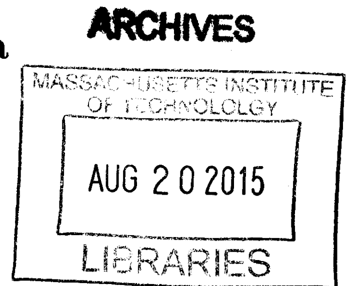
A Framework for Real Time Passive Data Visualizations


ARCHIVES
AUG 2 0 2015
LIBRARIES

by

Charles Edward Sims II

S.B., C.S., Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Author . . . . Signature redacted . . . . . .
Department of Electrical Engineering and Computer Science
September 5, 2014

Certified by . . . . . Signature redacted . . . .
Alex 'Sandy' Pentland
Professor
Thesis Supervisor

Accepted by . . . . Signature redacted . . . .
Prof. Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

A Framework for Real Time Passive Data Visualizations

by

Charles Edward Sims II

Submitted to the Department of Electrical Engineering and Computer Science
on September 5, 2014, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, I leveraged the openPDS and funf frameworks to design and implement a framework for visualizing and analyzing data collected passively from mobile devices. The framework integrates with the process of data collection on the Android platform, storage and task management on the backend server, and finally display in the user-facing browser applications. The two javascript applications were built to demonstrate the functionality of the framework and test the results. A test framework was also designed and created to simulate various numbers of mobile clients and find the limits of the real time functionality given reasonable environment resources.

Thesis Supervisor: Alex 'Sandy' Pentland
Title: Professor

Acknowledgments

I would like to express my deepest gratitude to my advisor Sandy Pentland who has not only guided me in the process but also changed the way I think and approach the world's biggest problems and questions. I'd also like to thank Brian Sweatt for all of his help and advice throughout this whirlwind of a year. He has been a mentor in more ways than one. I'd like to thank all of the other researchers in the Human Dynamics Group who have inspired me to think creatively through their fascinating and creative research. I'd like to thank my friends and family who have supported and encouraged me throughout my entire academic journey. I'd like to thank my parents for their never ending provision of love, support, guidance, and prayer. Finally, and most importantly I'd like to thank God for the skills and support network I've been blessed with to make the most out of every opportunity He has given me.

Contents

1 Introduction **13**

1.1 Motivation and Background 13

1.2 Goals 14

2 Architecture **15**

2.1 Overview 15

2.1.1 Android 15

2.1.2 Backend 17

2.1.3 Web 19

3 Implementation **23**

3.1 Features 23

3.1.1 Probes 23

3.1.2 Partial Updates 24

3.1.3 Web 25

3.2 Challenges 26

3.2.1 Scheduling 26

3.2.2 Concurrency 30

4 Applications **35**

4.1 Real Time 35

4.1.1 Overview and Functionality 35

4.1.2 Algorithms and Implementation 35

4.2 Replay 37
4.2.1 Overview and Functionality 37
4.2.2 Algorithms and Implementation 37

5 Add-ons **41**
5.1 Heat map 41
5.1.1 Overview and Functionality 41
5.1.2 Implementation 41
5.2 Prediction 42
5.2.1 Overview and Functionality 42
5.2.2 Implementation 42

6 Testing **45**
6.1 Overview 45
6.1.1 Phase 1 45
6.1.2 Phase 2 46
6.1.3 Phase 3 47
6.2 Implementation 48

7 Next Steps **51**
7.1 User Testing and Deployment 51
7.2 Improved Probe Scheduling 52
7.3 Extending the Framework 54
7.3.1 Adding New Probes 54
7.3.2 Displaying New Data 55
7.4 iOS Extension 55

A Figures **59**

List of Figures

2-1 Funf probe schema 21
2-2 Example funf entry 21

4-1 Answer list structure for the realtime application 40
4-2 Answer list structure for the replay application 40

7-1 Example probe settings 56
7-2 Example value selector for new probes 56
7-3 Example value selector for new probes 56
7-4 Example button interface for new probes 56
7-5 Example callback function for new probes 57

A-1 Original upload routine 60
A-2 Upload routine using only direct upload 61
A-3 Upload routine using both safe and direct upload 62
A-4 Complete, end-to-end cycle 63
A-5 Full cycle when testing 64
A-6 Screenshot of the realtime application 65
A-7 Screenshot of the replay application 66

List of Tables

6.1 Test Results for User Simulation 48

Chapter 1

Introduction

Chapter 2 describes the architecture. Chapter 3 describes the implementation, design, and challenges. Chapter 4 provides an overview of two example applications. Chapter 5 describes additional features needed for a real world environment. Chapter 6 provides an overview of the testing procedure. Chapter 7 describes next steps and future work.

1.1 Motivation and Background

openPDS along with the funf framework for Android has been designed and developed to passively collect data from a variety of sensors on one's mobile device and store the information in a personal data store. Using this system a person can then choose to share his data with other applications based on his own privacy concerns. This is a powerful model that gives control to the user. The goal of this project is to develop an application that leverages the data collected through funf and openPDS to create visualizations of real time and historical data. The visualizations will combine geospatial data with a variety of other data sets such as activity information and Bluetooth interactions. The system will support viewing the incoming data sets in real time or viewing the aggregate information over a specified time window. A possible area for further research using the application will focus on how social interactions are affected in light of particular conditions such as season, weather, time of day, and

even emergency.

1.2 Goals

The system will be designed to both visualize individual, personal data as well as aggregate data created by all users that have opted into the system. Aggregate data will be visualized in two ways. The first, the more simple of the two, will show all of the historical data collected for all of the users in a particular region. The second will show the real time data being collected by the mobile devices. Each provides different insights into the way people move and interact.

Chapter 2

Architecture

2.1 Overview

The end-to-end architecture features three main environments, the Android, mobile client, the backend server, and the web interface. Each layer of the pipeline, not only involved integration with the openPDS framework but also included several other technologies, a few of those specific to this project are described in the following sections. Each of the environments played a role in the complete cycle from the passive collection of a single user's data on a mobile device to the presentation of the aggregate geospatial data from multiple users in a browser application. Figure A-4 shows a diagram of the complete process.

2.1.1 Android

The choice between Android and iOS was not a difficult one in this case. Many of the technologies needed to begin development of the application described had already been developed for Android. Two of these include funf and openPDS, which are described in more detail in the following sections. Due to the access to passive personal data through Android applications, the initial choice was clear. On the iOS platform access to many types of data is restricted, and it is not even possible to gain permission or authorization by a user. Although all of the personal data made available through

Android API's is not of the highest importance, access to communication data such as SMS and call logs is restricted on iOS. These two sources of data are highly significant due to the insights they can provide about ones social interactions and connections. Although much easier to overcome, another technical obstacle to iOS development are the limitations and restrictions placed on applications running in the background. More information regarding the development of the application in an iOS environment can be found in section 7.4.

openPDS Integration

openPDS is a significant part of this project and required integration at all levels. It also has many implications outside of the scope of this project and is looking to be used to quantify social structures, interactions, and activity in order to better understand how a society or group of people interacts, makes decisions, or operates at the basic level. openPDS relies on many technologies two of the most prominent being Django and OpenID Connect.[5] Although most of the technology has been designed and built as a backend solution, an openPDS client has also been developed using the Android framework in order to more easily integrate applications with the platform.

With respect to architecture and storage openPDS stores each user's information in a separate database termed a Personal Data Store and gives the user control over which applications can access the data. This model provides the user with privacy but also the ability to more powerfully leverage the data collected by his applications. For example, the user has the opportunity to give all of his applications access to all of the location data that has been collected over time rather than each application collecting this data only when it is running. This is particularly useful for new applications that currently have to undergo a bootstrapping process of collecting data after initial installation. Using this model all of a users historical data could be accessed instantly.[5]

In order to provide the guarantees of privacy and fine-grained control, data used by applications is not provided directly. In other words, in most cases applications

are not given access to the raw data collected about an individual. Instead each application must predetermine the answers it is seeking based on the data provided by the users. Examples of this include: knowing whether or not a user has entered or exited a particular geo-fenced region rather than getting access to the raw location data in the form of latitude and longitude.

Funf

A framework developed originally at the MIT Media Lab, funf is a premier data collection framework for Android. Funf has been designed to collect a large variety of data types passively on Android devices. It uses a structure based around probes that can easily be configured for the needs of the developer.[3] Each probe has a variety of settings relevant to the particular data being collected. The data collected is stored in a structured format.[4]

As a result, the data fits well in a database system such as MongoDB where keys and values can be determined on the fly and do not even have to be equivalent across one particular data set. The flexibility of MongoDB combined with the extensiveness of funf makes for a natural partnership between the two frameworks.

In order to incorporate funf with openPDS a project was developed to make integration into the openPDS environment simple. The project is also open source and was developed at the MIT Media Lab. It is an extension of the previously mentioned openPDS client developed for Android devices and was also used in this project.

2.1.2 Backend

Due to its heavy integration with openPDS, the backend architecture had little room for adjustment or new technologies. From authentication, to storage, to ultimately content delivery, each step of the process was dictated in some way by the design of openPDS. For more information on the algorithms used in aggregation and computation see section 4.1.2 and section 4.2.2.

openPDS Integration II

The backend architecture again relies heavily on openPDS. Each upload by an openPDS client must be authenticated by the openPDS registry server where the application and the user were registered. The data is then stored in a user's PDS and later used in computations that can be accessed by properly authenticated applications. The following sections outline the unique database structure required of openPDS and the task management system used to create the data that can be used by applications. Although proper authentication is a significant part of openPDS and its privacy preserving architecture, it had little influence on the design and implementation of the framework. As a result, this document does not go into more detail about the specific challenges and advantages associated with its design and inclusion in openPDS.

MongoDB

The current version of openPDS uses MongoDB as primary storage. In the openPDS architecture each PDS is a separate database. Due to its distributed nature this model increases security and also allows greater control on privacy. Each users PDS contains two relevant collections (similar to tables in a SQL environment). One is labeled $funf$, the other *answerlist*.

The $funf$ collection contains all of the data collected through funf on a user's device. Although MongoDB is a key value store and makes no restrictions on schema, each entry collected and created through funf has three fields. The schema can be seen in figure 2-1. An example entry can be seen in figure 2-2.

Data stored in the $funf$ collection is meant to be used in computations and analysis but not directly given to applications that the user has authorized. On the other hand, the *answerlist* collection is made of computed results using the data from multiple $funf$ collections. For instance, the applications described in section 4.1.1 and section 4.2.1 aggregate and compute answers using data from all of the users on a particular registry server. Although there are no restrictions enforced by the

database, traditionally the *answerlist* collection data has a list format because it contains answers for multiple users on the system.

Celery

For this application celeryd is used in order to run multiple workers that will compute the data stored in each user's *answerlist* collection. Celerybeat is used as the scheduler and task creation engine. Jobs are managed and distributed among the workers by the celery engine.[8] For more information on scheduling see section 3.2.1.

2.1.3 Web

The description of the two browser applications created using the described architecture can be found in section 4.1.1 and section 4.2.1. The web architecture used a variety of technologies including: integration with openPDS, WebGL, the Google Maps API, and BackboneJS. While the use of WebGL and the Google Maps API were specific to the particular web applications many of the other technologies used and the overall structure for using data for visualizations must be replicated in some fashion in order to create applications that integrate deeply with openPDS.

openPDS Integration III

Although openPDS may have the smallest impact on the web interface within the entire system, it still has an effect on how the data is accessed and provided to an application. In order to view real time updates in a browser, an *answerlist* collection for an individual user is polled at regular intervals. When new data arrives it can then be used by the application in visualizations or computations. In this particular setup the application is stored on the server. As a result, a user must be authenticated in order to view the application. Another alternative model requires that the application be authenticated to gain access to a user's *answerlist* collection before using the data in any type of visualization or computation. The former setup was chosen due to the ease of integration and relevance in testing the end-to-end architecture of openPDS.

WebGL

WebGL is a web framework designed for in browser graphics. It uses the GPU for rendering and processing visualizations saving the CPU valuable computation cycles. Because of this design, it can more easily visualize large amounts of data. WebGL can draw thousands of points at once, but also perform computations while coloring, shading, or animating the points.[6] Although WebGL does provide the developer with fine-grained control and is a powerful tool, it has a high learning curve that makes quick visualizations or graphics time consuming. Even though it was selected as the tool for visualizations in the applications described in section 4.1.1 and section 4.2.1, it can easily be replaced with other frameworks, especially considering the relatively small number of users used in testing.

Google Maps API

The Google Maps API was originally used to not only display location but also to create heat maps of the data collected through funf and the openPDS client. WebGL was chosen in favor of the Google heat map API as a result of the lack of flexibility and the ability to only view up to 1000 individual points when using the Google heat map services.

The current architecture uses the Google Maps API along with an open source project called Canvas Layer. The styled map was created using a Google maps style creation wizard.[2]

BackboneJS

BackboneJS creates an event-based environment. First the database is polled for a particular query. Upon returning asynchronously with the results the environment is notified and the results in this case are then used to load the array buffers that will be used to draw the points on the map using WebGL. BackboneJS causes few problems or idiosyncrasies except for the tricks of programming in an asynchronous environment.

```
1: {
2:     key : string,
3:     value : dictionary,
4:     timestamp : datetime
5: }
```

Figure 2-1: Funf probe schema


```
1: {
2:     "_id" : ObjectId("5343b3fa1d41c8c7e596316d"),
3:     "key" : "edu.mit.media.funf.probe.builtin.ActivityProbe",
4:     "value" : {
5:         "timestamp" : 1396945894.377,
6:         "total_intervals" : 5,
7:         "high_activity_intervals" : 0,
8:         "low_activity_intervals" : 0
9:     },
10:    "time" : 1396945894.377
11: }
```


Figure 2-2: Example funf entry

Chapter 3

Implementation

3.1 Features

3.1.1 Probes

The combination of probes selected, activity, location, SMS, call, and Bluetooth make up a large selection of the probes used to compute the social health score, an algorithm designed in the Human Dynamics Group at the MIT Media Lab to better quantify and understand a user's social interactions using passive data. Due to the ease of incorporating each of these in the Android application and the real time application described in section 4.1.1, the final step towards real time analysis would simply involve integrating the social health algorithm or any other algorithm, for that matter, into the task file where the real time and replay algorithms are located (see section 4.1.2 and section 4.2.2).

Accelerometer

The accelerometer probe was selected due to its ease of use and potential insight into the movement and activity of users. It responds quickly to inputs from the user, and the movements can easily be replicated for testing. The testing allowed for a gradient of inputs from low motion to high motion.

Location

The location probe was also useful from a practical sense. From visualizations to analysis, being able to correlate data collected with a particular location has many uses. The location probe was not great for responsiveness testing seeing that testing the responsiveness of the probe would require physical movement in large distances. It can be argued that in an urban environment where most people are walking the frequency of location updates as opposed to other probes has a much lower requirement.

SMS and Call

The SMS probe looks at the incoming and outgoing SMS in a given time window as the call log probe measures call activity. These were both chosen due to their inclusion in the Social Health Score calculation referenced in section 3.1.1 and general insight into social patterns.

Bluetooth

The Bluetooth was chosen because of its insight into the number of mobile devices or machines within range of a particular user. This can be used to get an idea of the number of people a user might come into contact with at any given location.

3.1.2 Partial Updates

In order to combat some of the effects caused by conflicts and increasingly long update times as the number of users increases, the method of partial updates was implemented. In the case of partial updates, users are split into buckets based on their unique user ID. Instead of checking for uploads from all of the users registered with the application at once, only one bucket is checked during a run of a celery task. Part of the reason this setup was implemented was to greatly reduce the time between visual updates in the browser. Even if a quarter or a fifth of the data is updated, it still allows the user to get a feel for the way updates are occurring within a given region or environment.

The update routine created does little in light of redundancy. Currently there is no confirmation of an update occurring. As the users are split it into buckets, if something goes wrong while a particular bucket is being updated, the update for that bucket is just skipped. In order to make consistency more of a priority, a more complex update routine would have to be implemented. This routine would not only check to see if a celery process was in the progress of updating a bucket, but it would also allow the process to record whether or not the update was successful. Choosing the next bucket to update would be a matter of checking to see the number of updates that have occurred for each bucket over the last minute or two and then picking the bucket with the least number of successful updates to occur recently. Again this is an alternative implementation, but not the implementation of choice, due to the lack of concern with redundancy or consistency. As long as updates are occurring, it is a minor issue if one or several group updates are missed. Overall when testing, partial updates by buckets performed well and did relieve many of the issues with scheduling that were beginning to occur as the number of users increased.

3.1.3 Web

BackboneJS is used for a lot of the heavy lifting. It provides an intuitive interface to accessing the data stored in MongoDB. The results from the queries return asynchronously. Backbone checks for updates at regular, frequent intervals, stalling only for conflicts accessing a particular database. More information on conflicts can be found in section 3.2.2. Once the results are returned the data is then used by WebGL. WebGL relies on an array buffer of points. For the particular application three values are required for each point, x, y, and z. The x and y coordinates correspond to the latitude and longitude of the particular device, but the z value is determined by the particular selector. In the example application, selectors include SMS, call, bluetooth, and accelerometer. After the appropriate values have been loaded in the array buffer WebGL is then used to render the points on the Google map.

3.2 Challenges

3.2.1 Scheduling

Scheduling proved to be one of the greatest challenges in the development of the system. Each part of the system has various factors that cause discrepancy between the theoretical limits and the practical limits of the architecture.

There are at least four configurable schedules that affect the complete cycle time for a single user. A complete cycle includes data creation and collection, the upload of data to the users PDS, computation on the users data, and finally display in the real time application. Figure A-4 shows a diagram of the complete cycle. With each of those steps corresponding to the following variables:

$$A = Creation$$
$$B = Upload$$
$$C = Computation$$
$$D = Display$$

The worst-case cycle time is

$$B + 2C + 2D$$

This is the frequency of phone uploads combined with twice the time for both computation and display due to the fact poor timing could result in a previous step of the cycle not being completed before the next one begins.
With Times:

$$A = 9$$
$$B = 10$$
$$C = 4$$
$$D = 1$$

The worst-case total time is

$$10 + 2 * 4 + 2 * 1 = 20$$

Regular worst-case time is

$$B + C + D = 15$$

There are two key alternatives in reducing the worst-case time significantly, reducing time C and reducing time B. Time C is a much simpler fix and really only depends on the resources available on the particular PDS machine. Switching to a frequency of 1 second instead of 4 is simple with few other variables impacting its value

Although time B seems like it can also easily be reduced, the original equation should actually more accurately be:

$$Z + 2 * C + 2 * D$$

where

$$Z = A > B \, ? \, ceil(A/B * B) : B$$

As a result it is best to operate under the constraint that

$$A < B$$

meaning

$$Z = B$$

Therefore in order to reduce Z, both A and B must be reduced. Reducing B to a lesser value has proved to be trivial, but the same cannot be said for reducing the value of A. A is actually made up of two values, the frequency of the accelerometer probe and the length of time that the probe collects data during each scheduled interval. The following sections describe in more detail the challenges and complexities involved in each step of the process and provide a closer look at optimizing the previous equation.

Probe Scheduling

As previously stated the timing of the accelerometer probe is actually dependent on two values, the frequency of the accelerometer probe and the length of time that the probe collects data during each scheduled interval. As an example, the accelerometer probe can collect data for eight seconds every nine seconds. openPDS was originally designed and tested to collect data at much more infrequent intervals. One of the most common settings is to collect data for thirty seconds every fifteen minutes. As a result, there are issues that arise when using the probes at such a high frequency that have gone previously unnoticed. Initial tests using a probe window on the order of five seconds created an error that was hard to understand, but the results were undesirable. Data collection changed so that it seemed as though the probe never reset and was constantly collecting data. This is not a huge issue in the grand scheme of things but would and did lead to problems at the other end of the pipeline that made certain very reasonable assumptions about the data being collected and its format.

Phone Scheduling

Although there seemed to be no theoretical limits on the frequency of phone uploads, other than battery life, and total upload time, the practical limit was between nine and ten seconds with the original implementation of openPDS client and the funf framework. The current version of openPDS client uploads a small sqliteDB of data collected through funf based on the intervals set for each probe in the application. This works well for most openPDS applications, but when considering a real time framework that focuses on small frequent uploads, the overhead added by the current pipeline is unnecessary. The pipeline has been configured and designed for redundancy and safe storage of semi-frequent data collection. Safe storage is not a high concern in the chosen architecture due to the frequency of collection. In the case of real time data, a lost upload would only leave a five to ten second gap in a users data while in the case of the semi-frequent uploads (e.g. five minutes), a ten minute gap would

result. Two missed uploads would result in a fifteen minute gap. Seeing that the application described in section 4.1.1 is focused more on visualization than analysis, a missed upload will not show up in the visualization even if stored safely. When considering analysis, the best solution is a combination of direct uploads to openPDS, safe storage in the openPDS client database, and infrequent openPDS client database uploads in order to make sure data was not lost in the process of being uploaded. Testing of the applications involved both the original upload routine and the more streamlined process, excluding safe storage. Figure A-1 shows the original routine. Figure A-2 displays the adjusted routine with only direct uploads. Figure A-3 shows the combination of the two.

Backend Scheduling

Backend scheduling is one of the most crucial parts of the implementation but also one of the most open-ended. When considering new visualizations or new algorithms for analysis, this is where a lot of the complexity in scheduling comes in. The backend scheduling involves selecting the appropriate data depending on some requirement such as the most recent for example and using that data in a predetermined computation. Total compute time includes the time required to aggregate the appropriate data, the time required to run the designed algorithm, and finally the time required to update each users personal data store. The time required to run the designed algorithm is probably the simplest of the three. It is in proportion to the complexity of the algorithm and has low variance over constant testing. The time required to aggregate the data and the time required to update the database are related to how often the updates occur and how often the browser checks for updates because of the locking mechanism of the database system used. Because of the relative simplicity of the algorithms being executed, for the applications described in section 4.1.2 and section 4.2.2, a schedule with a high frequency was the best option.

Browser Updates

Browser updates are the last piece in presenting the data to the user, possibly the most trivial in light of scheduling. Currently the architecture is constantly polling for updates at a high frequency. This is partially due to the low cost of polling a single user's PDS and partially due to the use of Apache as the web server of choice. A deployment using NGINX has also been considered due to the fact that it requires less memory and can push updates to the client, instead of having to respond to constant polls. The architecture of the application is not heavily dependent upon the use of Apache over NGINX, so this migration is still within reason.

3.2.2 Concurrency

The current version of MongoDB uses a per database granularity of locks rather than a global lock as in previous versions.[7] This makes scheduling more convenient yet still non-trivial. openPDS stores the data of each user in a separate database. This is again an advantage over more traditional systems that use one large database for multiple records for multiple users, but it still has complexities of its own. The setup described in the application requires updates to a user's PDS from a mobile device, the aggregation of the data in the PDS, and an update to the users PDS in the form of an answer list determined by a particular applications requirements. For instance, the application described in section 4.1.1 requires the most recent updates for all of the users. The answer list in the users PDS is then read or polled by the browser when looking for updates. http://docs.mongodb.org/manual/faq/concurrency/ In summary there are four types of interactions with a users personal data store:

1. individual writes
2. group reads
3. group writes
4. individual polls

The reads and writes described as "group" are strictly scheduled and each user's database is read or wrote in sequence. MongoDB allows multiple reads to a particular database but only allows one operation to write at a time. The single operation, write lock also restricts all read access.[7] This style of read-write access has the highest chance of conflict and also the highest impact in light of individual writes conflicting with the group reads or the group writes. Although the effects of lock conflicts range from low to high impact, testing revealed that even in the case of those that were regarded as high impact conflicts didn't seem to have unmanageable effects on timing or scheduling when testing with ten users.

There are six types of conflicts. Four of them have a significant to reasonable likelihood. Two of them have significant impact on the timing of the application, specifically the real time updates.

individual writes - group reads (high likelihood, high impact)

Like individual writes and group writes, these conflicts occur often based on the number of users running the particular application or updating their respective PDS. These conflicts can be especially problematic if a user's PDS is updated by more than one client application. Every update has to touch the same database. Eventually, longer than expected wait times are inevitable after a certain threshold of users is reached.

individual writes - group writes (high likelihood, high impact)

These have a high likelihood due to the fact the individual writes are scheduled to occur constantly at a high frequency for each user. While a group write is occurring it is possible that multiple writes from individual mobile devices can conflict. Depending on the number of clients this could seriously effect response times and update times that an individual sees in their browser. It is likely that this conflict will occur each time a group write occurs after a certain threshold of users has been reached.

individual writes - individual polls (moderate likelihood, low impact)

These can occur frequently if a user is checking for updates through the browser and the users device is constantly uploading data as it should. There is a bit of timing involved in order for this type of conflict to occur, but by using the schedules determined for uploads and update checks the worst-case number of conflicts per minute could be estimated. Because individual polls occur at such a high frequency, any conflict has low impact.

group writes - group reads (low likelihood, moderate impact)

These occur due to multiple celeryd daemons running tasks simultaneously, either the same tasks or different tasks. It can be argued that the likelihood is low compared to some of the more common conflicts. The impact can vary based on the timing of the two tasks. As the processes are iterating over each user's PDS they could potentially conflict multiple times depending on the time it takes to read a PDS and the time it takes to update one with new computed values.

group writes - individual polls (high likelihood, low impact)

In order for a browser to get updates, a particular user has to be authenticated. Even if the data being displayed is collected and computed over multiple users, it has been approved so that the data is stored in the *answerlist* collection of a single user. Thus updates to a single browser only poll a single PDS. Tests have shown that setting a high update frequency returns the best results even with conflicts having a high likelihood. In the case of group writes conflicting with individual polls, a conflict implies that the user's PDS is being updated while the browser is looking for updates. If the update keeps the browser from immediately being able to check for an update, the time it takes to wait for the update is trivial compared to waiting for another cycle. Since the time it takes for an individual poll is trivial, in the case of the alternate conflict, the group write would occur soon after the initial conflict time and the individual poll would find fresh data on the next cycle.

group writes - group writes (likelihood varies, moderate impact)

These only occur if two celeryd processes are running at the same time. The likelihood depends on how long each task takes to run. This conflict occurs if a daemon scheduled to complete a task took longer than expected either to collect or begin writing due to other conflicts and the second daemon "caught up". Another possibility is a conflict between multiple tasks one that runs every 5 seconds with a short compute time and one that runs every five minutes with a longer compute time.

Chapter 4

Applications

4.1 Real Time

4.1.1 Overview and Functionality

The real time application features a Google map with various points that correspond to users with a mobile device running the client version of the application collecting data. Each point on the map changes location and color based on the data that is being collected in real time for each user. The color of each point is determined by the particular probe that has been selected in the browser application. Possibilities include: activity, SMS, and call. The application uses WebGL, and as a result, it is best viewed in a Google Chrome web browser. Other browsers with WebGL support include Safari (in Developer mode) and Firefox. Although updates are said to occur in real time, each user's corresponding point responds to changes in value on average every 10-15 seconds. This is due to the complexities of scheduling described in more detail in section 3.2.1. Figure A-6 shows a screenshot of the application.

4.1.2 Algorithms and Implementation

Most of the algorithmic complexity associated with the real time application depends on generating the appropriate answer list in real time. The answer list contains the most recent updates for each probe for each user. The structure of the data can be

seen in figure 4-1 where the length of the answer list is equivalent to the number of users.

Although described as the most recent updates for each user some of the values stored for the probes are totals or aggregates rather than singular entries. For instance, the location probe stores the most recent location uploaded by the user while the SMS probe records the total number of SMS received or sent in the last two minutes. Data from probes that are aggregates is not susceptible to becoming stale since it will not show up in a query to get data from the previous two minutes. Seeing that last entry data would be susceptible to becoming stale, queries are limited to going two minutes in the past.

In order to handle the differences between last entry probes and aggregate probes a format function was created for each that when supplied the most recent entries for the corresponding probe, the correct computation would be performed.

Algorithm 1 Example function for formatting call log data

```
1: procedure FORMATCALL(callData, count = None)
2:     if callData == None then
3:         return None
4:         item = {}
5:         item[count_key] = count
6:         return item
```

Algorithm 2 Example function for formatting accelerometer data

```
1: procedure FORMATACTIVITY(activityData, count = None)
2:     if activityData == None then
3:         return None
4:         item = {}
5:         item[ACTIVITY_HIGH_KEY] = activityData['value']['high_activity_intervals']
6:         item[ACTIVITY_LOW_KEY] = activityData['value']['low_activity_intervals']
7:         item[ACTIVITY_TOTAL_KEY] = activityData['value']['total_intervals']
8:         return item
```

4.2 Replay

4.2.1 Overview and Functionality

The replay application uses many of the same technologies as the real time application and also has many of the same features. Again each user is represented by a point on the Google map that changes location and color based on the value computed for the selected probe. The difference in the case of the replay application is the ability to move through different snapshots of the data and thus the real time application by adjusting the slider positioned at the bottom of the screen. The slider can be adjusted manually or automatically using the playback option by selecting the button labeled play. With the option to choose where to begin and playback, the replay application allows one to get a more comprehensive sense of the changes that occur in a given region over a selected period of time. Although the interface does not allow for easy adjustment of the day or time interval used, it does allow one to select from several different hours to step through in one minute intervals. Also the backend server can be easily adjusted to aggregate data from a particular day or for different interval sizes. Figure A-7 shows a screenshot of the application.

4.2.2 Algorithms and Implementation

In order to be able to move through the snapshots used in the replay application, an answer list was created with the set of points for each selected time. Seeing that there are an infinite number of possible snapshots in time that can be taken from a given dataset, a reasonable interval was chosen that could be used to represent a bucket. For each bucket, for each user, for each probe data is aggregated and saved.

Although it would be possible to collect the necessary data in this way by first iterating over the buckets or time intervals, subsequently iterating over the users, and finally iterating over the probes to query the appropriate data using each of the corresponding values for time interval, user, and probe, the run time would lack efficiency due to the number of individual queries to the db. The total number of

queries would be $num_buckets * num_users * num_probes$. The pseudo code for the algorithm would be something like the following:

Algorithm 3 Example algorithm for filling time buckets

```
1: procedure FillBuckets(buckets, users, probes)
2:     for bucket ∈ buckets do
3:         for user ∈ users do
4:             for probe ∈ probes do
5:                 start = bucket.start
6:                 stop = bucket.stop
7:                 db = user.db
8:                 entry = db.funf.find(start, stop, probe)
9:                 bucket.probe.value = entry.value
```

Using 10 second intervals over a period of 24 hours is the equivalent of $(24 * 60 * 60)/10 = 8640$ buckets. Even if there were only one probe and one user, this would be an unrealistic number of queries to the database. In order to account for this, the above algorithm was restructured to reduce the number of queries to the database. Although it added much complexity to the overall algorithm, the performance increases made it well worth the loss off simplicity.

The revised algorithm queries the entire time period (where $time_period = num_buckets * bucket_size$) for each user for each probe and then inserts the data in the appropriate bucket. If data is missing, it is supplanted with data used from a previous bucket unless the data from the previous bucket has been deemed stale. Otherwise the bucket with missing data is filled with the default data for the particular probe. The pseudo code for the new algorithm is as follows:

Algorithm 4 Example algorithm for filling buckets efficiently

```
1: procedure FillBucketsEfficient(buckets, users, probes)
2:     start = time_period.start
3:     stop = time_period.stop
4:     for user ∈ users do
5:         for probe ∈ probes do
6:             entries = db.funf.find(start, stop, probe)
7:             for entry ∈ entries do
8:                 bucket = find_bucket(entry.time)
9:                 bucket.probe.value = entry.value
```

Using the revised algorithm the number of queries to the db is $num_users * num_probes$. Over a period of twenty-four hours this is a decrease of almost 10000 fold. The final structure of the answer list can be seen in figure 4-2 where the length of the answer list is equal to the number of users.

```
1: [
2:         {
3:                 probe1 : value,
4:                 .
5:                 .
6:                 probeN : value
7:         },
8:         .
9:         .
10:        .
11: ]
```

Figure 4-1: Answer list structure for the realtime application

```
1: [
2:         {
3:                 0 : {
4:                         probe1 : value,
5:                         .
6:                         .
7:                         probeN : value
8:                 },
9:                 .
10:                .
11:                .
12:                num_buckets : {
13:                        probe1 : value,
14:                        .
15:                        .
16:                        .
17:                },
18:        }
19:        .
20:        .
21:        .
22: ]
```

Figure 4-2: Answer list structure for the replay application

Chapter 5

Add-ons

5.1 Heat map

5.1.1 Overview and Functionality

In order to better account for user privacy when true personal data is being used with the framework, the Real time and Replay modes both contain heat map modes. This displays an aggregate picture of the particular probe being displayed rather than the more fine grained picture of a users data described in chapter 4. The heat map decays slowly over time, but is replenished if the current data has not become stale or new data is found on the server. Although featured as an add-on, heat map mode not only has the bonus of being privacy preserving it also models reality well, with the effect of a phone call, SMS or other event decaying slowly over time rather than sharply over an instant. The intensity of the heat map can also be adjusted so that large or small amounts of data aren't overwhelming or underwhelming respectively.

5.1.2 Implementation

The heat map overlay is based on the codeflow library.[1] In order to establish the desired effect outlined in the previous section, decay rates are calculated in the browser and on the server. Seeing that replay mode, only features snapshots of various times throughout a particular day, only base values needed to be calculated for each location.

Since the real time mode features a more dynamic display of the data, update values were computed in addition to the base values at each location. The update values take into account the decay rate being displayed in the browser and previously determined stale threshold for events.

5.2 Prediction

5.2.1 Overview and Functionality

In order to better account for the infrequent updates standard to most openPDS applications, the prediction add-on adjusts user locations based on a user's historical data. Most openPDS client applications are based around probes collecting data at fifteen-minute intervals and that data being uploaded once every hour in the best case. This is unideal for a real time platform unless there are roughly four thousand users. This is due to the fact that the most responsive, ideal picture from the platform occurs when the ratio of the number of users to the number of seconds between uploads is approximately 1:1. When the ratio diverges greatly from this value in favor of increasing time between uploads, the real time effect is lost. This add-on uses the users past data to make guesses about the users current location and gradually represents the predicted location in browser over a predetermined period of time.

5.2.2 Implementation

The current prediction algorithm is far from robust, but can be easily upgraded to a more accurate algorithm when necessary. The current algorithm finds the start of the current hour and then makes predictions about the future locations in fifteen-minute bucket sizes. The future locations are currently computed by taking the centroid of all previous locations during the particular time window. The algorithm performs well for users with highly rigid and repeating schedules, but offers no real insights or value if a user's movements do not show either of these features from week to week. Again due to the focus of this project being on the design of the framework rather

than location prediction, the current algorithm is more than satisfactory.

Chapter 6

Testing

6.1 Overview

6.1.1 Phase 1

Initial testing of the framework involved a single phone running the Android application and a single server that ran both the registry server and the PDS server. The server had only 512 Mib of memory. It ran two celeryd daemons scheduled to check for newly uploaded data and compute answers approximately every four seconds. Uploads from the testing device occurred approximately every nine seconds.

In order to test the results the accelerometer probe collected data as the device was shaken at various frequencies. In order to record the time it took for a full cycle, a timer was set while waiting for updates to occur in the browser. A diagram of the full cylcle can be seein in figure A-4.

One partially hidden result of the architecture used for testing, is that the server had to respond to two requests every time data was uploaded because the registry server and PDS were located on the same machine. The first time the server had to authenticate the upload. The second request was to handle the actual upload of the data. Although the extra ping to the server in order to authenticate the request does not affect the number of requests by an order of magnitude, it does require a significant amount of resources, especially in an environment with low memory.

Performance in the described environment was unreliable at best. Although there were moments where the full cycle from upload to display occurred as expected in the middle of the estimated time range, there were many cycles that took much longer than the theoretical maximum. Ultimately the delays were a result of operating in a low resource environment and occurrences of many of the conflicts described in section 3.2.2. As a result of MongoDB using memory to quickly access data during queries, running on a machine with so many concurrent processes and little memory made each query take that much longer. Conflicts caused delays that lasted on the order of seconds rather than microseconds or even milliseconds.

6.1.2 Phase 2

After the initial testing, migration to a machine with increased resources was necessary. The machine of choice had around 3.1 GB of memory, an increase in six times the memory of the first machine. With the additional memory available, four celeryd daemons were used in the process of checking for new data and computing answers. It should be noted that it was not possible to run more than two daemons on the original machine due to the lack of resources. Again the full cycle included data being uploaded from a device, the data being queried and then used to compute an updated answer list, and finally the updated answer list being used to update a visualization in a browser with the browser application polling for updates every second. A diagram of the full cylcle can be seen in figure A-4.

When using similar settings as described with the initial environment, except for the use of two times the number of celeryd daemons, the full cycle was much smoother and also much more reliable. Most of the cycles occurred within the expected time range, with a minor portion of the updates occurring outside of the theoretical maximum. Another difference in the environment was the use of a remote registry server. As a result, each request opened a port on the PDS machine only once. The low memory machine served well as the registry server and freed up valuable resources on the more powerful machine.

This phase of testing revealed that with the initial version of the application the

most reasonable and reliable upload time was every eight seconds. The accelerometer probe ran every seven seconds and collected data every six. Anything less than this began showing the previously mentioned undesirable outcomes. Section 3.2.1 goes into more in depth about changes to the Android application that would allow for quicker upload speeds. Although there are adjustments that could be made, the question becomes what is the necessary or desired frequency for uploads given the plan to scale and have a large number of users. Updating each users data every five seconds even may not be very rewarding or useful. If instead updates for a single user occurred every fifteen seconds and were mixed in with the updates of 100 or 1000 other users, valuable resources could be saved and a similar experience had when viewing updates in the application.

6.1.3 Phase 3

The last stage of testing involved increasing the load on the server from a single user to various magnitudes of users. Seeing that it would be difficult and almost impractical to acquire the number of devices necessary, go through the setup and testing procedure, and make minor changes on each device through the android Google Play Store or whatever means necessary, it was decided to replicate the upload of data using other machines. As described in section 2.1.2, the upload process includes both authentication and upload. Seeing that in the chosen architecture authentication occurs on a remote server, it seemed as though inclusion of the authentication process could have been potentially insightful but not necessary. Instead an extra second or two could be added to the upload times in order to account for the missing round trip time. Testing began by simulating ten followed by simulating twenty, fifty, and multiple hundred users sending updates to the server at regular intervals. An example of the testing cycle can be seen in figure A-5. The data created by each user then went through the same process of aggregation and display that data uploaded by a device using the openPDS client would. A more thorough description of the testing architecture and logic used can be found in the following section.

Uploads occurred once every $(num_users* \sim 1.5) + 3$ seconds, so with the lowest

Table 6.1: Test Results for User Simulation

Simulated Users	Updates per Minute	Browser Update Frequency (secs)
10	3.3	3
20	1.8	3
50	.77	4
100	.39	4-5
200	.19	3-7
300	.2	5-8

number of users updates were occurring several times a minute. While with the most extreme number of users, updates were occurring once every ~8 minutes. With the initial testing, updates and responsiveness were as expected. Updates to the browser occurred consistently once every three seconds for the first two testing routines. As seen in table 6.1, when simulating two hundred and three hundred users, the browser updates occurred once every 3-7 and 5-8 seconds respectively. This increase in browser update time is most likely a result of one or more of the database conflicts described in section 3.2.2. Future tests would involve an increase in concurrent updates when simulating one hundred and two hundred users.

One thing to note is that in the midst of testing the reliability and the capabilities of the server, the effects of increasing storage over time was not included in the testing. The described environment more closely resembled a new environment, one with a larger number of new users and low amounts of data. Although this does not affect most of the operations, an increase in data may affect query times based on the complexity of the query. This aspect of the application has not been extensively tested and would need further time if large-scale deployment is in consideration. Although, ultimately an increase in memory is almost always enough to combat most of the issues involved in increased query times.

6.2 Implementation

In order to test the limits of the PDS server in conjuntion with the real time application, a testing framework was developed to simulate multiple users running the

openPDS client and funf frameworks. The framework consists of two classes, user and uploader, and a set of helper functions for creating and authorizing new users. The functions for creating and authorizing new users, streamlined the original process which involved creating the users manually through the openPDS portal. Instead a base username and number of users can be specified. In this case the users will be created authenticated and all necessary and potentially necessary future information is stored locally. This includes *uuids* and *bearer_tokens* respectively.

A user instance is initialized with a unique user ID that has already been registered on the registry server and it is randomly assigned a starting location within the selected region. The class has getters and setters for various attributes that have corresponding probes in the funf framework, and it also has a move function to change and store the user's location.

The uploader class manages a collection of users by updating the instances with constrained random values and uploading data for each instance at semi-regular intervals. The uploader class can handle any number of users as long as the appropriate user ID's have been created. Although openPDS usually goes through a process of authentication for all uploads from a mobile client, a non-authenticating endpoint was created on the server in order to minimize the tedious work of updating tokens for each user. Although authentication is a key aspect of openPDS the test results were not greatly skewed by the absence of authentication and still reflected a reasonably accurate portrayal of the resource limits because of the architecture described in section 6.1.2.

Chapter 7

Next Steps

7.1 User Testing and Deployment

One of the first next steps that needs to be taken is deployment of the application in a more strenuous environment with more users. This would increase the feedback on the real time application and would hopefully provide more insight on how battery life is affected by the application for the average user that interacts with his phone regularly. This is probably one of the biggest concerns when it comes to the practicality of uploading data to a server every five to seven seconds. The current use of MongoDB makes each upload take longer than necessary. Although the structure of the data through funf is stable, the same keys are uploaded with every upload of a value. As a result, a SQL solution is in development that would eliminate the need to constantly transfer metadata, saving space and time. Deployment would also give more insight into how people move and how often location data needs to be collected.

One could imagine the amount of battery life saved if as the data collected got more interesting, the frequency of upload increased and decreased in the alternative case. This would save a lot of battery life when the user was idle or moving slowly and allow the backend infrastructure to continue using the last updated data to relay the information. The following section goes into more detail about improved probe scheduling.

7.2 Improved Probe Scheduling

In order to optimize for memory usage and battery life one addition would be to decrease probe collection rates and upload frequencies when a particular user is normally inactive. One would expect most users to have a five to eight hour period of rest each night. It's during this time that location tracking, accelerometer data, and possibly even Bluetooth monitoring become much more predictable. In many ways predictable means stable, and stable means non-interesting. In order to possibly decrease the number of samples taken during that period of time, one could use a basic learning algorithm that evaluated potential patterns of rest, relating location, time of day, phone orientation, Bluetooth sensors and accelerometer data. Using all of these sensors it seems as though it would not be too difficult to pinpoint when to decrease samples for a majority of users.

Currently location updates occur about once every thirty seconds. If a user normally sleeps for seven hours a night, there are $2 * 60 * 7 = 840$ probes of the users locations over this time period. A probe even once every hour would still reduce the number of readings by 2 orders of magnitude. The effects on battery life would be tremendous and ultimately provide the user with a more quality experience.

One of the most useful indicators for smart probe scheduling is the accelerometer. The phones accelerometer provides valuable insight into whether a user is walking, running, or riding. As a result, it can be used to more closely direct the collection of data using funf. Thirty-second location updates might be best while a user is walking but insufficient when they are running or riding in a vehicle. It also may be a great waste of battery life and data if the user is sleeping. By constantly having access to this insight through the use of the accelerometer, overall performance in terms of the battery life but also server load could be significantly improved. Uploads to the server would be reduced based on the variety and instability of the data being collected. Current phone models use a significant amount of battery life with constant use of the accelerometer. With the current models the only value impact of running the accelerometer constantly and applying the described level of learning would be to

reduce the number of uploads to the server.

In spite of this, there are currently some phone models such as the MotoX and the iPhone 5S that have low power accelerometers that can constantly track and process movements by the user. As these become the standard for Android devices, one can expect an API to be released as with iOS to be able to respond and spin off processes based on the data being collected. This would more fully express the advantages of the method described and in most cases eliminate the need for an accelerometer schedule at all.

Currently most of the algorithms used collect data from a particular time window. The algorithm used to gather the data deemed as most recent in the real time application has a maximum window that it looks into the past. It is on the order of two to three minutes. If data is being collected and uploaded at a decreased rate eventually the user will be assumed to be non-active and will no longer show up in individual or aggregate computations or visualizations. In order to account for the discrepancy a process will have to be determined on the server to either replicate the data currently living on the server or find the associated user and account for the missing data. The former option seems as though it will be the most reasonable but will take some clever planning. Two methods of creating the data would be on-demand and pre-determined with lazy deletion.

The pre-determined would fill in the expected data for a particular time window W into the future and assume most of the algorithms would look at the current time T and backwards. Seeing that T would occur somewhere in the midst of the created data, the data that was marked with a time after T would not be analyzed or visualized. The only issue is how to delete the data if it is found that at some point T within the range W a user changed his normal pattern of movement. Deletions would have to occur on a look-ahead basis. If the newest upload at time T was as predicted, the data for the rest of time window W would stay the same, otherwise delete all data from T until *W.end* and return to a more frequent probe schedule. The cost of an incorrect prediction would be the time it takes to find and delete all of the entries replicated for the particular period.

Algorithm 5 Example algorithm using the pre-determined method

```
1: procedure PREDETERMINED(upload, user)
2:     if user.sleep == True then
3:         if user.predicted.value ≠ upload.value then
4:             delete(upload.time, user.predicted.end)
5:             user.db.funf.insert(upload.value)
6:             return
7:         return
8:     user.db.funf.insert(upload.value)
```

Another potential alternative would be to create new points on the server at every time to stale interval S. If a change in data occurred at some time T, the mobile client would upload the change and return to a more frequent probe schedule. This would greatly reduce the cost of deletion but would increase the work of the server if over some seven-hour period each user had data created every three minutes. Compared to the previous alternative the number of potential conflicts within MongoDB would be much greater.

7.3 Extending the Framework

7.3.1 Adding New Probes

Incorporating a new probe is simple yet multifaceted. Seeing that there are multiple touch points in the full system architecture, from the collection of the data on the phone, to the aggregation of the data on the backend, and finally the display of the data on the front end, naturally each of these requires adjustment when adding new probes. Adding a new probe to the openPDS client is rather straightforward due to the use of funf and only takes adjustment of the android permissions in the manifest file and strings.xml, where one can select the frequency and length of data collection using json. An example of the probe settings can be seen in figure 7-1

Adjusting the manifest file is the equivalent of adjusting the funf settings. No code has to be written and ultimately functionality is only inhibited by the factors sighted in section 3.2.1.

7.3.2 Displaying New Data

In order to extend the framework to display new data on the Google map, most adjustments would affect the z value displayed at each particular geographical location. The steps necessary to make these additions are trivial. First a new selector would have to be written. The selector would include the database query and choosing which values to use from the returned objects. Figure 7-2 shows a basic selector. Figure 7-3 shows an example of a selector with a computed z value.

The only other step is to add the particular selector to the UI. In the example application each possible display value corresponded to a particular button. Choice of interface is flexible, but even in the case of corresponding buttons for each computed z value, the additional code is trivial. An example of the initialization of a button can be seen in figure 7-4. Figure 7-5 gives an example of a potential call back function.

7.4 iOS Extension

Although the Android environment was chosen for this project due to the reasons specified in section 2.1.1, bringing some aspects of the functionality to the iOS environment is still desirable. Previous work has already been done on developing an openPDS client for iOS. The main challenge involves the design of an open-sensing framework similar to funf (see section 2.1.1 for more information). Although due to developer restrictions it is impossible to collect the same breadth of passive data sets including SMS and call logs, there are a few advantages to data collection in an iOS environment like the advantages of the low-power motion tracking device described in section 7.2. Also, in many ways any data collected from iOS users would be better than the current alternative.

```
1: {
2:     "@type" : "edu.mit.media.funf.probe.builtin.ActivityProbe",
3:     "@schedule" : {
4:         "strict" : true,
5:         "interval" : 8,
6:         "duration" : 6,
7:         "opportunistic" : true
8:     }
9: }
```

Figure 7-1: Example probe settings

```
1: this.callSelector = function(a) {
2:     val = a.call.count;
3:     location = a.location;
4:     lat = location.latitude;
5:     lng = location.longitude;
6:     return {"lat" : lat,"lng" : lng,"val" : val};
7: }
```

Figure 7-2: Example value selector for new probes

```
1: this.activitySelector = function(a) {
2:     total = a.activity.total;
3:     high = a.activity.high;
4:     low = a.activity.low;
5:     val = Math.ceil((total > 0)?10 * ((high + low)/total) : 0.1);
6:     location = a.location;
7:     lat = location.latitude;
8:     lng = location.longitude;
9:     return {"lat" : lat,"lng" : lng,"val" : val};
10: }
```

Figure 7-3: Example value selector for new probes

```
1: call = $('< div >');
2: call.addClass('button2');
3: call.addClass('call');
4: $('#buttons').append(call);
5: varreal = this;
6: call.click(function(){
7:     real.showCall();
8: }
```

Figure 7-4: Example button interface for new probes

```
1: showCall : function() {
2:     this.selector = this.callSelector;
3:     this.points = this.answerLists.at(0).get("value").map(this.callSelector);
4:     this.loadArrayBuffer(true);
5: }
```

Figure 7-5: Example callback function for new probes

Appendix A

Figures

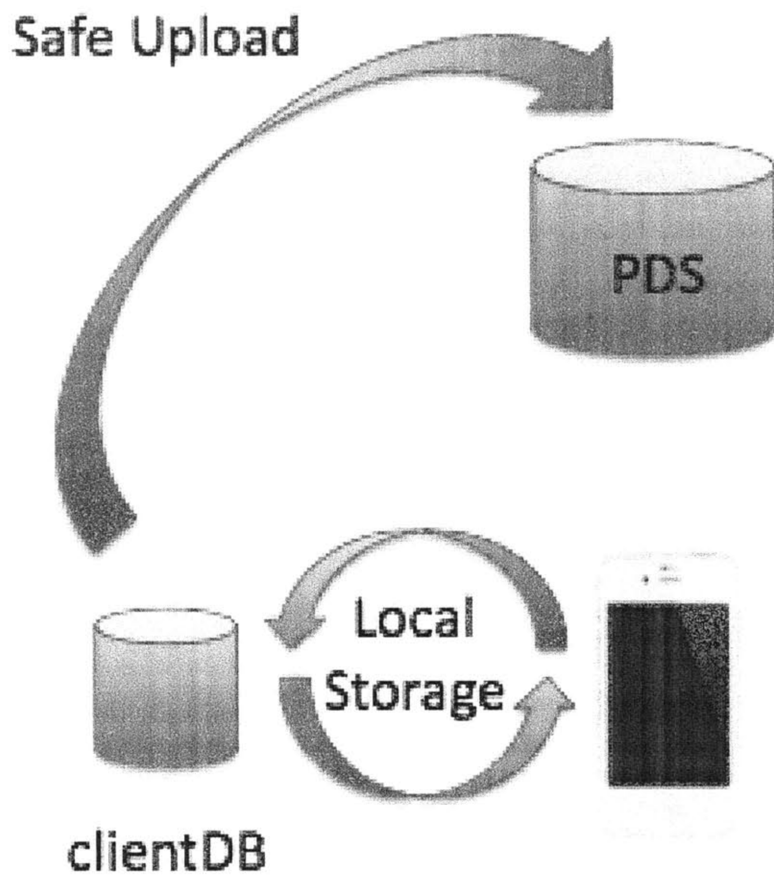


Figure A-1: Original upload routine

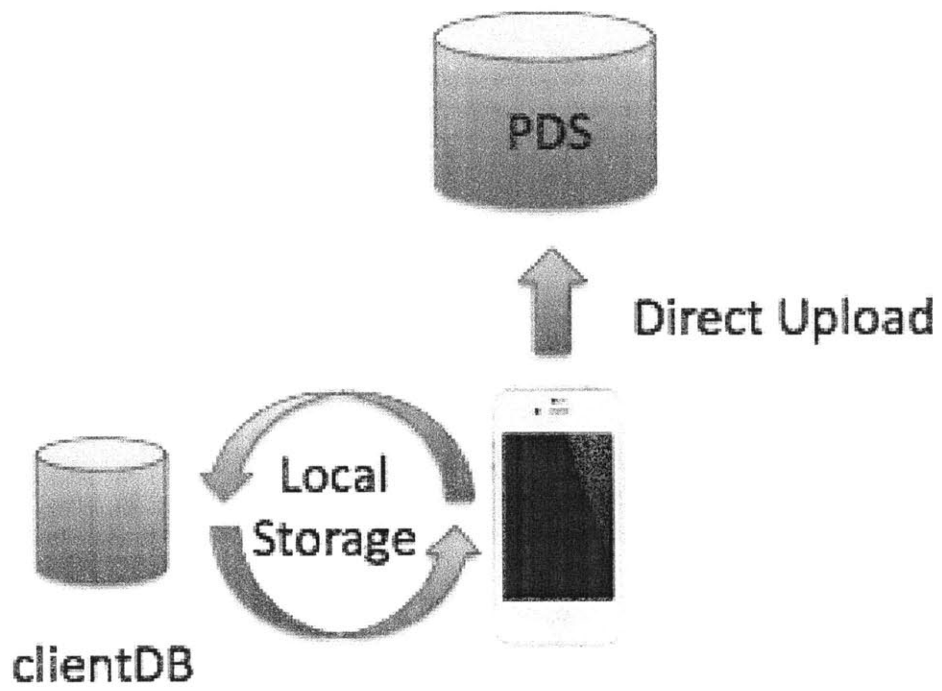


Figure A-2: Upload routine using only direct upload

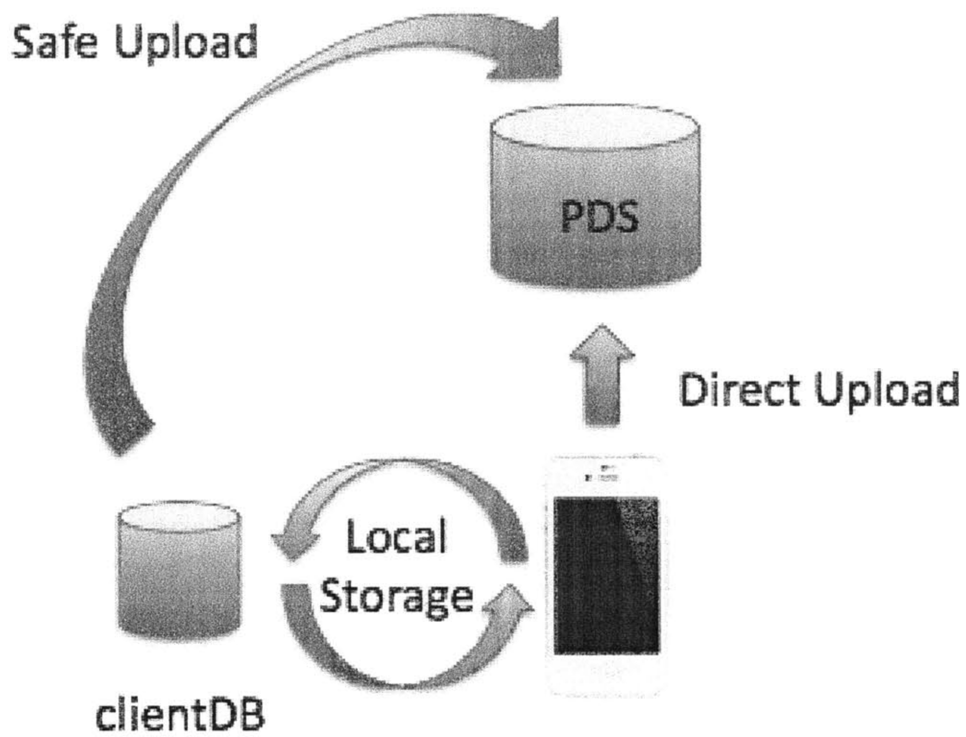


Figure A-3: Upload routine using both safe and direct upload

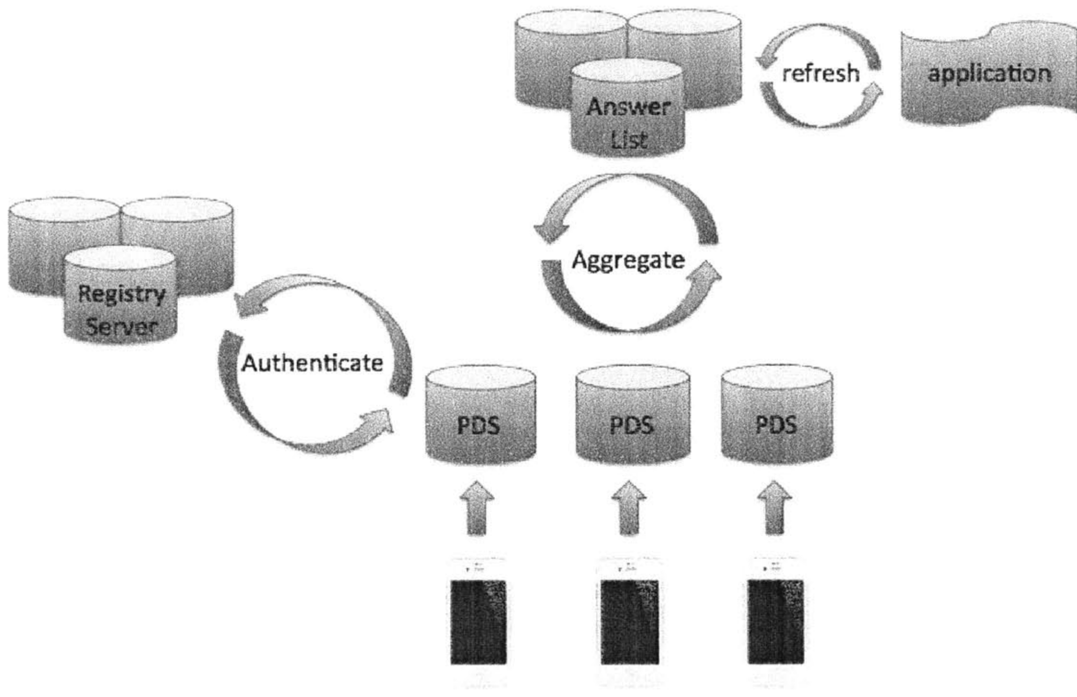


Figure A-4: Complete, end-to-end cycle

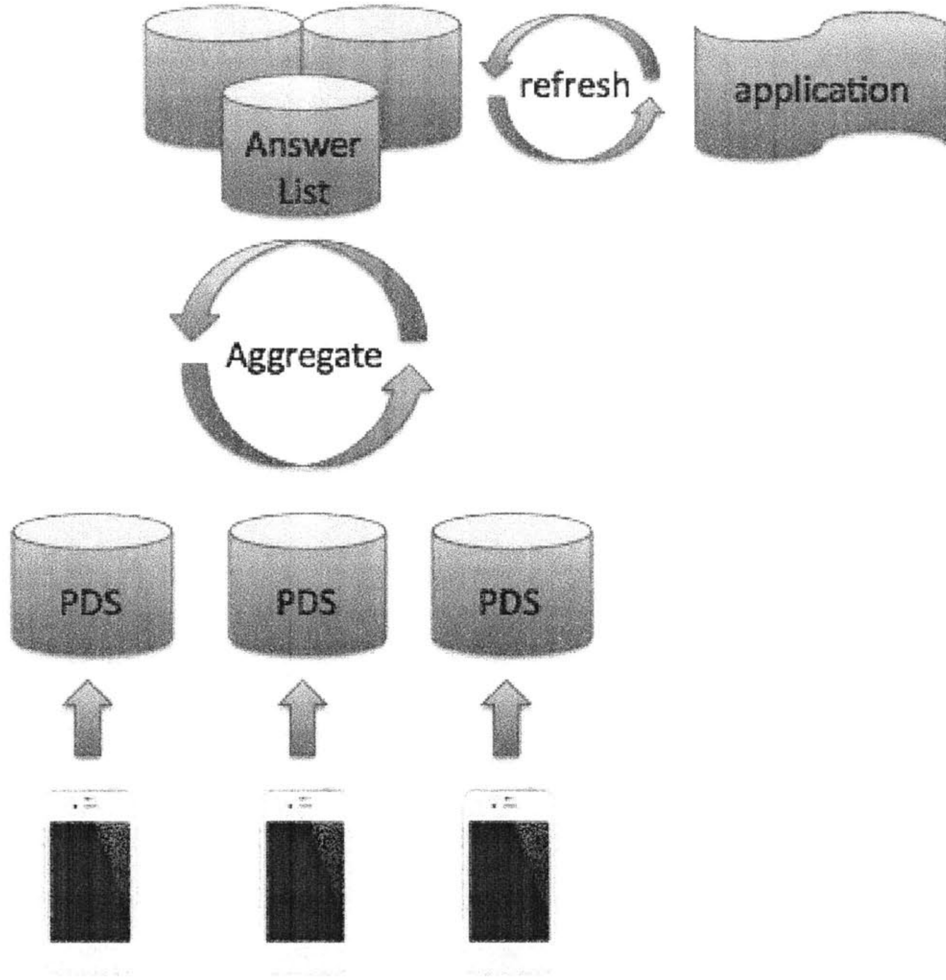


Figure A-5: Full cycle when testing

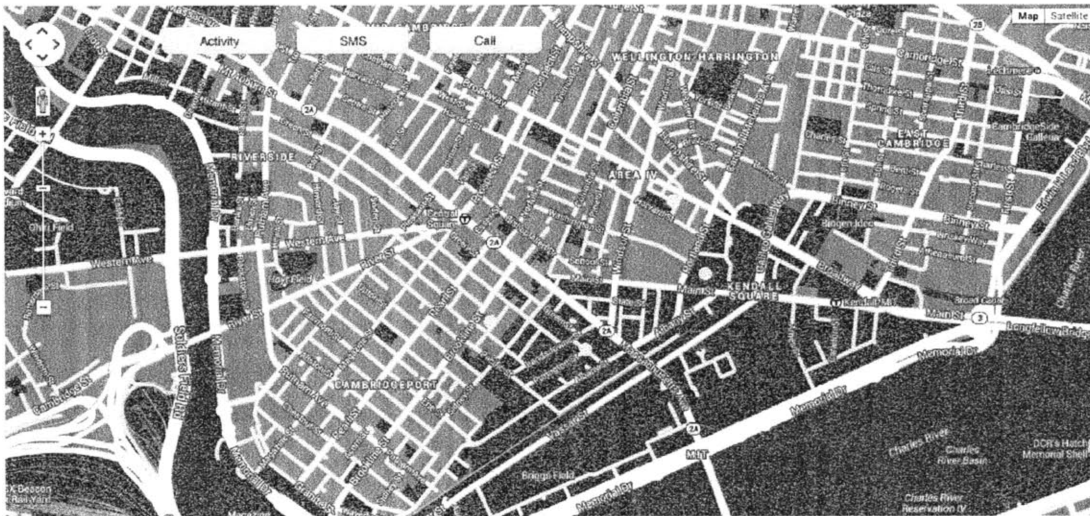


Figure A-6: Screenshot of the realtime application

Figure A-7: Screenshot of the replay application

Bibliography

[1] Codeflow - high performance js heatmaps, September 2014.

[2] Google maps api styled map wizard, June 2014.

[3] Nadav Aharony, Wei Pan, Cory Ip, Inas Khayal, and Alex Pentland. Social fmri: Investigating and shaping social mechanisms in the real world, pervasive and mobile computing. 2011.

[4] funf framework. funf-open-sensing-framework, June 2014.

[5] Human Dynamics Group. openpds - the privacy-preserving personal data store, June 2014.

[6] Khronos Group. Webgl - the industry standard for high performance graphics, June 2014.

[7] MongoDB Inc. Faq: Concurrency - mongodb manual 2.6.1, June 2014.

[8] Ask Solem. Celery - distributed task queue, June 2014.