ECE 575 Team A3

Cellular Coverage

April 19, 2015
Final Report

Ryan Clarke
(rkclarke@ncsu.edu)

Swarupa Ramakrishnan
(sramakr6@ncsu.edu)

Shivakumar Kupparu
(skuppar@ncsu.edu)

Saravanan Varadharajan
(svaradh3@ncsu.edu)

Nandini Rangaswamy
(nrangas@ncsu.edu)

Abstract

The cellular coverage mapper system is a modular set of tools that cooperate to collect and display recordings of network properties in an accessible format. The aim of this project is to facilitate rapid analysis of network coverage in a given area through real-time observation. In comparison to the claimed coverage maps that a cellular carrier may present, this system allows for unbiased analysis of the environmental coverage area along with much more detailed data such as signal strength and data throughput. This is accomplished via an Android background service that regularly polls the system for location and network characteristics and then pushes any noteworthy changes to a central data server. This server then stores the network information in a database and provides access to it by remote query. At any given time the map interface is capable of displaying a single network characteristic from a single network provider as a set of markers on a map. Data can be collected from nearly any network provider, and multiple filter parameters allow the user control over what data is displayed at any given time.

Introduction

Accurate and impartial data collection is an invaluable asset to both network providers and customers. To most efficiently build new base stations in order to fill coverage gaps, a network provider must first know where and when its customers are losing cellular coverage. Equipped with this information, the network provider will be knowledgeable of its areas of relative strength and weakness and will be able to both advertise and expand appropriately.

As a consumer of a network service, an accurate and unbiased collection of network characteristics is of great benefit when determining the best provider for a given area and usage habits. While many cellular providers publish coverage maps displaying their service areas, a customer has no verification of the accuracy of those maps. Regardless of whether the provider is being deceitful or merely optimistic of its own capabilities, assessing the practical service capacity is important. A distributed collection of network data from users provides an impartial data set independent of network provider claims, enabling a more informed decision.

The cellular coverage mapper service fulfills a need for both network provider and customer, which is accurate network characteristic sampling from actual customer use, rather than theoretical analysis. The data collection module is a small Android app that runs in the background and polls for location and network state updates. Any time a noteworthy update occurs, a new data packet is generated and sent to a central database server. This server receives the data from the data collection app and adds it to a MySQL table corresponding to the network provider. The utility of this service is chiefly limited by the size of the user base, as more users will obviously increase the density of the data collection and improve the subsequent data analysis potential. Because of this, the database server is designed with scalability in mind and should remain serviceable even when working with many users and very large databases.

The visualization component of the service consists of two modules: a web-based map display utilizing the Google Maps API and an Android app which provides all of the features of the web display in a mobile-friendly format. The maps are capable of displaying a number of network characteristics, including signal strength and throughput, from a variety of network providers. The display applies color coding to the markers according to network quality, which enhances the display's visual feedback at a glance. Additionally, filters for date and time ranges allow for trimming of the data for both readability and time-based network analysis.

Prior Work

In researching the problem, we discovered that approaches to solve the issue have already been attempted with great success. Two key examples that were discovered were OpenSignal [1] and Sensorly [2]. Both consisted of similar designs, with mobile applications collecting data and sending it to a central server, and the server providing data back to the app for display on a map interface. While our team is obviously at a disadvantage for both development time and resources, emulation of these two solutions seems an admirable goal, though we shall also attempt a new take on the subject with the separation of data collection and visualization into two separate apps in a modular approach.

Problem formulation

This team elected to pursue a modular approach to the project, with each major role being handled by a separate application. This greatly facilitated labor distribution among team members, as each was able to take the lead role in development of a single module, while contributing to the direction of the project as a whole through team coordination. Coordination between modules was established early through a common interface using HTTP POST and GET to transfer data to and from the server.

The data collection module is an Android background service designed to populate the database with localized network characteristics. For this process, the app needs to monitor changes in both network state and location. The app should measure signal strength, download speed, and upload speed. All network providers should be supported along with as many network infrastructures as possible, including CDMA, GSM, LTE, and optionally Wi-Fi.

Updates should be handled using an event-based program flow as shown in Appendix B. Once an update occurs, a data packet should be prepared for transmission to the server in an HTTP POST. This packet should include the network provider, latitude, longitude, timestamp, signal strength, download speed, and upload speed.

The data storage module is a server application written in Java. It should accept HTTP POST from the data collection module and store the incoming data for later use. To facilitate both scalability and parallel access a relational database should be used to store the data. This database will also allow for efficient filtering of the returned data. A separate table should be used for each carrier to improve logical data separation and query efficiency.

The server should also respond to HTTP GET queries by supplying network information from the database corresponding to the specified parameters. The request should allow for specification of any combination of network provider, data type, date interval, and time interval. If date or time intervals are not specified then the default should return all collected data matching the specified data type and network provider. If data type is not specified then the default should be signal strength. The server shall use the Java spring framework [3] for its RESTful web request handling. An Apache HTTP server built into the application shall manage concurrent access of multiple clients by spawning worker threads for each new connection.

In order to present the data in a simple and intuitive way, two display apps are required. The first is a web-based application written in JavaScript which consists of a HTML form for selection of network provider, data type, and filters for date and time ranges. Upon submission of the form a HTTP GET query should be generated according to the server URI using jQuery [4], and the JSON-encoded results should be passed to the map generation code.

A dynamic map is then generated using the Google Maps API, and the displayed data values should be embedded into color-coded markers on that map. The external library "gmaps.js" [5] was utilized to streamline usage of the Google Maps API for JavaScript. As a contingency in case the web query interface became non-operative, a separate database handler was also planned so that the web app could query the database storing the network information directly.

However, the web display was not suitable for mobile devices, so a separate Android app with comparable functionality is needed. After a simple user interface allows the user to specify the desired parameters, this application will use the same request URI as the web application and interpret the JSON-encoded results to generate color-coded markers on a map rendered using the official Google Maps API. This should provide all of the functionality of the web display with a mobile-friendly interface.

Experimental Results

Two interfaces were specified for communication between service modules. The first, to transmit data from the data collector to the server, is an HTTP POST with the URI '<server>:8080/update'. The second interface is used by the display components to request data from the server, and is an HTTP get with the URI '<server>:8080/request?carrier=<name>'. In both URIs, <server> represents the domain name of the database server. Additionally, HTTP request parameters can be specified by appending '&parameter=<value>' to the query for parameters 'dataType', 'minDate', 'maxDate', 'minTime', and 'maxTime'.

The sensor app was built according to the aforementioned specifications, with a few notable exceptions. For throughput, only download speed is recorded at present. Upload speed analysis would require an additional server component that was not implemented due to time constraints. This was deemed an acceptable compromise because download speed is usually far more relevant than upload speed for mobile devices, which in practice are almost exclusively data consumers. Additionally, the app currently only contains code to collect data from GSM and LTE networks. While enabling CDMA support consists of a trivial two lines of code, it was impossible to test due to a lack of readily-available hardware using that network infrastructure.

All other key aspects of the application are fully functional, though for future work it may be beneficial to refine what constitutes a "noteworthy" change in network state or location. At present, the location threshold is 10 meters and the signal strength is divided into 4 ranges, where a transition between two different ranges triggers an update.

During initial development, the data server utilized JSON files for storage to facilitate rapid deployment and small scale testing, but this was soon phased out in exchange for a MySQL database providing much greater scalability. The database uses the Java spring framework for HTTP web processing as specified in the problem formulation above. The two interface URIs are implemented as RESTful web request controllers using this framework. All filtering capabilities specified above were tested and verified, along with extensive debugging of the automatic database transactions.

Initially, problems occurred when querying the server due to access control errors. Therefore, a direct database handler was planned to bypass the issue. However, after the access control headers were fixed this database interface became redundant with the data server, so the web app was refactored to use the unified URI for data queries. Additionally, all markers are now rendered translucent after it was discovered that a high marker density completely obscured the map underneath. All filters and data queries behave as expected and the markers are displayed and labeled according to their respective properties, including color-coding based on the network quality. A preview of the web display is shown in Appendix C.

The Android app was likewise able to fulfill the requirements and experimental results were found to be comparable with actual expected results. The app was able to navigate seamlessly from one page to another without any interruptions and all the user preferences were captured as required. The Google Maps Android API was successfully integrated with the app and it was able to show the markers with appropriate color coding just like the web display. Additionally, the map is automatically centered on the user's location, a feature which the web display does not implement. For a preview of the app's interface, see Appendix D.

Analysis

Most issues encountered with development of the data collection module stemmed from the team's general lack of prior Android development experience, so much research and learning of the system processes were required. While Java, the language of Android, was familiar to all of us, the specific conventions and activity-based code structure were not. One initial issue resulted in the signal strength update callback being invoked many times every second, often differing by only one or two out of a range from 0 to 31. This was resolved by dividing the signal strength range into four intervals and limiting the updates to only occur when the signal strength crossed from one interval to another, ignoring insignificant minor fluctuations. This reduced the signal update frequency to about once per minute.

Measuring download speed proved to be another interesting development problem. Video streaming tests proved unreliable due to server-side throttling of the data rate, so instead fetching of high-resolution images of a known size and measuring the download time was used to measure the average download speed of the network connection.

Database server development progressed quickly and smoothly with few exceptions, which was expected as most of our team was most proficient in Java, which the server is built on. The main errors concerned database query formatting and remote access control. Debugging the database queries was a very rapid process of trial and error, first tweaking a MySQL query by hand until it produced the desired outcome and then replicating that syntax exactly in the automatically-generated server queries.

Of more interest was the access control failure, which initially prompted development of an independent database handler for the web display module, required some research in order to resolve. While web browsers and the mobile app were able to access the data as expected, the web server was not. Instead, an access control error was returned. Upon researching the issue, we discovered that Cross Origin Resource Sharing (CORS) headers were needed in the server reply to the HTTP GET. These were then implemented in the spring framework filter configuration, and the web display was able to properly query the server using the defined URI.

The primary setback in development of the web display was not actually a problem with the web application itself, but rather the previously-mentioned server access control issue. Thankfully however the partially implemented database handler was rendered unnecessary by the relaxed access control from enabling the CORS headers. Because of this, it could be replaced with the original URI for data queries that both visualization modules can share. The web display application utilizes the jQuery library for web request and response handling.

Additionally, this means that the web display can now run from anywhere on the Internet, and does not require access to the local database on the server. In fact, its portability allows it to run in a web browser successfully from a local directory on the user's computer, without the need for a web server at all.

Another minor issue was the discovery that too many markers displayed at once made the map unreadable, appearing only as blobs of color. To remedy this, 50 percent transparency was added to the markers so that the map would still be visible underneath. While gmaps.js proved invaluable in streamlining access to the Google Maps API, most of the desired marker properties – including transparency – were beyond its capabilities, so referencing the underlying API options was needed for the marker creation functionality.

The core goal of the Android visualization app was quite simple – replicate the web display within a mobile interface. Rather than a single page like the web display, parameter selection and map display are separate pages within the app. This was done to improve ease of use on a small screen. Apart from this the feature set is identical to the web display, with the added improvement of centering the map on the user's location. This was made easier to Android's good built-in location polling capabilities.

Conclusion

This project was a great learning experience for a team previously unfamiliar with Android development. Developing a modular system for collection and display of cellular coverage data provided general insight into both the operation of the location and network state update callbacks in Android as well as GIS visualization technology like Google Maps. The modular approach was a very good design decision that greatly streamlined parallel development by multiple individuals. With a common, well-defined interface to communicate between modules, each could be tested and operated independently without regard for the implementation of the others. This abstraction of roles proved very helpful for both development and debugging of all aspects of the project. When all completed modules were combined in a test of the entire system's functionality, everything performed according to the team's expectations.

Appendix A: References

[1] OpenSignal. < opensignal.com >

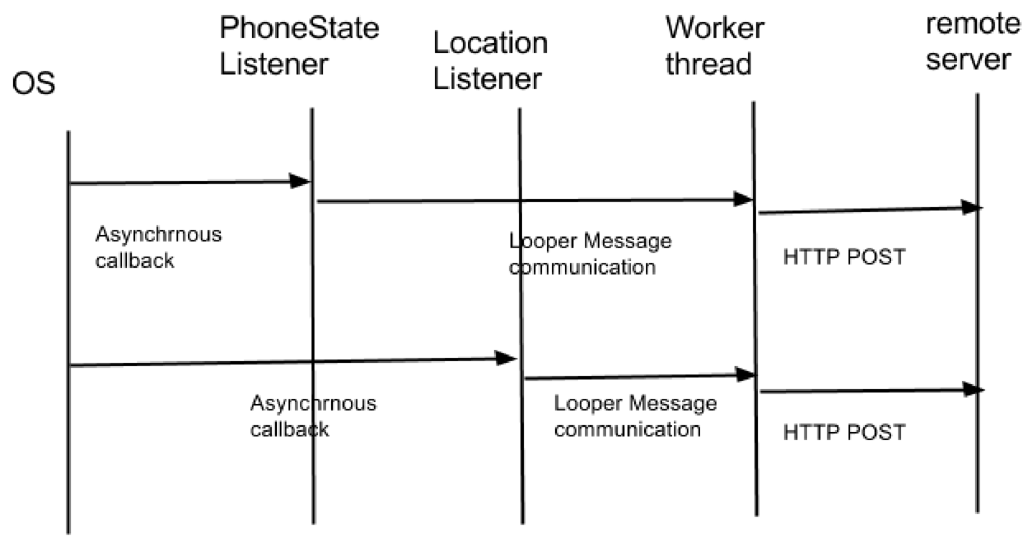
[2] Sensorly. < sensorly.com >

[3] Java Spring Framework. < projects.spring.io/spring-framework >

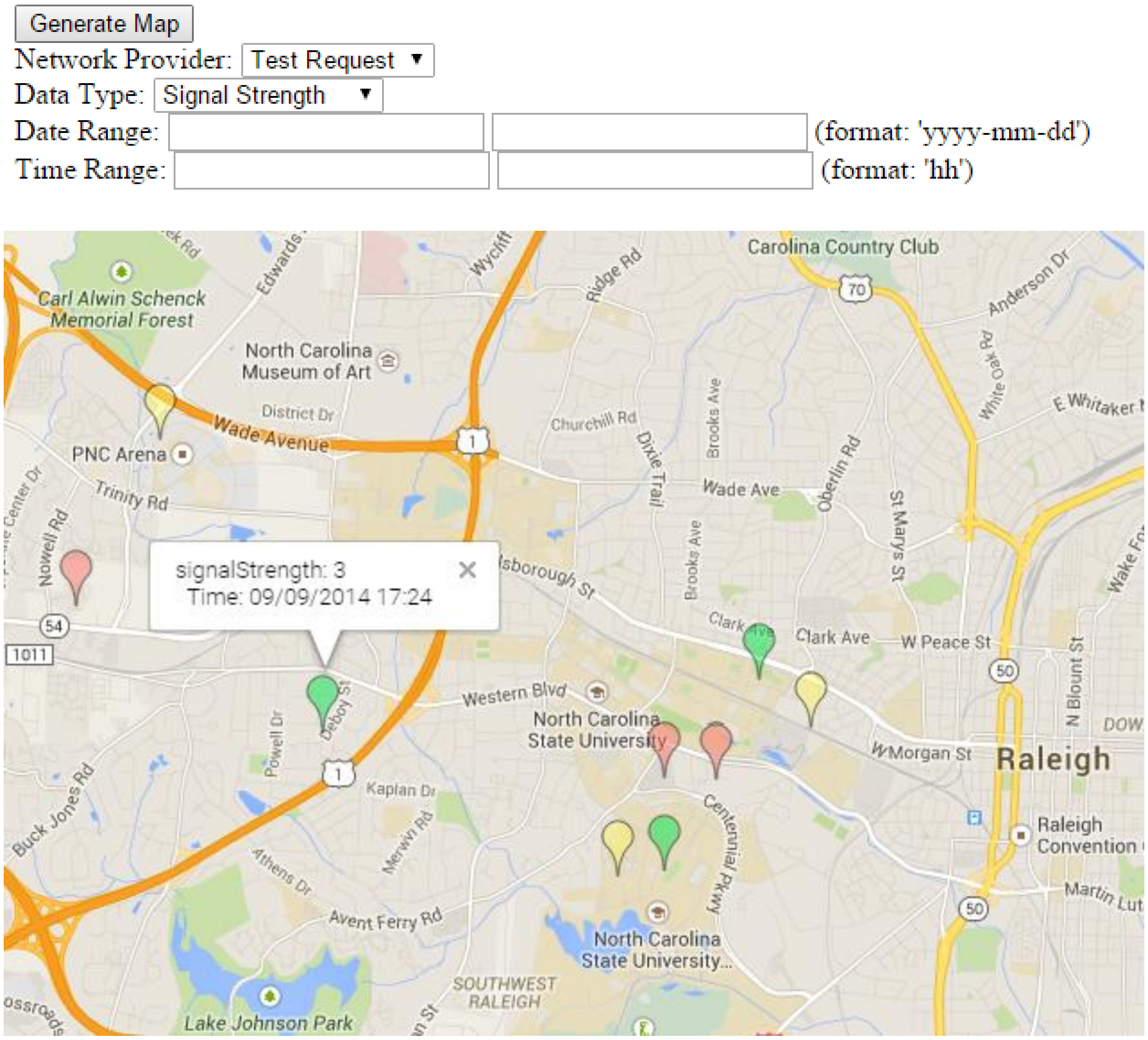
[4] jQuery. < jquery.com >

[5] gmaps.js. < hpneo.github.io/gmaps >

Appendix B: Data Collection App Event Flow



Appendix C: Web Display Interface Example



Generate Map

Network Provider: Test Request ▾

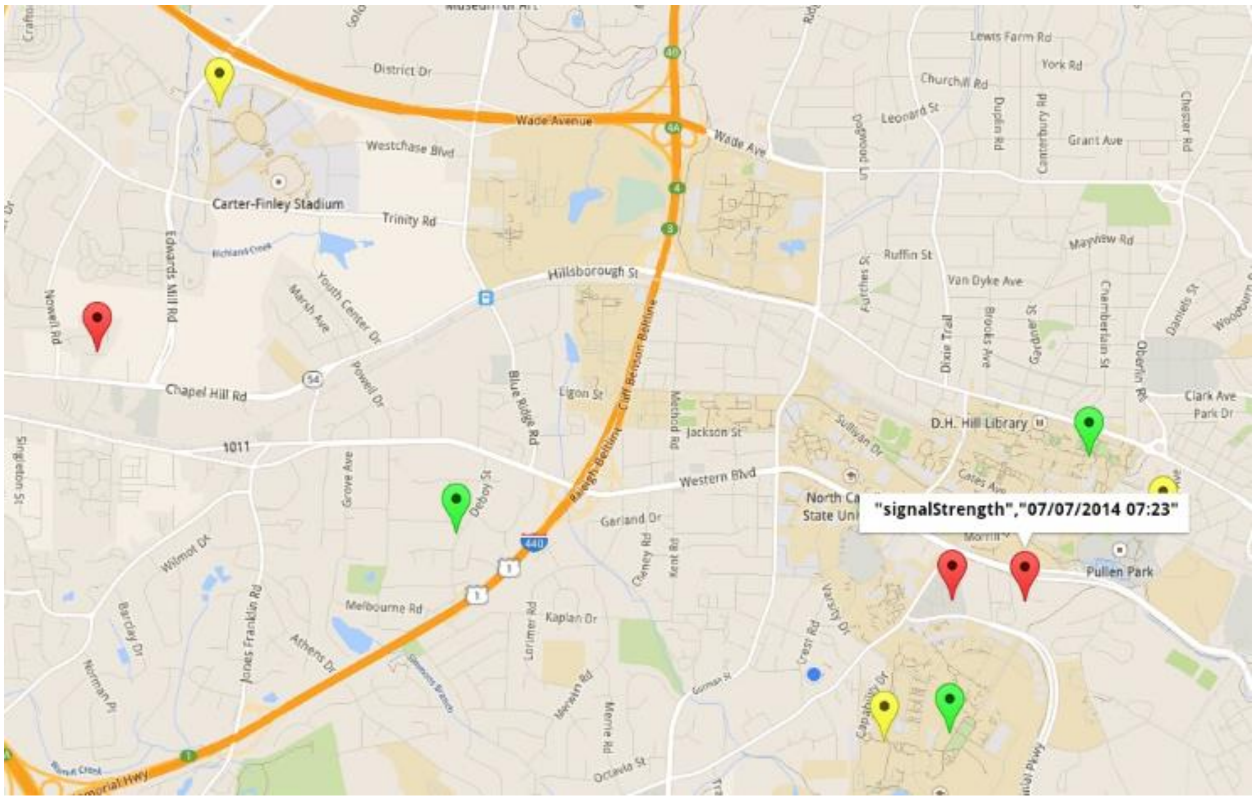
Data Type: Signal Strength ▾

Date Range: (format: 'yyyy-mm-dd')

Time Range: (format: 'hh')

signalStrength: 3
Time: 09/09/2014 17:24

Appendix D: Android Display Interface Example


"signalStrength","07/07/2014 07:23"