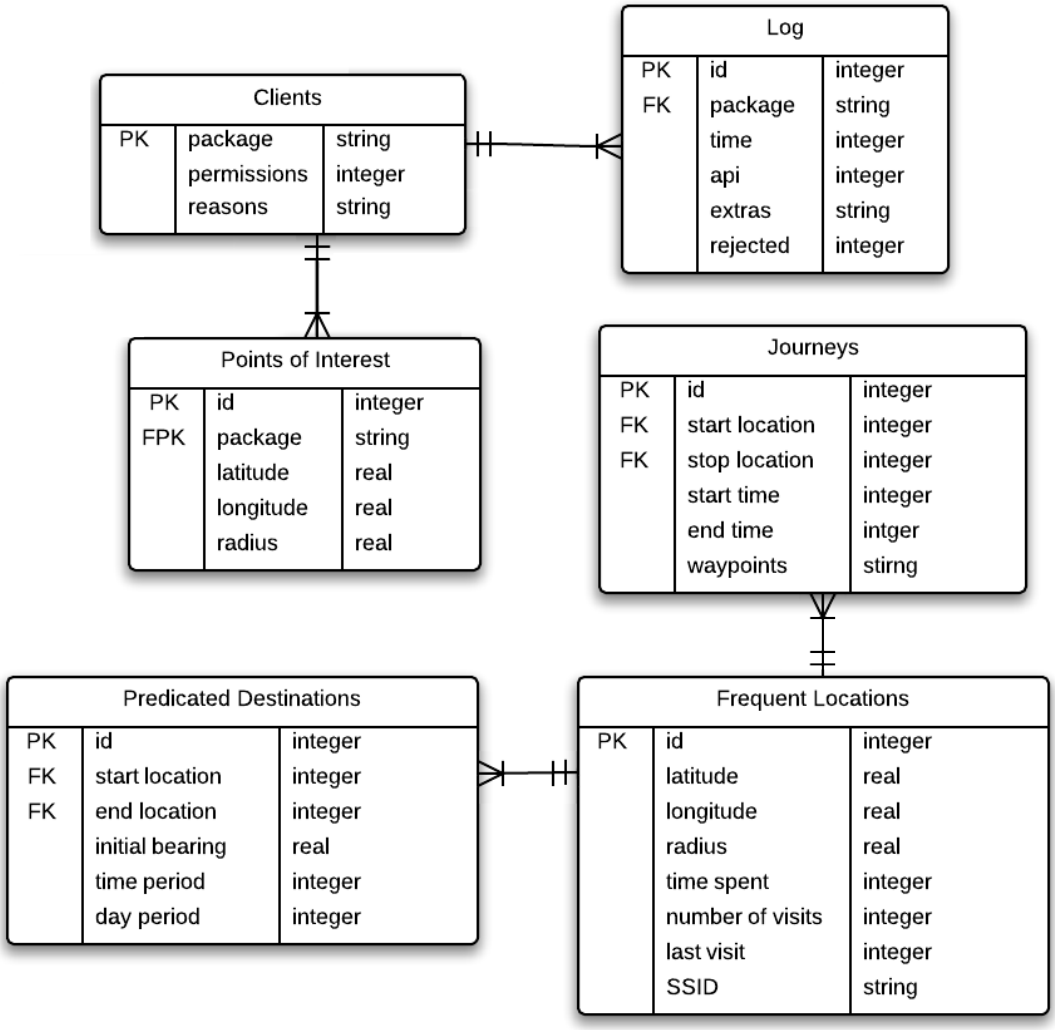

Figure 4.3: Proposed database structure

MAC addresses will be different when the SSID indicates the location is the same, for example at university.

Journey Table This will save journeys the user has taken. It will store the start location and time, end location and time, along with a list of all the waypoints encountered. The waypoints are stored as a string because it would be inefficient to create a new table to hold all the waypoints and their associations (especially because it is highly unlikely two exact same waypoints are ever observed).

Predicted Destinations Table This database will store the Markov models for each of the frequent locations. Each database entry will hold the probability of the destination being a certain location, given the initial location, the time and the initial bearing.

4.2.3 Permissions

As well as the calculation and storage of the information mentioned above, the main service will also have to control access to this information. To do this, we will implement a custom inter-app permissions system. This will allow end users to grant and revoke permissions to specific actions by individual clients, through the Lokey user interface. Clients can then be stopped from accessing certain information if the user so wishes.

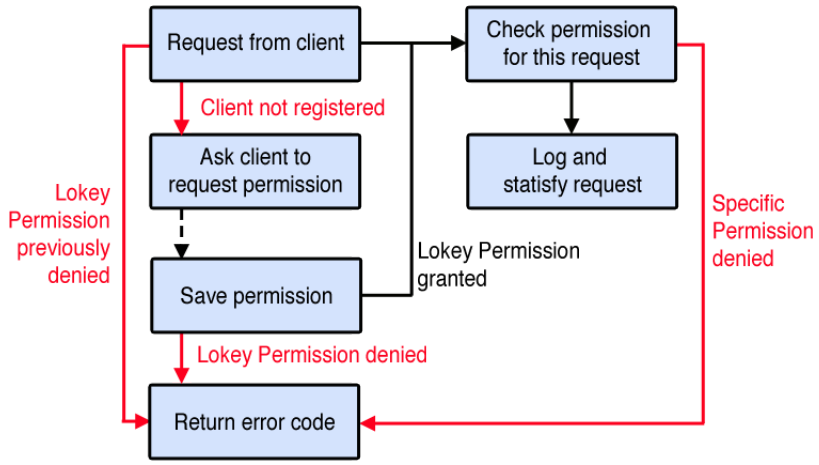

Figure 4.4: Overview of the actions carried out by the main service when a client makes a request

The following permissions will be enforced:

Lokey Manifest Permission This will be a fundamental requirement for any client as it will be necessary to allow communication with Lokey. This permission will have to be declared in the Manifest of the client, a process which is explored further in the User Guide (Appendix A).

Lokey Permission This permission will be controlled by the user, and will determine if the client can access any information on Lokey, i.e. it is a quick way for the user to deny a client any access to his location information. This permission will initially be determined when the client is opened for the first time and the user is presented with a dialog asking to grant the client permission to access Lokey (see Appendix A). If this permission is denied or revoked at a later time, any of the following permissions will be rendered useless and will not even be checked.

Current Location Permission Required if the client wants to access the current location based on calculations made by the LocationTracker.

Journey Details Permission Required if the client wishes to access details about the current journey or if they wish to receive notifications about journey starts/stops.

Track Location Permission Required if the client wishes to receive notifications from the LocationTracker whenever it updates its estimate for current location.

Geofencing Permission Required if the client wishes to register a point of interest.

View Routes Permission Required if the client wishes to see previous journeys.

Predicted Destinations Permission Required if the client wishes to know where Lokey estimates the user is trying to get to.

Frequent Locations Permission Required for a client to access locations this user most frequently visits.

Activity Permission Required if the client wishes to use the results obtained by the MovementTracker to estimate what activity the user is carrying out.

4.3 Lokey Client Service

The LokeyClientService (referred to as the *client service*) will be bundled into each client application, and will enable clients to access Lokey's functionality. The size of this bundle will be important as an app's size has been shown to effect an end-users decision to download/delete the app. We do not want clients to be impacted too much by the addition of our library, so will need to make it as small as possible.

This service will have three core responsibilities:

1. Maintain a 'binding' to the main service, establishing where to send messages.
2. Bundle requests and their parameters in the correct format, ready to be interpreted by the main service.
3. Unpack replies sent by the main service and forward them on to the client application.

The client service will also be in charge of asking the user to grant the client the 'Lokey Permission' when the application is first started. It will also need to alert the user if the device does not have Lokey installed. Both of these will be implemented in the form of dialogs which will inform the user what they can do to ensure the client works correctly. These processes are highlighted in Figure 4.5.

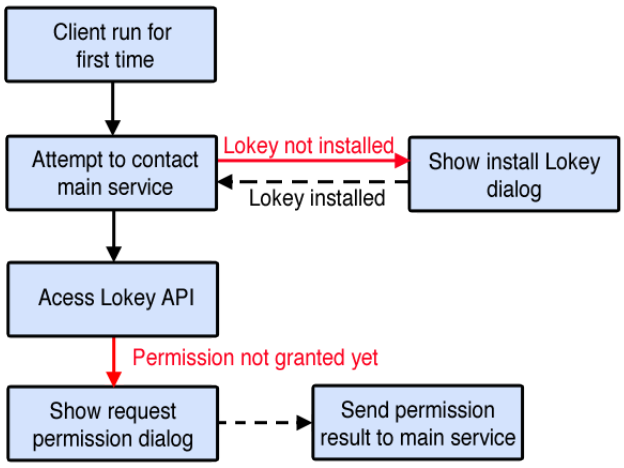

Figure 4.5: Actions carried out by the client service when used for the first time

4.4 Service Communication

On Android, one application cannot access the memory of another. For applications to share information they must communicate in some way. This was one of the more technical decisions we had to make during design of this framework; how to enable secure communication between the LokeyService and any LokeyClientService (which will be running on two separate processes due to the nature of Android applications). There were two main options we could have used to implement this interprocess communication:

1. The **Android Interface Definition Language (AIDL)** can be used to define the programming interface that both the client and main service agree upon in order to communicate. This requires the client service to bind to the main service and then allows direct invoking of methods. This method would allow multiple client services to bind to the main service at the same time.
2. Android allows **message passing** between processes using a Messenger. This is a more primitive method, as message formats will have to be manually agreed upon beforehand.

We decided to go with option 2 as it is the simpler method. It is applicable here because there should never be a situation where two client services are communicating with the main service. This is because communication should only occur when the client is in the foreground (i.e. the user is actively using the client). Using option 2 also makes updating the APIs simpler as we can add new messages which can be ignored by older versions of the app. In contrast, the AIDL will not be compatible if Lokey updates and clients do not.

4.5 Summary

In this chapter, we have described the architecture of the proposed system. Lokey will be a downloadable, standalone application. From this application, users will be able to access their information, change clients permissions, and monitor client usage. This application will maintain a service that continually tracks the users location and other interesting information such as frequent locations.

We then covered the core components of this service. We employ a modular design, whereby components are each in charge of one aspect of Lokey's functionality. The data stored will be in a SQLite relational database. The service will also manage permissions and access to APIs by clients.

Developers will be provided with a small library (the LokeyClientLibrary) which they must include in their applications. This library will contain a service that automatically binds to Lokey, and will enable communication. We have chosen to implement a message passing system due to its ease of development and its extensibility.

5 | Implementation

This chapter will explain the implementation of the proposed framework. We will describe any noteworthy work and techniques we have developed through the course of this investigation.

5.1 Journeys

A key component of the work done by the *location tracker* is to separate location readings of users into times that the user is on the move and detecting when the user is staying in a fixed place. By maintaining a sense of whether the user is on the move or stationary, we can refine location estimates as highlighted in the next section. Accurately knowing if the user is currently moving or not also allows us to refine the implementations in the rest of this chapter to use as little battery as possible. For example, the update frequency can be significantly reduced when we are sure the user is stationary and only needs to be increased when we know they have set off.

However, detection of whether the user is moving or stationary is not a trivial task amidst noisy location updates.

5.1.1 Detecting Starts

When the user starts moving from their previous location, we start a new journey. Detection of movement from a stationary location is complicated by the fact that arbitrary updates may incorrectly place the user some distance away from their current location.

Maitaining the best estimate for current location

Accurately detecting true initial movements depends largely on having the best estimate for the users current location. In areas of bad signal, it may be the case that every reading is highly inaccurate. To combat this, we use a simplified version of the Kalman filter technique to maintain an accurate estimate of the actual location:

$$\mu_{i+1} = \frac{\frac{1}{T_i^2}\,\mu_0 + \frac{1}{\sigma_i^2}\,y}{\frac{1}{T_{i+1}^2}} \tag{5.1}$$

$$\frac{1}{T_{i+1}^2} = \frac{1}{T_i^2} + \frac{1}{\sigma_i^2} \tag{5.2}$$

where μ is the current estimate, T is the variance of the current estimate, y is a new reading and σ is the measured variance.

This equation outputs what is essentially a smoothed version of the readings, with all readings *squashed* towards the estimated stationary location.

Detecting set offs

By maintaining an accurate estimate of the users current location, detecting deviations becomes a lot easier. To detect if a user has set off from their stationary location, we check if the new reading clearly differs from the stationary location. The difficulty comes from the fact that even though the user is moving, it does not necessarily mean they have started a journey. For example, at university a student may frequently move between classes or rooms, but they have not actually set off on a new journey.

To check this accurately, we go through all readings obtained at the stationary location. We analyse any readings that have a significant variance from the mean (but were still classed as part of the stationary location) and use these to determine if the new location indicates a set off.

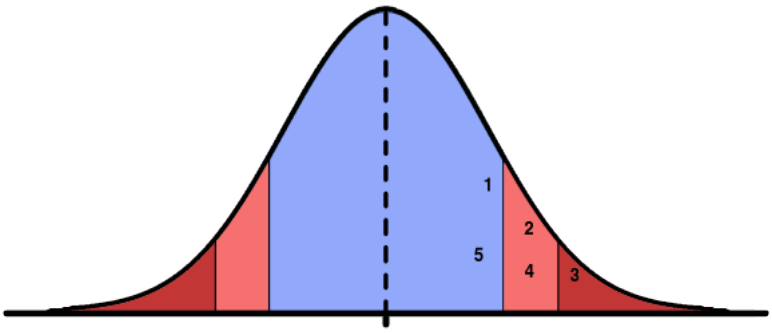

Figure 5.1: An example sequence of the distances of readings from the estimated mean while in a stationary location. The pink area indicates points which are considered outliers but are kept, and the dark red area indicates points which are judged to either be too inaccurate or start a new journey.

From diagram 5.1 we can see an example of what our detection algorithm aims to achieve. If the reading at 3 was witnessed on its own, it would either be judged to be the start of a new journey or a random outlier (based on the accuracy of the reading). However, its

predecessors form a path of readings, each more distant from the mean, which seems to indicate the user is travelling within a small area. For this reason 3 is accepted as a value for the stationary location. If points 4 and 5 actually lay further it would be apparent that in fact a journey has started. In this example we can see that points 4 and 5 get closer to the mean which means point 3 was definitely part of the stationary location.

Missed Journey Points

The technique discussed for detecting journey starts is very strict, i.e. it may often only detect a journey has started after wrongly saying a few of the updates belonged to the stationary location. For this reason, it is important that we save previous locations which deviate from the current stationary estimate by a significant amount. When a journey start is detected, we then go through these saved readings (in reverse chronological order) and judge wether these readings should actually be part of the journey.

To determine if a specific reading should be part of the current journey, we use the following algorithm:

```
fullDist = disntaceBetween(stationaryLocation, currentLocation)
reversefor (reading in savedReadings):
    if (reading.time() > lastTimeAtAverage)
        dist1 = disntaceBetween(stationaryLocation, reading)
        dist2 = disntaceBetween(currentLocation, reading)

        if (dist1 + dist2 < 1.2 * fullDist)
            journey.add(reading)
```

The final if statement effectively creates an ellipse which has its foci at the stationary location and the location that started the journey. By creating an ellipse focused at these points, we will only be looking for potential locations that could have laid on the route to the current location. The value 1.2 was chosen after conducting a range of tests to determine which value produced the best results. This process is shown pictorially in Figure 5.3.

5.1.2 Detecting Termination

Detecting journey termination uses similar methods. Essentially we want to confirm that the user hasn't moved in a while to be sure they have reached their destination. There is a caveat here that often somebody may stop over at an auxiliary destination for a short amount of time (for example to pick something up) on the way to their main destination.

The idea for the algorithm we use to detect termination is to go through all previous locations in reverse chronological order in the last 15 minutes, and check if we have moved at least 50 meters from any of them. This has the advantage that it will account for short stops in the journey. However, it also introduces the disadvantage that it will normally take about 15 minutes to detect termination.

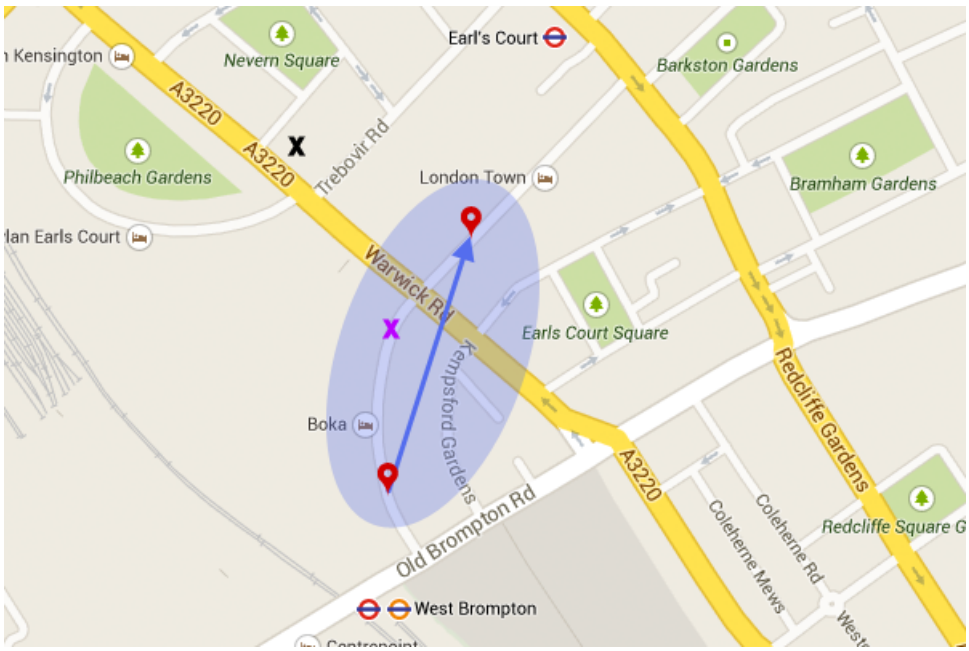

Figure 5.2: Example readings showing the area searched for any missed readings when a journey is started

An optimisation we created for this algorithm is to inject intuition to remove the constant values. The idea we decided on was that the longer a journey is, the more likely any stops along the journey will occur. For example, a short journey from a user's flat to the shops will be unlikely to have any stops. However, on the walk to university a user is more likely to stop to get a coffee or a snack. Using this, we adapted the algorithm to account for stops of length based on the length of the journey, with something along the lines of:

```
maxStopTime = journey.length()/3
```

Since $maxStopTime$ represents an upper bound on the possible length of a stop, we decided to use a more conservative value (i.e. it allows for longer stops than may be common).

Wrongly assigned readings

As with detecting journey starts, since we have used a model that allows more locations to be assigned to the journey than may be accurate, we have to save any entries that deviated from the expected values by a significant amount. After the journey is seen to have definitely terminated, we go through this list and remove any values that were, in fact, indicating the journey had terminated earlier. This also ensures that journeys do not have any duplicated information when they are saved.

While checking previous locations to see if they actually indicated a stationary location, we had to ensure that outliers did not sway our decisions. For example consider the following sequence of readings:

Figure 5.3: An example sequence of readings obtained at the end of a journey

In the above example, if we only looked for readings at the end of the journey that were the same we would get 9, 8 and 7 (as we look backwards) and then see 6 is somewhere else and stop. However, as we can see, 4 and 6 are actually just outliers. So we developed a method which is robust to outliers by maintaining a count of values observed, and we decide a value is an outlier if the count of values before it in the journey is above some threshold (which is relative to the length of the journey).

5.1.3 Using Other Sensors

If we encounter a value that seems to indicate a start (or end), but is not strong enough to absolutely declare this (such as point 3 in Figure 5.1), we use other sensors to increase our knowledge of the current state and make a better decision.

Accelerometer

To determine if a journey has started, we use the activity detection module discussed in 2.6.1 to judge wether the user is moving. If the result comes back with a high probability that a new activity is being carried out, then it is more likely that a journey is currently taking place. Similarly if there is a low probability that an activity is being carried out, it is more likely that the user is in a stationary position.

WiFi

When a journey has possibly started/ended, we query the WiFi manager to check wether there has been a change in WiFi connection in the time frame created by the readings. This uses the intuition that if the users are usually connected to a router when they are in a location that is stationary, i.e. connected to wifi whilst at work or home. Therefore a change in the connection of wifi usually indicates that the user has indeed started/ended a journey.

5.2 Improving Location Estimates

One of the biggest questions posed by this investigation was to see how much we can improve the location estimates given by the system. In order to provide clients with a more accurate estimation of where the users are, we employ a number of filtering techniques. A key advantage of splitting location readings into journeys is that we only have to apply these filtering methods when the user is actually on a journey, which allows us to consume a lot less CPU time.

5.2.1 Adaptive Location Updates

When Android services register for locations updates, they provide thee pieces of information:

Provider Which Location Provider should be used. Possible values are the GPS provider and the network provider

minDistance Throttles location updates so successive updates are at least a certain distance apart.

minTime Throttles location updates so they are only sent when a certain amount of time has passed between successive updates. The documentation stresses that this is not intended to be exact, and updates can (and probably will) occur at time intervals not consistent with the value passed here.

To achieve more accurate readings, we should use the GPS provider whenever needed. However, Android does not provide an in-built solution to automatically change providers, so we had to build a custom solution to handle this. The process we implemented detects when the GPS provider has been enabled/disabled, and changes providers accordingly.

We switch to GPS (if available) whenever network readings consistently fall below a certain threshold. For example, if we have not received a new update that has accuracy above 300m, we will turn on GPS. Instead of completely switching providers, we will actually have both running for a short period of time. If we find network location is still not performing well after a certain period of time, we will turn it off and use GPS only. We then intermittently turn on network locations to check if we have reached an area that has better readings. The system is, however, heavily biased towards using the network provider as much as possible. We found that turning on GPS to look for updates too frequently does not help due to the time to first fix. Therefore, GPS is only turned on when the network provider is seriously struggling to provide accurate updates.

Another component is how we change the minTime and minDistance to achieve more accurate readings when we needed them. The default values used were 2 minutes for minTime and 150m for minDistance. These values were calculated as an approximation to the distance that is larger than the length of a house. Using this we were able to ensure we do not get unnecessary readings for a user whilst they move around their house. The values, however, also work well for larger stationary locations, e.g. university, where they provide a good update interval for estimating the size of the location.

When we have confirmed that a journey has started, we decrease these minimum values to 60 seconds for minTime and 50m for minDistance. These values allow far more accurate readings, which are useful for when the user is moving. By only lowering the values when we have confirmed a journey has started, we ensure the battery is not over-used by constantly alerting Lokey. When we have confirmed the journey has been completed, we revert back to the higher minimum values.

Finally, when a client has enabled live tracking (section 5.3), we decrease the values to the minimum possible values. We do this so that clients remain as updated at all times as possible. Due to the filtering techniques highlighted in the rest of this section, clients will still only receive highly filtered values, reducing the amount of processing they will have to carry out.

5.2.2 Filtering

A key contributor to the success of this investigation will be the techniques used to filter location readings to reduce their uncertainty, and the extent to which we can improve clients awareness where the user is. The main techniques we used to achieve this were dead reckoning, and a simplified form of the Kalman filter.

The first filter we pass all readings through checks whether two readings are very close to each other. If a new reading is less than 10 meters away from the last reading, we do the following:

- If the new location has a higher accuracy, we update the previous location with the new locations' coordinates and accuracy value.
- If the old location has a higher accuracy we completely ignore the new location.
- If the new location has a bearing associated with it and the previous location did not, we update the previous location to use the new locations bearing.

We then use dead reckoning and a technique inspired by the Kalman filter to reduce the inaccuracy of further readings. To calculate a more accurate position based on the raw reading, we do the following:

Calculate a 'base' location which lies on the edge of the circle depicting every position which this reading could possibly include. The base location is found by using the following algorithm, which we use as a computationally cheaper alternative to finding the intersection between the circle and the line created by following the current bearing from the last position:

1. Find the distance from the last position to the centre of the new reading.
2. Subtract the new accuracy from this distance, giving the distance to the point on the circle which is closest to the current position.
3. Find the 'base' by following the current bearing (between the last two positions) for the distance calculated in 2.

From this base position, we use a factor of the previous speed to calculated a new position within the radius of the reading. The direction used is a weighted average of the bearing

between the last two positions and the bearing towards the reading.

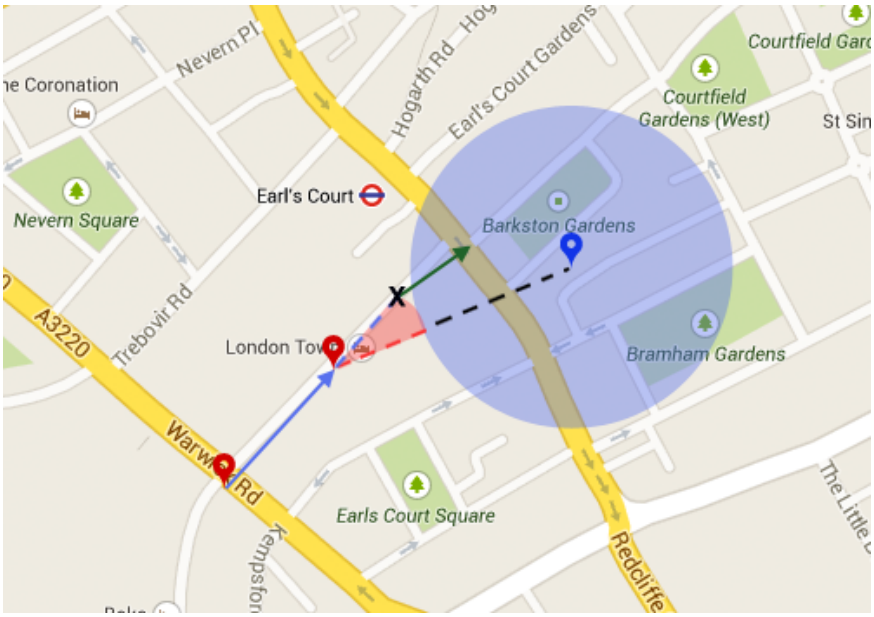

Figure 5.4: An example sequence of readings showing how a more accurate location is extracted from an inaccurate reading. The x marks the calculated 'base'

5.3 Location Tracking

This feature replicates the standard Android location updates, but uses Lokey's technology to filter and enhance location values sent to the client. The client uses the following API to request location updates:

```
public void requestLastLocation()

public void startTrackingLocation(LokeyLocationTracker tracker)

public void stopTrackingLocation()
```

where LokeyLocationTracker is:

```
public interface LokeyLocationTracker
{
    public void gotNewLocation(Location location);
}
```

In addition to the filtering mentioned in section 2.3.2, we also use the techniques discussed in Chapter 5.2 to deliver precise locations to clients. One key observation we made during development of this feature was that it is better for clients if location updates are made less frequently. Instead of sending through a number of locations which are very close, clients can make more use of fewer, but more accurate, updates. By also providing the user with

accurate measures of the current speed, overall speed, current bearing and overall bearing, clients can themselves predict where the user is going up to a certain extent, and the goal of Lokey is to update them frequently enough to keep their estimates correct.

5.4 Frequent Locations

Saving the users frequent locations was a key feature, as the destination prediction module depended heavily on this. One of the early design choices we made was to ensure that a location is only saved as a frequent location if the user does indeed spend a sufficient amount of time there. For this reason, frequent locations are only saved once Lokey is sure they are indeed important to the user.

When a journey is started, we will have a good measurement of where the user was in between the end of the last journey and the current time. This information is the primary source for our frequent locations calculation, as we can accurately say the user spent a signification amount of time in this area.

Each frequent location is saved as a latitude-longitude pair, along with an estimated radius of the area that location covers. The area for this is determined from the stationary location used to calculate the frequent location. When a stationary location is processed, it saves any values that were deemed to be part of the same place but still deviated from the mean. These deviants are then used to calculate an approximate radius for the frequent location. This is highlighted in Figure 5.6

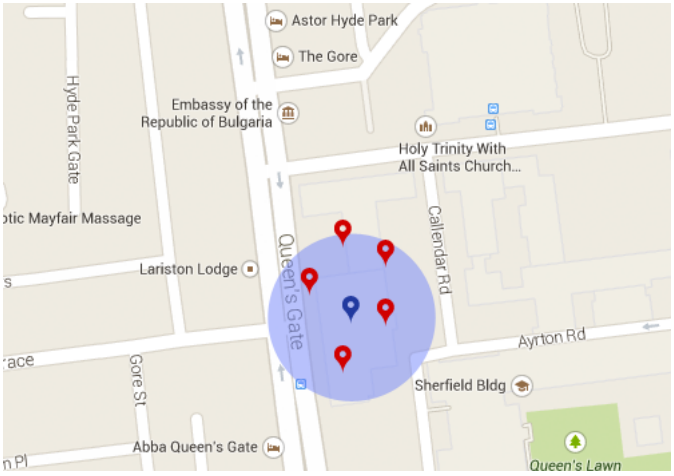

Figure 5.5: Diagram showing a number of readings observed at a stationary location. The red markers indicate readings and the blue marker indicates the saved locaiton, with the blue circle indicating the radius saved.

When another stationary location is observed, it is compared to any saved locations in a similar area. It can then either be saved as a new frequent location, or merged with an existing one. To merge two locations together, we use the following algorithm:

```
// find furthest point on line from center1 (going away from center2)
point1 = center1.followBearingForDistance(center2.bearingTo(center1), radius1)

// find furthest point on line from center2 (going away from center1)
point2 = center2.followBearingForDistance(center1.bearingTo(center2), radius2)

// find midpoint of these two points
newCenter = averageBetween(point1, point2)

// get radius
radius = newCenter.distanceTo(point1)
```

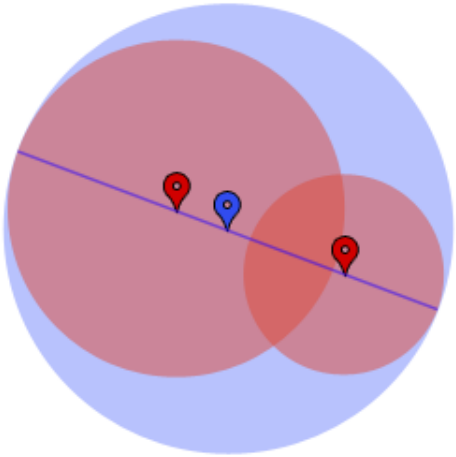

Figure 5.6: An illustration of the algorithm to detect the smallest circle that covers two circles.

At this point it may actually be the case that the new location overlaps with another existing location. However, it would be wrong to merge these two locations if they overlap where the calculated location has covered areas that were not covered by the original, smaller, locations.

Developers can access frequent locations with the call:

```
public void getFrequentLocations(latitude, longitude, radius)
```

5.5 Destination Prediction

The destination predictor is built as a natural extension to the frequent location tracker. The destination predictor creates a Markov chain consisting of the frequent locations as states. The transitions between states represent journeys the user has been observed to take (between frequent locations) in the past.

Each transition between locations in our Markov chain is associated with the following elements:

Time The times at which this transition should be enabled. Values were chosen to reflect patterns users are likely to exhibit through their journeys:

- EARLY_MORNING - 5 A.M. to 8 A.M.
- MORNING - 8 A.M. to 11 A.M.
- NOON - 11 A.M. to 1 P.M.
- AFTERNOON - 1 P.M. to 5 P.M.
- EVENING - 5 P.M. to 8 P.M.
- NIGHT - 8 P.M. to 11 P.M.
- SLEEP - 11 P.M. to 5 A.M.
- BREAKFAST - EARLY_MORNING ∪ MORNING
- LUNCH - NOON ∪ AFTERNOON
- DINNER - EVENING ∪ NIGHT
- ALL_DAY

Day Days of the week for which this transition should be enabled:

- MONDAY : SUNDAY
- WEEKDAY
- WEEKEND
- ALL_WEEK

Intial Bearing The bearing the user took for the first portion of this journey. This is used to determine the probability of a transition being taken.

A key technical challenge here was choosing when to update the values for a transition. Due to the discrete time and day values, it became very important that the destination predictor tried to keep the values as specific as possible, to maintain correct probabilities for transitions. We decided to keep another variable with each transition that tracks requests for changes. When this reaches a certain threshold, we update the prediction to become more general (predictions always start very specific).

Users access destination predictions with the call:

```
public void getPredictedDestination()
```

This will send a message back to the user containing the predicted destination for the current journey, or *null* if the user is not currently on a journey or the destination cannot be determined. This was a design choice, as we could have potentially given the user each of the possible predicted destinations. However, we decided it would be easier for the user to process the results if we simply gave them the most likely destination based on the current state.

5.6 Geofencing

One of the features we felt was essential to the success of this framework was the ability for clients to be notified when the user enters (or exits) certain areas. This feature is exposed to the developers through the following APIs:

```
public void registerPointOfInterest(String id, long latitude,
                  long longitude, float range)

public void unregisterPointOfInterest(String id)
```

These locations are then saved into the `PointsOfInterestDatabase`. Each point of interest can be uniquely identified with the id given and the package of the client, which does not need to be passed as a parameter as it can be determined at runtime.

The complexity of this problem lies in knowing when the device is in one of these points of interest. A naïve solution would be to go through every point of interest whenever we get a new location estimate, and check if any have been entered (or exited). However, since Lokey may be used by a large number of applications, each of which may have a large number of points of interest, this is not a practical solution. Also, location estimates may come very frequently (especially if GPS is on), so evaluating the whole list of points at every update would be very time consuming.

A simple optimisation to the above problem would be to sort the points of interest in terms of distance from the current location. This way, we can stop searching through the list as soon as we determine we are not near a certain point (as any point further down the list must be further away). Maintaining the sorted list will not be as expensive as going through the whole list on every update.

To obtain our general solution, we use information we already know about the user to lazily load only the points of interest that we judge are likely to be encountered. There are two cases:

- If a user is stationary then only points of interest which are incredibly close are loaded (currently set to 100m).
- If a user is on a journey, every time a new location estimate is calculated we do the following:
    1. Get the new overall bearing. Use the journey speed and journey time to calculate a search area radius.
    2. If the previously used bearing and distance are not very different (up to some delta) then exit
    3. Retrieve form the database all points of interest which are within the distance `journey.getOveralDistance()` from the last waypoint. This approximates a circle around the last waypoint.
    4. Filter these points, so only those that satisfy the following condition are kept:

```
|origin.bearingTo(point) - bearing| < δ
        ∧ origin.distanceTo(point) < δ'
```

where δ is the angle added to the bearing in both directions to create a sector and δ' is the distance calculated in step 1.

A crucial part of the success of this system is to reduce the number of times steps 3 and 4 are carried out. To do this, instead of running them for every location update, we only run them if the new proposed segment is significantly different from the previous segment. The idea is that at the start of the journey this process will happen relatively frequently, but toward the end the segment will have become fairly large as the search radius grows smaller as journey time decreases.

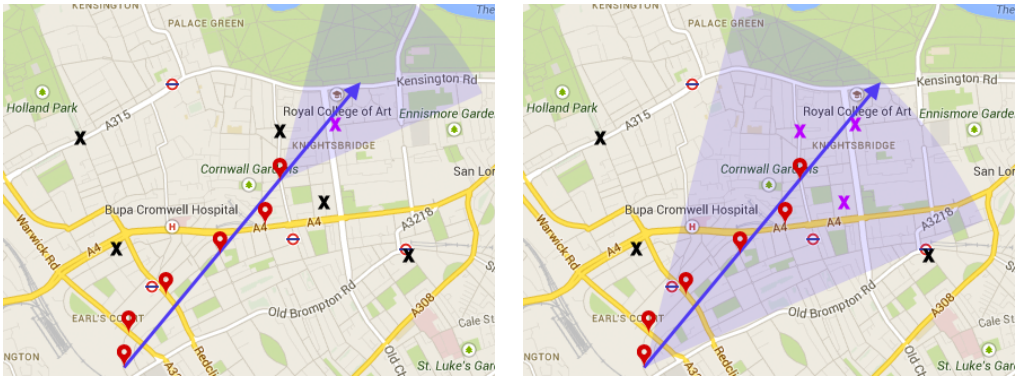

Figure 5.7: Example showing which points of interest are lazy loaded by our system. Red drops indicate the location estimates along a route, the blue arrow indicates the average bearing at the last drop and the blue section shows the area in which points of interest are considered (purple crosses are points that are loaded and black crosses are points that are not)

As shown in figure 4.1, our choice of where to place the search area makes a big difference. If the search area has its origin at the last observed waypoint (as shown on the left), we are able to reduce the overall size. However, this has the negative property that the search is focused mainly at points away from the last waypoint, and any points of interest which is close to the last waypoint but do not follow a very similar bearing are excluded.

On the right, we see the solution we have taken, which is to place the origin at the initial waypoint. This way, even points which are a little further out but follow a similar *overall* bearing are considered. The downside is that as the journey gets longer, the search area will grow very rapidly. We have come up with two possible solutions to this problem (but have not implemented either as they were not a key part of this investigation):

- Since the journey is so long, we are more certain about the overall bearing, and so we can reduce the delta added to the bearing to obtain the upper and lower limit as the length of the journey increases.
- If the journey gets long enough (say more than 10 miles), we should only consider more recent waypoints, as the start of the journey could become either redundant

or irrelevant (e.g. if a driver is travelling to get to the motorway then their initial journey will have a completely different direction to their motorway journey).

5.6.1 Interpolation

A key observation we made is that due to the intermittent nature of the location updates, smaller geofences can be easily missed. This is shown in Figure 5.8.

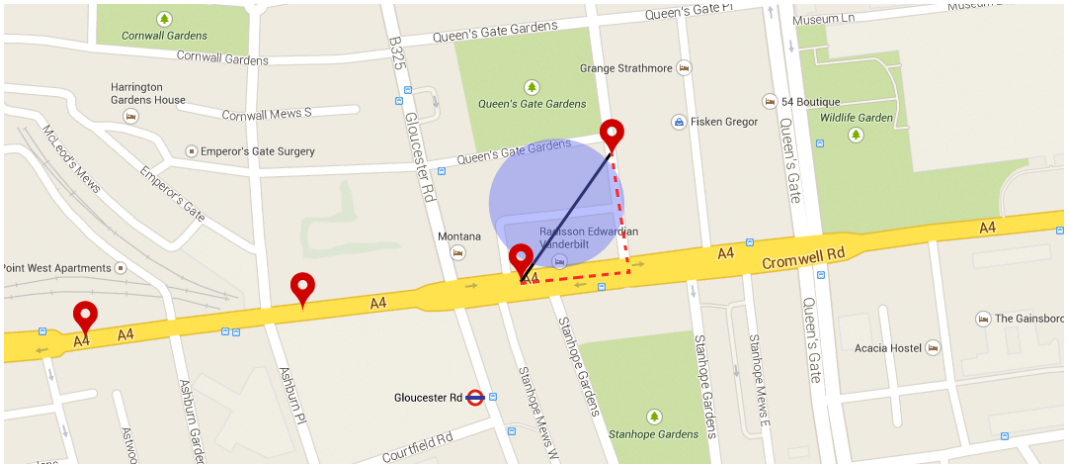

Figure 5.8: Example showing a point of interest missed due to intermitent updates.

As we can see from the figure, it is very unlikely this geofence was not entered. However, our current system would not be able to detect this as the location reading does not lie directly inside its perimeter. To alleviate this problem, we use the algorithm described in Listing 5.1. This algorithm uses a number of heuristics, which means it does not catch all cases. We feel that while this algorithm can be greatly improved, it provides a very stable foundation. In fact, we developed a more accurate model that used circle line intersections, but this algorithm performed better as it heuristically missed geofences that were not crossed but looked as though they were.

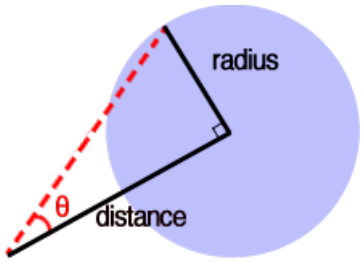

Figure 5.9: Calculation of θ to determine the maximum difference in bearing between the last location with the geofence center and with the current location.

Listing 5.1: Algorithm to detect geofences missed due to intermittent updates

```
distanceDiff = distanceBetween(current, last)
timeDiff = current.time - last.time

// won't work for readings that are too far away
if distanceDiff < 200 && journey.avgSpeed > 1.2*(distanceDiff / timeDiff)
    lat_last_diff = geofence.center.latitude - last.latitude
    lat_cur_diff = geofence.center.latitude - current.latitude
    lon_last_diff = geofence.center.longitude - last.longitude
    lon_cur_diff = geofence.center.longitude - current.longitude

    if last_last_diff > 0 != lat_cur_diff > 0 && lon_last_diff > 0 != lon_cur_diff > 0
        bearingGeo = bearingBetween(last, geofence.center)
        bearingCurrent= bearingBetween(last, current)
        bearingDiff = angleDifference(bearingCurrent ,bearingGeo)
        angleMargin = atan2(geofence.radius / distanceBetween(geofence, last))
        if beringDiff > angleMargin
            return
    else if lat_last_diff > 0 != lat_cur_diff > 0
        if ! (abs(lat_last_diff) < geofence.radius && abs(lat_cur_diff) < geofence.radius)
            return
    else if lon_last_diff > 0 != lon_cur_diff > 0
        if ! (abs(lon_last_diff) < geofence.radius && abs(lon_cur_diff) < geofence.radius)
            return
    else
        return

    // if code reached this point, geofence has been entered and exited
```

Alerting The Client

To alert the correct client that a point of interest has been entered or exited, we have decided to use *Broadcasts* and *BroadcastReceivers*. We found this to be the most effective way to alert clients, as the Lokey Service will be able to alert clients even if they are not currently bound to the service. Using this method, whenever a point of interest is entered or exited, the LokeyService will send out a generic broadcast to the system, containing the package (which client registered this point) and id of the point of interest.

To intercept this alert, clients must register that they want to receive this broadcast in the manifest for their application. They must also interpret the Intent they receive correctly. We have highlighted this process in detail in the User Guide (Appendix A).

5.7 Activity Tracking

```
public void getCurrentActivity()
```

This returns a message after a couple of seconds containing one of the following values:

- ACTIVITY_NONE
- ACTIVITY_WALKING
- ACTIVITY_RUNNING
- ACTIVITY_DRIVING

5.7.1 Step Detection

To detect steps, we filter the signals provided into a new signal where steps can easily be detected. Figure 5.10 shows an example of the readings obtained by the accelerometer whilst a user walks with the device in their pocket.

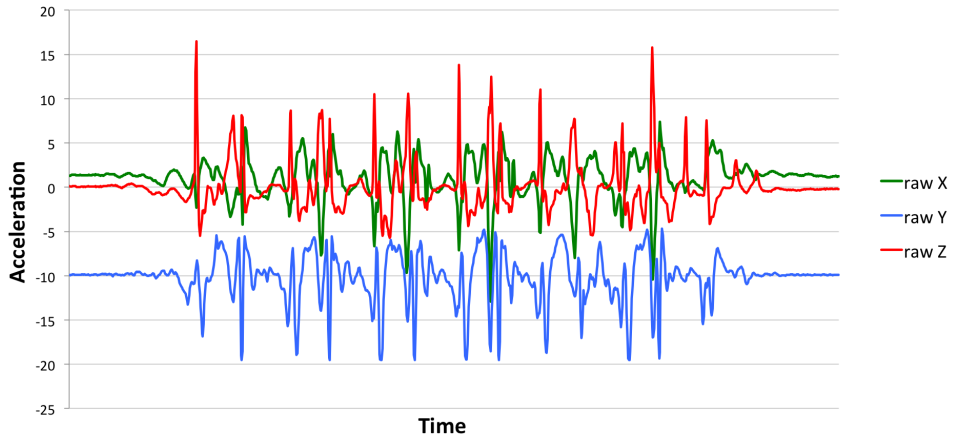

Figure 5.10: The raw values obtained from an accelerometer whilst a user walks with the device in their pocket.

We then take the magnitude of these values and subtract 9.8 (for gravity). The results of this are pass through a low pass filter and the differential of this gives us a signal in which the 0 crossings represent footfalls. An example of these processing steps are shown in the Figure 5.11.

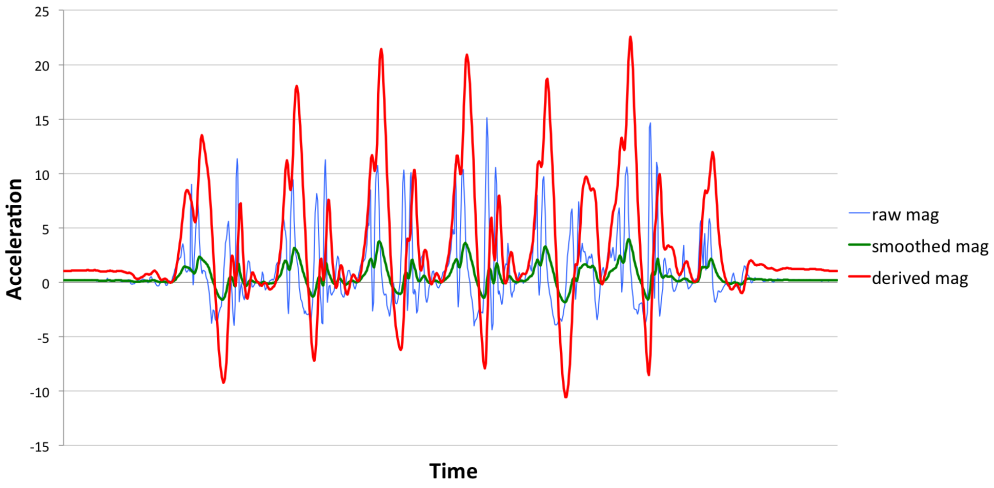

Figure 5.11: The derived values obtained from an accelerometer whilst a user walks with the device in their pocket.

By analysing (in real time) the output of these functions (the derived magnitudes in Figure 5.11), we can estimate an average speed in terms of footfalls per second. We use a moving average with a window larger than the size of the average time between footfalls to maintain a constant estimate of the current speed in footfalls per second. From this speed value, we can detect whether the user is walking or running; we use 70 steps per minute or below to classify the activity as walking and over 70 to classify the activity as running.

An important thing to note here is that we only consider steps from one foot by only considering 0-crossings that go from negative to positive. This is because we found that when the user has the device in their pocket, steps by one foot are a lot more prominent that the other (as is shown in Figure 5.11 by the fact that every alternate peak in the red line has a much lower amplitude). This led to the observation that in some cases it was very hard to detect the opposite leg moving at all, resulting in step counts being flawed. By taking only one foot into account, we greatly improved our estimated speed value.

5.7.2 Driving

Detecting that a user is driving is carried out in conjunction with the location tracker. To detect driving, we see if the user is on a journey and has a certain speed, but the accelerometer does not detect any repeated motion (as is observed by walking or running). This does mean that many activities (such as skateboarding) would be incorrectly classified as driving. However, we have decided to keep this functionality in anyway because a majority of the time, if the user is moving without repeated motion, they are most likely driving.

5.8 Continuous Running

One of the key components for the LokeyService to be successful would be to ensure that it is always running on the users device. This does not necessarily mean it is regularly computing something, but does mean it should always be alive to receive a location change alert from Android. To ensure this was the case, a number of changes have to be made to the code style of the service.

The first thing we had to ensure is that there is as little RAM used as possible. As mentioned in Chapter 3.2.1, the operating system may choose to kill a service at any time if it needs to reclaim some memory. To ensure our service is killed as infrequently as possible, we ensure there very little memory used while the system is running.

We also must ensure that any information in local memory is always backed up to a persistent storage. This allows the service to resume form its previous state if it were to be killed. To allow this, we created a small library that allows services to save and resume their state as required. Due to the fact that Android does not inform the app when a service is killed, this does mean that we have to save the current state whenever it has changed. An example of this is shown below:

```
private void save() {
    String js = journey == null ? null : journey.toString();
    preferences.putString(KEY_JOURNEY, js);
    preferences.save();
}
```

5.9 Permissions

For the application itself, we created an inter-app permission system that was easily extendable. The basic idea is that in the Client table in our database, we save all the permissions a user has granted to each client. We have, however, made the choice that users are only in control of the raw data a client has access to. For example, if a user has said a client should not have access to their frequent locations, this only mean the client cannot directly access the frequent locations. The client will still be able to access other features (such as the destination predictor) which use the frequent locations.

The permissions system relies on the permissions defined in LokeyPermissions.java. Here, we specify all the permissions and their associated text (as shown to the user). Adding a permission is as simple as defining a new permission here and using it where required.

The biggest challenge we had in implementing the system was how to alert the client that a permission has been denied. As we have chosen to only allow communication through messages, we have to send a message to the client saying a permission has been denied. However, this does mean that the client may chose to completely ignore the callback. In this case, they may be kept waiting for a different callback which will never come. The only way to combat this is to inform the developers so they change their approach accordingly.

5.10 Summary

In this chapter, we have detailed the implementation of features discussed in Chapters 3 and 4. The underlying core technology involves analysing location updates to extract useful information. Many of the features are build on the journey tracking component, which uses location updates to track whether the user is currently on a journey or not.

Another important component is the adaptive location updates. By analysing updates, we are able to judge when to switch from the network based location provider to the more accurate GPS provider. We also detail our approach to solving the various problems with Android's current geofencing solution. Other features detailed include the creation of the Markov model for destination prediction and clustering of frequent locations.

6 | Results and Evaluation

This chapter provides results observed during our final stage of testing. We show significant reductions in battery use as well as improvements in accuracy. We will also describe the final product, and how the user is able to interact with it.

6.1 Lokey Application

The Lokey application will be available to users for download from the Google Play Store. This section will describe what the end users will see when they open up the application, and how they can interact with the data their clients are using.

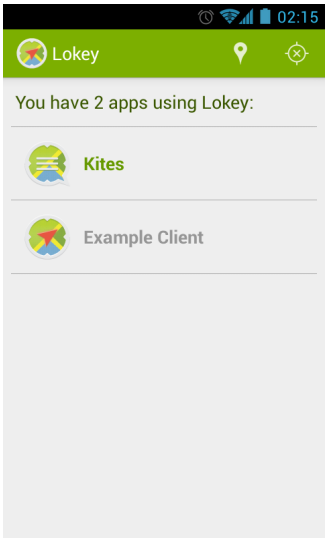

Figure 6.1: Lokey home page

Figure 6.1 shows what the user sees when they open Lokey. The button on the top right is the power button. Using this button, the user can turn off Lokey's background service, thus turning off all functionality. The next button over takes the user to a view which shows them details about the frequent locations Lokey has found. This is shown in Figure 6.7 and Figure 6.8.

Any active clients are listed on the home page, and the user can click on them to see more details about the client.

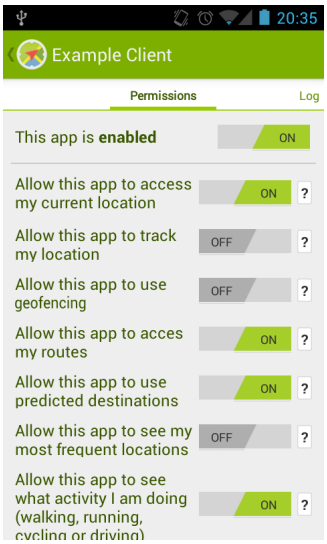

Figure 6.2: Clients permissions

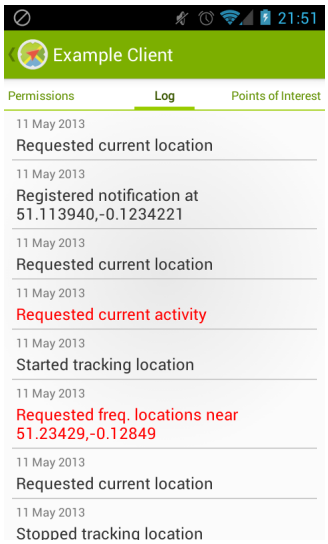

Figure 6.3: Clients call log

Figure 6.2 shows the user all permissions the client currently has, and allows the user to change them. The topmost switch will turn off all functionality available to the client. The little question marks allow users to ask why the application wants to use certain features. When clicked, the user will be shown a small text bubble which displays the reason given by the client when it first registers with Lokey.

Figure 6.3 details calls previously made by a client. The user can scroll through the Clients history and view how they have been using their data. Entries in red mark calls that have been rejected.

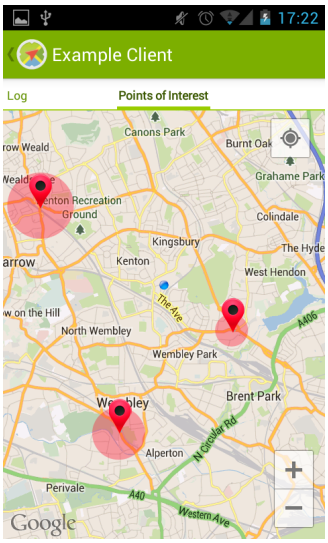

Figure 6.4: A clients registered points of interest

To give users more information about what applications are doing, we let users see which

points of interest a client has registered. Figure 6.4 shows the interface the user sees when they access a clients points of interest. While the user cannot edit any points, they can choose to disable the clients permission for points of interest so future notifications will not go through.

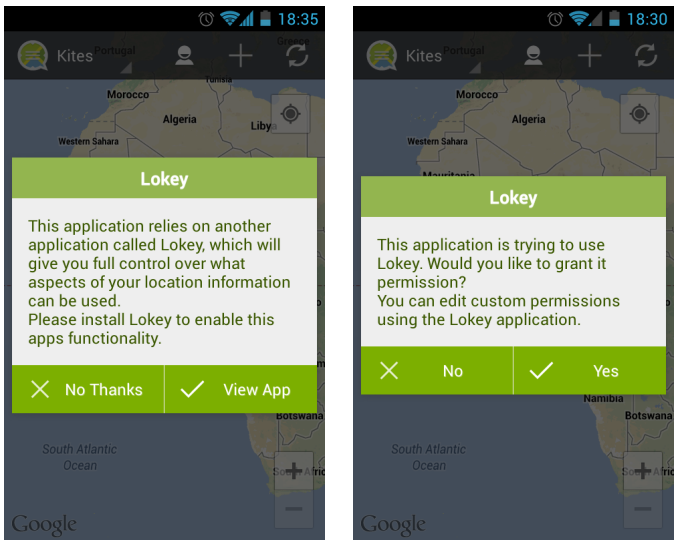

Figure 6.5: The dialogs shown by Clients

Figure 6.5 shows the dialogs a client may present to the user. This first is a dialog asking the user to install Lokey if it is not present on the users device. The second is a dialog asking the user to grant the client permission to use Lokey's services. The result of this dialog is passed directly to Lokey.

6.2 Client Library

The other half of the product is the component to be bundled with every client app. Unfortunately, Android does not currently allow library projects to be exported as Jars (although this feature is promised for the future [10]). Instead, the source code itself must be provided to clients in the form of an Android Library Project. This does mean that third party developers have access to the source code for the client side of Lokey, meaning they can see how communication is carried out. To avoid any negative repercussions of this design, we have ensured that the main service will only accept messages from the client service. We also ensure that all permission checking and granting is only done within the main app, which means developers cannot illegally grant their own permissions.

A lot of work was done to ensure that the client library remained as small as possible. This has two advantages:

- By ensuring there is as little code as possible in the client library, we reduce the exposure of Lokey's internal workings to developers.

- Developers are very keen to ensure their applications remain as small as possible, so having small libraries is often key to the success of a project.

By the end of this investigation, we managed to reduce the client library down to only 24KB, which is practically insignificant compared to the size of most applications ($\sim$ 7MB).

The following files are included in the client library:

LocationNotificationReceiver A helper class which makes it easier for clients to process notifications that a user has entered or exited a point of interest.

LokeyClient An interface that must be extended by all Activities that wish to communicate with the client service.

LokeyClientService The communication endpoint which forwards requests from the clients on to Lokey in the correct form, and forwards responses from Lokey back to the appropriate listeners.

LokeyClientServiceConnection A helper class which makes it easier for developers to connect their Activities with a LokeyClientService.

LokeyLocationTracker An interface to be extended by any class wishing to listen for updates about the users current location.

LokeyJourneyListener An interface to be extended by any class wishing to listen for updates indicating the user has started or stopped a journey.

LokeyVars A private class which holds key information about the communication protocols to be used between the main service and the client service.

LokeyMovementType An enum defining the kind of movement the user is currently exhibiting.

LokeyPermissions An enum defining all the permissions the user has control over.

LokeyJourney A class encapsulating information sent to the clients about the state of a journey.

LokeyFrequentLocation A class encapsulating the information sent to the clients about frequent locations

These files are discussed in more detail in Appendix A.

6.3 Resource Usage

One of the key measures of performance for a continuously running application is the amount of resources used. Applications that use extensive resources are often turned off by users. Continuous applications that get turned off will often function incorrectly or loose information when they are artificially stopped by the user. To avoid this, we rigorously monitored resource usage throughout the investigation, optimising wherever possible to keep usage to a minimum.

6.3.1 Battery

Android provides users with a list of the most battery consuming applications on their device. There are two major caveats associated with this list:

- The list will only contain applications that have drained the battery by a significant amount, other applications will simply not be shown. Considering that components like the screen, Google's continuous polling of servers and the telephone take up a large amount of battery usage, it often takes heavy usage by an application for it to make it into the list.
- Google's developers have expressed numerous times that these statistics are simply estimates and are not to be relied upon heavily.

However, since there are no other ways to measure battery usage, we decided to use the provided statistics. To conduct the tests in the most fair way, we used a device that was not used for any other purpose. This reduced the impact of components like the screen on the battery life. The device was then carried on an example users' person for a week, yielding the following results:

Table 6.1: Lokey battery usage over a week

Day	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Travel Time (minutes)	51	196	28	56	63	131	348
Battery Used	<1%	2%	<1%	<1%	1%	2%	5%

These results show that we were able to reduce the battery usage to very low amounts even for extended periods of time. This is a result of the adaptive location updates only getting frequent updates when the user is on a journey. This explains the direct correlation between the time spend travelling and the battery used.

6.3.2 Main Memory

Memory is important for two reasons. Firstly, users are always wary of applications that use too much memory as they often (wrongly) feel their device will slow down if there is too much stored on the main memory. Secondly, Android actively stops background applications that use too much memory in an effort to reclaim it.

While actively tracking location in the same setup as above, we observed the following results:

Table 6.2: Lokey main memory usage over a week

Day	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Travel Time (minutes)	51	196	28	56	63	131	348
Maximum RAM Used (MB)	9.3	12.4	8.9	8.7	9.1	11.2	12.6

A testament to the efforts of reducing the memory are shown by the fact that over the whole

week, Lokey was only killed 4 times. Comparatively, when we started the investigation Lokey would frequently get restarted approximately 4 times every day.

6.3.3 Persistent Memory

Persistent memory is not as important as the two measured above, but is still worth noting as databases storing location information can often become very large and need to be controlled. Lokey uses a significant amount of persistent memory to save the users information.

During analyses of the memory, we found that the Journey table would grow very large over time due to the fact that all waypoints were stored. To alleviate this, we came up with a system that would merge common routes into one. By using this strategy we reduced memory usage by 600%. In fact, this figure only represents the number of journeys repeated in a certain time, meaning the actual memory saved by incorporating this summarisation technology will increase as the number of repeated journeys increases.

6.4 Geofencing

We conducted the following experiment to highlight the successes and failures of our system when compared to Android's current solution. Figure 6.6 shows

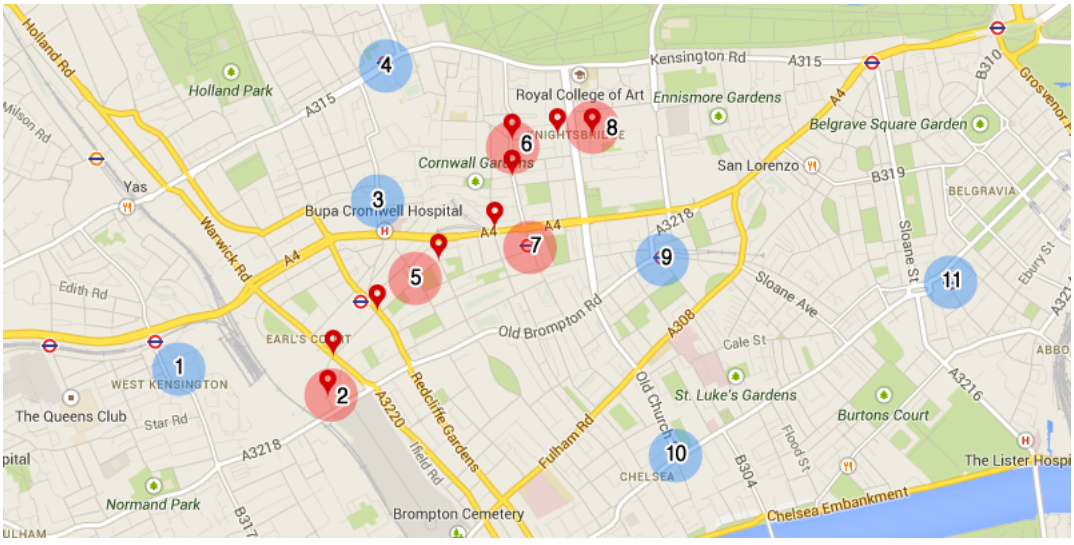

Figure 6.6: The experiment conducted to analyse our solution. Red circles indicate geofences that were actually crossed and blue circles indicate geofences that were not crossed.

6.4.1 Accuracy

We observed the following results compared to Android's solution:

Table 6.3: Geofences entered according to our solution and Android's solution

Point	1	2	3	4	5	6	7	8	9	10	11
Lokey		✔			✔	✔		✔			
Android		✔	✔			✔		✔			

These results provide a number of interesting discussion points. The first is Android's registration of geofence 3 being entered. This is clearly erroneous. After analysing the logs of our filtering mechanism, we realised that Android has registered this as entered because an update near geofence 3 had a large area of uncertainty which happened to cross into the third geofence. Whilst Lokey filtered this out, the Android implementation seems to have kept it.

The second interesting point is the Lokey registered point 5, whereas Android did not. This is due to our interpolation between points. We check if any geofences lie between consecutive updates to make sure no obvious geofences were missed. In this case we found that we had missed geofence 5, and so notified clients that it had been crosses. On the other hand, Android does not do this and so missed it completely.

The last point to note is that both our solution and the current Android solution missed the seventh geofence. Because there was no location update in this area, neither solution was aware the geofence had been crossed. This highlights the limitation of our interpolation system; it only checks for geofences that were crossed directly between subsequent location updates. This geofence lay in the path taken by the user, but was not directly between the two updates and so was completely missed.

Overall, our solution improves upon the current Android solution in terms of accuracy. By using the filtered values and our interpolation system, we can more accurately determine when geofences are entered and exited.

6.4.2 Battery Use

One of the key faults with the current Android 'proximity alerter' is the battery use. Developers have been complaining for years that this feature is almost un-useable once there are a significant number of geofences. While conducting the experiment above, it was impossible measure battery use, as the two solutions had to be run within the same application to ensure they got the same updates. Therefore, we had to conduct the experiment two more times, one with only Android's solution enabled and one with only our solution enabled.

By the end of this journey (which was a 30 minute walk), the Android solution had consumed 3% of the users battery. In contrast, our solution had consumed $< 1\%$ of the battery (it didn't even register on the scale). It is interesting to note that our solution loaded up geofences 2, 3, 5, 6, 7, 8 and 9. This shows allowed Lokey to use far fewer comparisons over the course of the journey than if it simply looked through everything. You can imagine there may be more geofences nowhere near this area, all of which Lokey ignored.

On the other hand, the Android solution starts a new location listener for every new point of interest. This means there are far more comparisons carried out. Table **??** highlights the differences when there are more and more geofences added:

Table 6.4: Amount of battery consumed by Geofencing modules over a 30 minute journey

# Geofences	10	100	1000
Lokey	<1%	1%	3%
Android	3%	7%	22%

6.5 Frequent Locations Found

Figure 6.7 shows the frequent locations that were found by Lokey over a period of 6 days. Going from left to right, the markers indicate my flat, the department of computing at Imperial, and the offices of a startup in London. Interestingly, I visited a number of restaurants around London over this period, which are not shown. As mentioned in Chapter 5, a frequent location is only saved if the user visits it more than once in a certain period of time; I had only visited these restaurants once each.

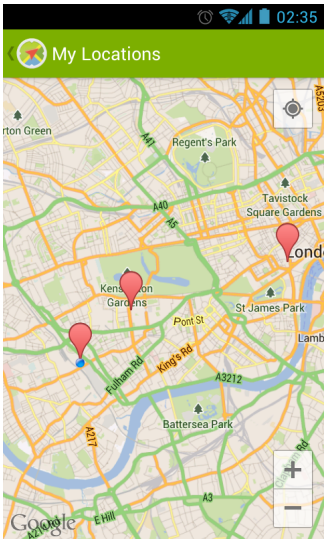

Figure 6.7: The frequent locations found after 6 days of use

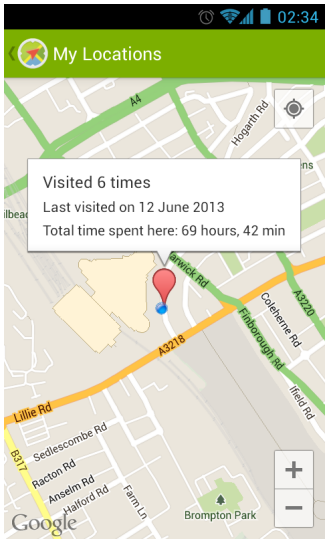

Figure 6.8: The details of a frequent location

6.6 Destinations Predicted

Figure 6.9 describes the model that had been created at the end of the six day period described above.

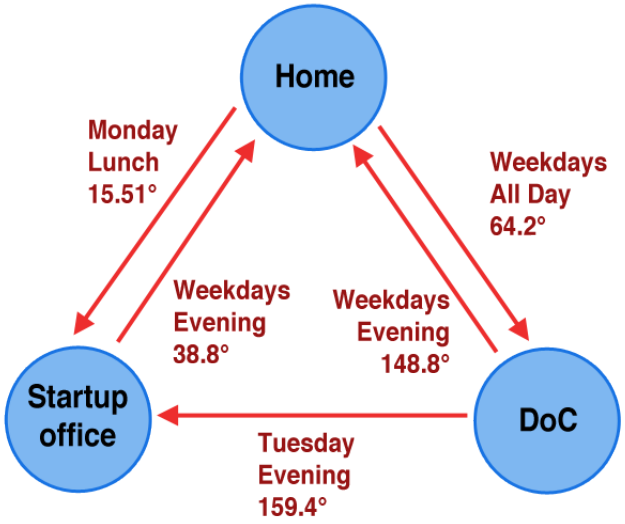

Figure 6.9: The model created after 6 days of use

As we can see, journeys which are conducted frequently, such as form my home to Imperial, quickly converge to a correct value. However, something we did not think of whilst implementing this feature is that often users will walk towards the same place no matter what destination they are headed towards. For example, from my home most of the places I will go to involve me going to the closest train station. This bearing cannot accurately be split into multiple destinations.

Therefore, after gathering our results we changed the weighting of the Markov model to account for this additional factor. The final product uses a more complex weighting; suppose there was a location has many outgoing routes with the same initial bearing and only a few different initial bearings. If a new journey starts such that the initial bearing matches the very common initial bearing, then bearing will not be used at all to judge the likelihood of destinations. However, if the bearing matches one of the less commonly seen bearings, the bearing will be used heavily to influence the predicted destination.

6.7 Kites

We decided early on that one of the principal ways to evaluate the success of the project will be to create a sample application which uses Lokey for all its location services. We would then compare the development time for this compared to how long it would take if we were to develop the application from scratch.

The application we decided to develop primarily uses Lokey's points of interest. The app is called Kites, and allows a user to leave notes for their friends at a location. When a user approaches an area around which there is a note that has been left for them, Kites will alert the user. Screenshots of Kites being used can be seen in Appendix B. It also makes use of the users frequent locations to show friends where users like to hang out.

Producing this app without Lokey would take a far longer time. Implementing a service that constantly tracks the users location and checks any registered points of interest when

required would require a significant amount of time and would be essentially duplicating functionality available for free in Lokey. As a rough estimate, we concluded it would take at least another week of solid effort by us to re-implement this feature for Lokey; this is an estimate of how long it would take us to replicate the work on geofencing done in Lokey.

6.8 Developer Feedback

After conducting the investigation, we asked a subset of the initial group of developer consulted on their thoughts about the results. The consensus reached by the majority of them (who were the more amateur Android developers within the group) was that they would not know how to connect to the service. They were worried about Android component life cycle events not being handled correctly as certain things need to be done. We then showed them the Developer Guide (Appendix A), which they said they understood and were more confident they could implement it.

The more advanced developers seemed interested by the product. They were asking more technical questions about resource usage and accuracy. Any developers who had not been convinced by the in-app permission system during the initial questioning had their doubts settled when they actually played around with the user interface. The majority of comments praised the ease of use (for the users).

However, some developers did still show a degree of distrust in relying on another application. They were worried that developers may not understand how the system works, leading to even further mistrust. We agreed that there may be an initial hurdle the users will have to mentally get over to understand what is being done. As this is not standard Android behaviour, developers were also worried future releases of Android may cause some parts of the communication between apps to change, meaning their apps may be left stranded without Lokey.

6.9 Summary

In this chapter, we present the results achieved by our framework. By filtering locations and applying smarter technologies, we are able to drastically cut the amount of battery consumed by our geofencing service. We achieved results significantly better than Android's current system. The framework was also able to accurately learn the users frequent locations. Using this, Lokey was able to generate a Markov model accurately describing possible journeys between frequent locations.

Initial developer feedback was positive, with developer praising the results achieved. We also built a sample application, Kites, which uses Lokey for all its location requirements. Minimal effort was required to get complex features such as frequent locations and geofencing up and running in very little time.

7 | Conclusion

The aim of this investigation was to produce a framework that allows developers to easily integrate complex location features within their mobile apps. We also wanted to help users understand what data apps were requesting about them. These aims are both qualitative rather than quantitative, but initial feedback seems to suggest users and developers have responded well to the framework.

During the course of this investigation, we have produced an application that will reside on the user's device and continuously track their location. The novel concept is that this data is then used by developers (so they can build more location aware applications) as well as the users themselves (so they can see what the app knows about them and even change things if they want to). A further novel concept is logging all calls made by clients to allow users to keep track of what data clients are requesting. We also created an in-app permissions system allowing users to dynamically allow or disallow individual clients to use certain aspects of the framework.

Features of the framework were chosen with a group of Android developers whose experience ranges from beginners to experts. We implemented 5 core features, which were well received when the developers were re-questioned at the end of the investigation. On May 16th at Google I/O, Android announced they would be improving their geofencing features in a number of ways similar to the techniques we have developed through this investigation. While this is disappointing as we had been developing accurate geofencing as our primary selling point, it shows we are on the right direction. This emphasises that developers do not have enough to work with yet, and more location features will continue to make apps smarter.

Initial results have shown that our techniques are able to accurately track users movement, frequent journeys and frequent locations. We have implemented a geofencing solution that is more efficient and accurate than Android's (current) solution. While these results are positive, there is still much room for improvement. The current modules do not leverage each other for maximum potential. For example, the geofencing module could make use of the saved journeys and destination prediction to improve its lazy loading.

7.1 Future Work

While the framework is in a complete form, Lokey will not be released to the public yet. The user interface shown here is merely a proof of concept as the intended goal for the final product will allow more interaction by the user. We have shown here that users can

access their data and change certain parts (the frequent locations). However, the final release will allow users to access and change anything that is saved about them.

There are a number of other features and ideas we contemplated during this investigation, but decided they were not necessary in making this project a success.

Learning

While we have used some machine learning techniques in this investigation, we believe extending the system to learn and truly understand the user would be very helpful for developers. For example, in the destination prediction module we have used certain time intervals to split up the day. However, this would be a lot more accurate if we used a more probabilistic approach. We could then potentially create a Bayesian classifier to help predict the next destination.

Similar to the work done by Zhuang et al. [22], we have implemented adaptive providers. However, they extend this work to learn the applicability of these providers along the users common routes and frequent locations. This would require extending our current system to allow routes and locations to be saved with the preferred providers. While our current solution performs moderately well, implementing a more robust adaptive strategy would definitely improve the accuracy of readings, although maintaining our level of battery use may be a potential issue.

Routes

During the course of this investigation we have treated a journey as a collection of waypoints. A more advanced technique would use maps to deduce an actual route from these waypoints. This would involve understanding where roads are and estimating where users have turned and crossed over etc. Using routes would allow a far more thorough analysis of a users movement. However, this kind of calculation would require far more resources, most likely requiring a large amount of communication with a server calculating these values.

Density Prediction

This idea came about in one of the talks with the developers. If a large enough number of users start using Lokey, a system can be built to predict the number of people in certain areas. For example, this data could be used to calculate the number of people that visit a gym at certain times of the day.

Sub-Locations

Currently, when constructing frequent locations, we either merge or split them. We do not allow overlapping or any other relations of any sort. Ashbrook et al. [1] use sub-locations when learning a users frequent locations. This allows for a single location to contain a number of smaller locations e.g. different departments in university.

Other Devices

A key observation made by the developers we consulted was the disparity of features available on Android compared to iOS and Windows Phone 7. This meant developers had to implement their own versions of features if they wanted a cross platform application that behaved in a similar way across devices. Extending Lokey to be available on other platforms would allow developers to have a common interface when implementing their applications. A system like this would allow developers to feel more comfortable when creating cross platform apps as they would know any work done on one platform can be easily replicated on another.

Cloud Processing

Initially, this investigation aimed to produce a framework which was to use cloud services for data storage and calculations. However, after talking to a number of users we found they would not be comfortable with all their location data stored somewhere else. Developers also expressed concerns about the speed and availability of results if data was only stored in the cloud. For this reason we decided to cut any cloud services form the framework. However, we still feel an online component for Lokey would be beneficial to developers as some features (e.g. the density prediction mentioned above) will only be possible with a more global picture rather than individual user information.

Allowing some work to be done in the cloud could also provide enhanced security for users. For example, we could monitor how users are allocating permissions. If we find that lots of users are blocking a certain application, we could contact the developer and ask them to address specific issues. Or if we find some apps are definitely misusing data, we could remotely add applications to a blacklist which stops them access Lokey on any device.

Mocking

An interesting idea developed by researchers at Cambridge is to allow users to 'mock' permissions granted to applications. They call this *MockDroid* [2]. We see a lot of potential in this idea. The way Lokey has been built would allow a very useful extension following this principal. We could allow users to specify that they want clients to see certain 'wrong' information. For example, the user could set up Lokey to show a specific application completely different frequent locations from their actual frequent locations. This would allow users to potentially see how their data is being used (not just what data is being used) by observing the behaviour of applications once their data has been mocked.

A | Developer Guide

This guide will walk you through Lokey's features. It assumes knowledge of how to build Android applications and their various components. The guide describes techniques used to produce the sample application Kites.

A.1 Downloading and Installing

The first step to integrating Lokey services within your application is to download the client library. The client library is available from *www.rockolabs.com/lokey/download/client.zip*. After extracting this folder, the library can be imported into your IDE using the *import existing project* command. Your project must then reference this library project (as with normal library projects).

The next steps are to declare the following in your applications manifest:

```
<uses-permission android:name="com.rockolabs.lokey.permission.ACCESS_LOKEY"/>
<application>
    ...
    <service android:name="com.rockolabs.lokey.client.LokeyClientService"/>
    <service android:name="com.rockolabs.lokey.service.LokeyService"/>
</application>
```

The first line is the permission that is required for any application to use Lokey. The other two lines simply declare that this app will use the services mentioned. Your IDE will probably complain that it cannot locate *com.rockolabs.lokey.service.LokeyService*. You should ignore this error, as it simply declares it cannot find the class but does not mean the program will not run.

A.2 Hooking up to a LokeyClientService

All interaction with Lokey is done through the *LokeyClientService*. This service can be treated as any other service. The first thing to do is ensure the Activity you want to access Lokey within is declared to implement the interface *LokeyClient*. This allows use to monitor certain aspects of the activities lifecycle while you can still extend third party activities, e.g. the common *SherlockActivity*.

We have provided a class *LokeyClientServiceConnection*, which makes connecting to the *LokeyClientService* simpler. To use this, you simply create a new instance in your *Activity*. You must then bind to the client service using this connection in the *onStart* method and unbind in the *onStop* method.

All of the above points are highlighted by the following *Activity* snippet:

```
public class MainActivity extends Activity implements LokeyClient {
    private LokeyClientServiceConnection serviceConnection;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        ...
        serviceConnection = new LokeyClientServiceConnection(this);
    }

    @Override
    protected void onStart() {
        super.onStart();
        if (!serviceConnection.bound) {
            Intent intent = new Intent(this, LokeyClientService.class);
            bindService(intent, serviceConnection, BIND_AUTO_CREATE);
            startService(intent);
        }
    }

    @Override
    protected void onStop() {
        super.onStop();
        if (serviceConnection.bound) {
            unbindService(serviceConnection);
            serviceConnection.bound = false;
        }
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        stopService(new Intent(this, LokeyClientService.class));
    }

    @Override
    public void serviceConnected() {
        // Do whatever you want with the service!
    }
}
```

Note: using the *serviceConnection.bound* field allows you to check whether the service has bound to the client yet or not. You should not make any calls before you have checked the service has been bound.

A.3 Querying Current Information

There are a number of ways to get information about the users current location state. It should be noted that all calls to the LokeyClientService are asynchronous due to the

nature of communication between this service and the main service run by the Lokey application. While the results are expected to be calculated within seconds, certain calls may take a bit longer and your applications should be written to accommodate this.

Current Location

Using the call *getCurrentLocation()*, you will immediately be returned the last location calculated by Lokey. This will often be a location that has been heavily filtered and refined from Android's own location tracking system. We also manage sources efficiently so GPS will be used whenever possible, and network location will be used at all other times.

This call requires the *CURRENT_LOCATION* permission.

Current Journey

We also provide a couple of calls to check the current status of the users journey. The first call is *isCurrentlyOnJourney()*. This will return *true* if the user is currently moving around, and *false* if the user is currently stationary or is in an area deemed to be a single place e.g. in a shopping mall or university.

The second call related to journeys is *getCurrentJourneyDetails()*. If the user is not currently on a journey this will return *null*. If the user is on a journey, you will receive a *JourneyDetails* object. This holds all the information known about the current journey the user is on.

This call requires the *JOURNEY_DETAILS* permission.

Common Routes

Lokey allows you to view the users more common routes. To access these, you can use the call *getCommonRoutes(latitude, longitude, radius)*. This will return a list of *CommonRoute* objects encapsulating routes the user has frequently taken. The list is ordered in descending order of number of visits. The parameters passed prevent abuse of the users data. You can only retrieve common routes in a certain area. The radius parameter has a maximum value of 20km (any value above this will simply be cut off).

This call requires the *VIEW_ROUTES* permission.

Frequent Locations

Lokey also exposes an API for applications to access a users most frequently visited locations. This is done using the call *getFrequentRoutes(latitude, longitude, radius)*. This will return a list of *FrequentLocation* objects, which encapsulate a place the user has frequently visited. The parameters, similarly to the above call, are used to stop abuse of Lokey's information. Only frequent locations within the specified area will be returned by this call.

This call requires the *FREQUENT_LOCATIONS* permission.

Predicted Destinations

Lokey allows your applications to access where we the think user is going. If the user is on a journey, the call *getPredictedDestination()* will return which of the frequent locations we predict the user is headed towards. The result of this call will be a *FrequentLocation* which will encapsulate the area we believe the user is headed towards.

This call requires the *PREDICTED_DESTINATIONS* permission.

Current Activity

Lokey can use the accelerometer to predict the user's current activity. Applications can access this using the call *getCurrentActivity(seconds)*. The call returns a member of the LokeyMovementType enum. The parameter passed in is the time for which the user's motion should be analysed. The minimum value is 5 seconds and the maximum value is 60 seconds. the lower limit enforces there is enough time to gain a somewhat accurate estimate, and the upper limit is enforced to stop the battery being drained too fast.

This call requires the *ACTIVITY* permission.

A.4 Location Updates

As well as requesting the current location, Lokey provides an interface for applications to receive constant, filtered and adjusted location updates. This enables your applications to have a constant knowledge of the users location using Lokey's proprietary filtering techniques. To register and unregister your listener you must simply use:

```
1 lokeyService.startTrackingLocation(listener);
2 ...
3 lokeyService.stopTrackingLocation(listener);
```

The listener passed in must implement the class *LokeyLocationTracker*. It is very important that clients unregister their listeners when their application is closed, otherwise the client service may crash and the user will be notified of this.

A.5 Geofencing

Geofencing allows your client to be updated when the user enters or exits a certain area. To enable geofencing for your client, you must first create a BroadcastReceiver that will receive these updates. To make this even easier, we provide the class *LokeyPointOfInterestReceiver*. Your receiver should extend this class, and implement the required function. An example of this is shown below:

```
public class KitesLocationNotificationReceiver extends LokeyPointOfInterestReceiver {
    private NoteManager noteManager;

    @Override
    public void enteredLocation(String id) {
        Note note = noteManager.findNoteBy(id);
        alertUserAboutNote(note);
    }

    @Override
    public void exitedLocation(String id) {
        ...
    }
}
```

You must also register to receive the broadcast. This is done in the manifest, as shown below (making sure to replace *KitesPointOfInterestReceiver* with your own receiver cast:

```
<application>
    ...
    <receiver android:name="com.rockolabs.kites.KitesPointOfInterestReceiver">
        <intent-filter>
            <action android:name="com.rockolabs.lokey.geofencing"/>
        </intent-filter>
    </receiver>
</application>
```

Geofences can then be registered as:

```
for (Note note : retrieveMyNotes())
{
    lokeyService.registerPointOfInterest(note.getId(), note.getLatitude(),
        note.getLongitude(), note.getRadius());
}
```

It is essential that developers unregister geofences as soon as they can using the call *unregisterPointOfInterest(id)*. This allows Lokey to reduce the amount of computational load incurred whenever the location changes.

This functionality requires the *GEOFENCING* permission.

A.6 Journey Updates

Journey updates work in a similar way to Geofencing. Your application can register to be woken when the user starts or ends their journey. The receiver you must extend to receive these notifications is the *LokeyJourneyChangeReceiver*:

```
public class KitesJourneyChangeReceiver extends LokeyJourneyChangeReceiver {
    @Override
    public void journeyStarted(long startLatitude, long startLongitude, long time) {
        ...
    }

    @Override
    public void journeyEnded(JourneyDetails details) {
        ...
    }
}
```

You must then declare the receiver to receive the following action:

```
<application>
    ...
    <receiver android:name="com.rockolabs.kites.KitesJourneyChangeReceiver">
        <intent-filter>
            <action android:name="com.rockolabs.lokey.journey"/>
        </intent-filter>
    </receiver>
</application>
```

This functionality requires the *JOURNEY_DETAILS* permission.

A.7 Permissions

An important factor of Lokey is the users ability to deny certain privileges to applications. Following this, your application must be responsive to the fact that users may not want certain parts of their information used by your application. There are two main ways Lokey provides clients with information about permission.

The first is by requesting the status of a permission yourself. Your application may use the following call to check whether the user has granted a specific permission:

```
public void checkPermission(LokeyPermission permission)
```

The other way to check for the status of a permission is to wait to get a permission denied call. Every LokeyClient must implement the function

```
public void permissionDenied(LokeyPermission permission) { ... }
```

Inside this function, you should write your code which deals with the denial of a permission. For example, you may show the user a dialog explaining why the permission is required, or simply turn off certain functionality within your application. It is important to note that the *LOKEY* permission is required for all calls.

B | Kites Screenshots

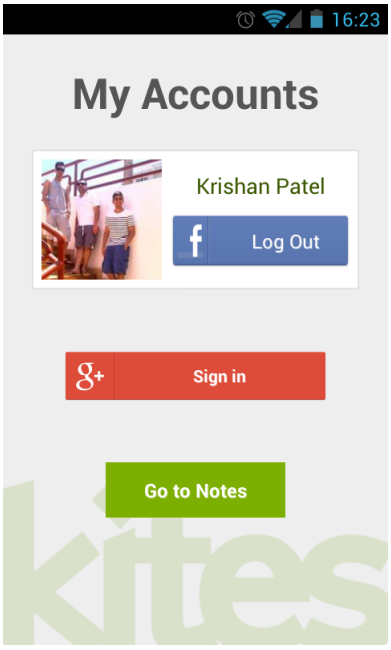

Figure B.1: Users are allowed to log in via either Facebok or Google Plus

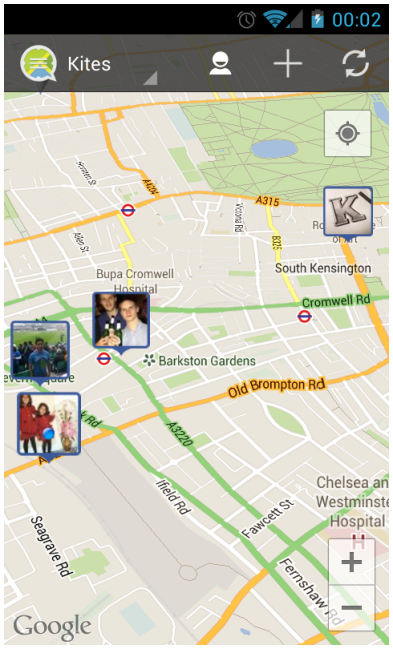

Figure B.2: The map view shows where the users friends have left notes for them

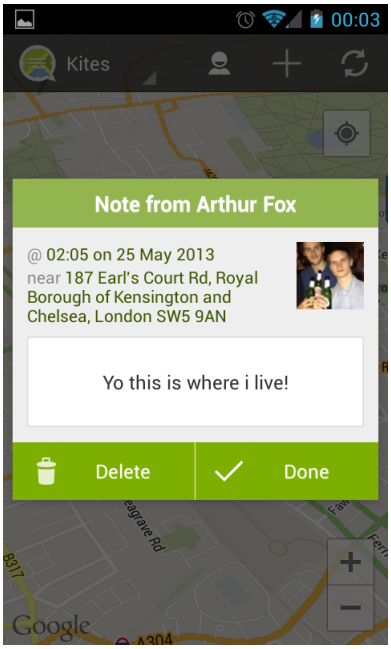

Figure B.3: The view allowing the user to read a note that was left for them, as well as more details about the note

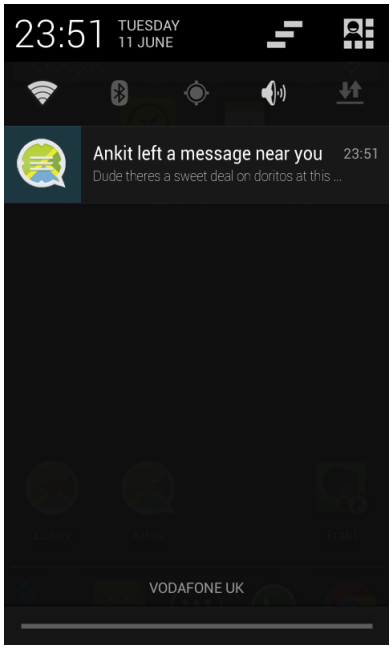

Figure B.4: An example notification created by Kites when the user approaches a note left for them

Bibliography

[1] Daniel Ashbrook and Thad Starner. "Learning significant locations and predicting user movement with GPS". In: *Wearable Computers, 2002.(ISWC 2002). Proceedings. Sixth International Symposium on*. IEEE. 2002, pp. 101–108.

[2] Alastair R Beresford et al. "MockDroid: trading privacy for application functionality on smartphones". In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. ACM. 2011, pp. 49–54.

[3] Amiya Bhattacharya and Sajal K Das. "LeZi-update: an information-theoretic approach to track mobile users in PCS networks". In: *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*. ACM. 1999, pp. 1–12.

[4] Max J. Egenhofer. "Spatial SQL: A query and presentation language". In: *Knowledge and Data Engineering, IEEE Transactions on* 6.1 (1994), pp. 86–95.

[5] Apple Inc. *Location Awareness Programming Guide*. URL: `http://developer.apple.com/library/ios/#documentation/userexperience/conceptual/LocationAwarenessPG/CoreLocation/CoreLocation.html` (visited on 05/21/2013).

[6] Google Inc. *Android Location Services*. URL: `http://developer.android.com/guide/topics/location/index.html` (visited on 05/21/2013).

[7] Google Inc. *Android Location Strategies*. URL: `http://developer.android.com/guide/topics/location/strategies.html` (visited on 05/21/2013).

[8] Google Inc. *Android Open Source Project*. URL: `http://source.android.com/` (visited on 05/21/2013).

[9] Google Inc. *Android Services*. URL: `http://developer.android.com/reference/android/app/Service.html` (visited on 05/21/2013).

[10] Google Inc. *Android Services*. URL: `http://developer.android.com/tools/projects/index.html` (visited on 05/21/2013).

[11] Groupon Inc. *Groupon Android Application*. URL: `https://play.google.com/store/apps/details?id=com.groupon` (visited on 05/21/2013).

[12] Adrian Kingsley-Hughes. *It's revenue, not market share, that's attracting devs to iOS.* URL: http://www.zdnet.com/its-revenue-not-market-share-thats-attracting-devs-to-ios-7000016615/ (visited on 06/10/2013).

[13] R. Libby. *A Simple Method for Reliable Footstep Detection in Embedded Sensor Platforms.* 2009.

[14] Greg Milette and Adam Stroud. *Professional Android sensor programming.* Wrox, 2012.

[15] Andrew Ofstad et al. "Aampl: Accelerometer augmented mobile phone localization". In: *Proceedings of the first ACM international workshop on Mobile entity localization and tracking in GPS-less environments.* ACM. 2008, pp. 13–18.

[16] Jeongyeup Paek, Joongheon Kim, and Ramesh Govindan. "Energy-efficient rate-adaptive gps-based positioning for smartphones". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services.* ACM. 2010, pp. 299–314.

[17] PCMag.com. *Android Nears 75 Percent Smartphone Market Share.* URL: http://www.pcmag.com/article2/0,2817,2418945,00.asp (visited on 05/21/2013).

[18] Nishkam Ravi et al. "Activity recognition from accelerometer data". In: *Proceedings of the national conference on artificial intelligence.* Vol. 20. 3. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2005, p. 1541.

[19] Jun Rekimoto, Takashi Miyaki, and Takaaki Ishizawa. "LifeTag: WiFi-based continuous location logging for life pattern analysis". In: *Lecture Notes in Computer Science* 4718 (2007), p. 35.

[20] Steven J. Vaughan-Nichols. *How Google–and everyone else–gets Wi-Fi location data.* URL: http://www.zdnet.com/blog/networking/how-google-and-everyone-else-gets-wi-fi-location-data/1664 (visited on 05/21/2013).

[21] H. Ying et al. "Automatic step detection in the accelerometer signal". In: *4th International Workshop on Wearable and Implantable Body Sensor Networks (BSN 2007).* Springer. 2007, pp. 80–85.

[22] Zhenyun Zhuang, Kyu-Han Kim, and Jatinder Pal Singh. "Improving energy efficiency of location sensing on smartphones". In: *Proceedings of the 8th international conference on Mobile systems, applications, and services.* ACM. 2010, pp. 315–330.