Final Report:
Parking Information App for Commuter Rail

Apper
Adam Wenneman
adam.wenneman@mail.utoronto.ca

Programmer
Tai Nguyen
tai.nguyen@outlook.com

Programmer
Mike Qin
mikeandmore@gmail.com

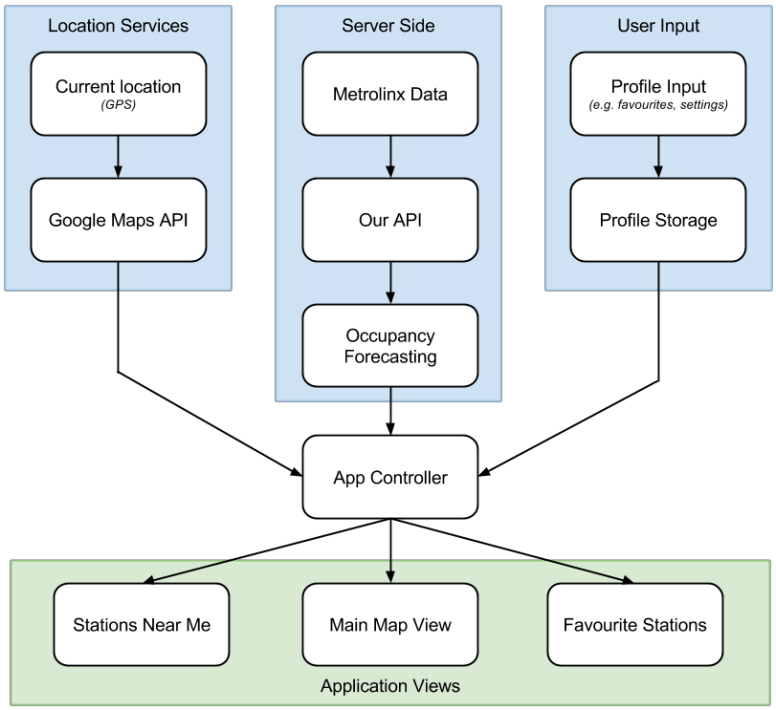
Word Count
Main Report: 1,829
Apper Context: 285

Introduction

GO is one of the largest commuter rail services in North America. In 2012, they served over 80,000 morning commuters, and they project this number will grow to over 120,000 morning commuters by 2020. While GO trains have adequate capacity for all of these commuters, GO parking facilities do not. Already over 50% of station parking facilities are at or near capacity, and GO has observed both the time spent searching for parking and the incidence of illegal parking rise as lots approach capacity. The inability to quickly find free, convenient parking is a major frustration for riders that have become accustomed to it being readily available at GO stations. At the same time, many GO lots are underutilized. This is not a problem GO will be able to simply build themselves out of, due to the high costs of building and maintaining large parking structures and rising land value surrounding the busiest stations. GO is concerned about this looming parking problem for two reasons: over 60% of riders currently access GO stations by personal automobile, and historic morning ridership has closely matched the available parking supply. To solve this parking problem for it's users, GO is trying to find ways to make better use of the existing excess parking capacity at other stations.

Our app addresses the parking problem at GO stations by attempting to shifting commuters from stations with little remaining parking capacity to stations with excess capacity. This results in a more efficient use of GO's excess parking facilities and less frustration for users previously unable to find convenient parking. Our app does this by providing information on the availability of parking at each GO station for the time at which the user will arrive at the station. This is important because in peak commuting periods, most commuters arrive within a small time window. Many heavily used parking facilities approach their capacities in a short amount of time during the morning rush. If our app simply reported current occupancy, it is likely .... By forecasting parking occupancy, we can tell the user whether or not they will find a spot when they arrive before they even get in their car. With this information, users can then decide to drive to a different station, or even to bike, walk, or take local transit to GO instead.

Overall Design



Server Side

The server functions both as the repository for all parking information, as well as the prediction engine. It contains information on each parking lot including maximum capacity, GPS coordinates, and lot occupancy. This information is currently populated from historical data, but can also be updated in real time. MySQL was chosen as the backing data store because of the data's highly relational nature. The database was designed as an abstraction layer between the lot operator's data and our API so that we can easily consume and make use of data from different operators. The web server is built on the Play! Framework. The decision to use a web framework allowed for rapid prototyping of our ideas. The entire server side implementation is built on Amazon Web Services, again, because it was quick to deploy and free to use.

The Metrolinx Data block is complete, but not to the extent we had initially planned. We were hoping to get a connection to real-time occupancy information on Go Station parking lots. We only received historical data in Excel files. These files contained surveys of station managers on when the lots reached 90% capacity, and month to month variations, and other statistics.

The API block is complete. It is the intermediary between the database and the app. It serves requests from the app with JSON formatted data.

The forecasting block is currently very rudimentary. We report that a lot is at 90% capacity based on the time indicated in the surveys. Analysis of the data manually collected from Malton station shows that the pattern of occupancy for that particular station resembles a step function. In the future, when we have fine-grain, real-time information at our disposal, a station may be modeled by a collection of logistic functions. This work remains to be done.

App Side

While the server side merely provide an interface to access our prediction server, the App side will consistent all of the functionality that users need in a user-friendly way. In our App, we are able to show all the stations on the map, list user's favorite station and show station details. In the Map view, we show a station as a marker, color of the marker will indicate how full the station parking lot is, and how long does it take for user to get there from current location; station detail view will show a predicted occupancy overtime, in case of user getting there late or early.

Our app contains the following modules: local database storage, communication, and UI. Local database storage is used for storing frequently accessed data, such as station list or favorite stations, so that user won't have to fetch that every time through the internet. Communication module was used for communicate our prediction server. As mentioned previously, our prediction API will return a JSON response, communication module is responsible to construct appropriate request and parse the JSON response. UI module will construct the user interface from the data structure it read from either communication model or local storage.

We choose to use Scala programming language and Scaloid library to implement our app. Scaloid is the Scala wrapper of Android SDK. Scala was chosen because it could dramatically reduces the size of readability of our app code. Also a lot of standard library in Scala turns out to be very useful, such as Future and JSON parsers.
While we implemented our app, we found that HTTP library provided by Android SDK have a fatal flaw: it does not support HTTP 1.1 pipeline[1]. While we need to request more than 20 responses simultaneously, we decided to implement our own HTTP client.[2] Fortunately, it only takes less than 900 lines to implement a HTTP client in Scala.

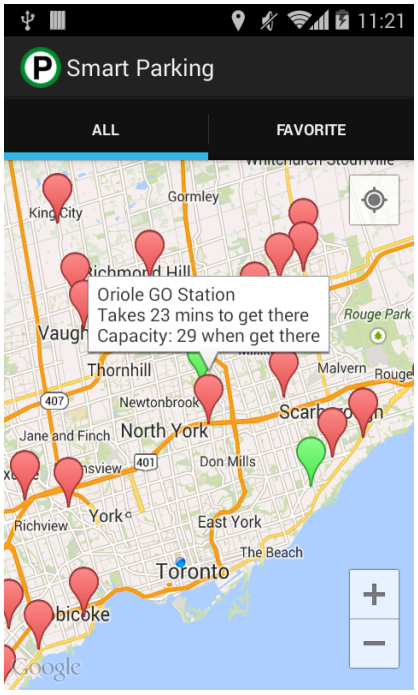
In the future, we decided to work on the portability of our app. Right now there are two issues we have that limit our portability. First, Scala JSON parser is a recursive descent parser and will consumes a lot of stack space. This is not allowed on many phones. Second, when we started this app, for consistency behavior and easy to debugging, we chose the stock Android SDK rather than the support library. This will

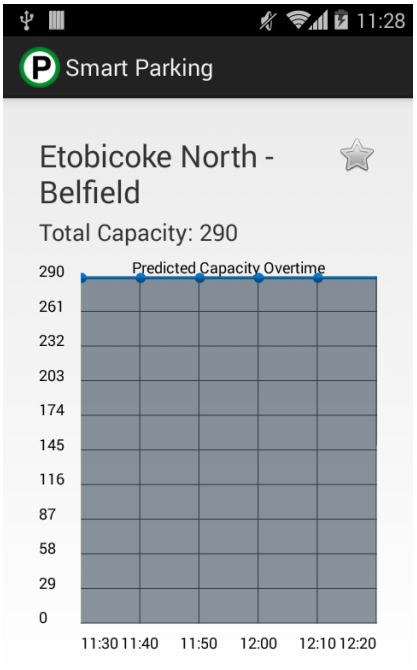
[1] https://code.google.com/p/android/issues/detail?id=3273
[2] Note that other Android App is suffering from this issue as well. The Opera Mobile and the Android Browser both have its own HTTP client that supports HTTP 1.1 pipeline.

greatly limit our portability. Since this app is still in a prototype stage, so we decided to push our portability issues to the future work.

Statement of Functionality

Recall that our goal was to present the user with a tool to quickly and easily determine the availability of parking at the time of his or her arrival at a lot. To this end, our efforts are largely a success. We have a functional app backed by all the required services. Where we fell short was the ability to predict future lot occupancy. This was due to lack of data. GO does not charge its riders to park at its stations. As such, it has little incentive to collect real-time occupancy information. Without this information, we were unable to build a model or even to explore what such a model might look like. We continue to develop our relationship with Metrolinx and other transit operators. Should we eventually gain access to this data, we have designed the system such that it will be easy to integrate in the future.





Lessons Learned

From the App side, we have learned that getting a app working efficiently is a great challenge. Android provide a Java/JVM environment, but programming on it feels different than other traditional Java Platforms. First, we have to be aware of the lifecycle. In the traditional Java programming platform, lifecycle is simple and Java programmer does not

need to think too much about it. However, on Android, lifecycle is complex and important, we have to think explicitly how the lifecycle of resources works.

Second, it's the resource constraints. On traditional Java platform, memory is usually cheap, but on Android, it depends. Some latest phones have 3GB RAM, while some other still decent phones, like Nexus S, only have 380MB RAM. This limitation causes various resource constraints and performance problems, like stack size or frequent GC pauses. Therefore, in order to make our App portable on as many phones as possible, we want to make sure we use and allocate memory efficiently.

Contribution by Group Members

Tai was responsible for the server side implementation. He built the web server and database, and imported the Metrolinx data. Additionally, he and Adam performed experiments to determine the feasibility of using the TI sensor tags to collect real-time occupancy information. When the sensor tags proved insufficient, he and Adam went on site to manually collect occupancy information.

Mike was responsible for the Android App, implementing all the features we need on the app side: including mapview, bookmarks, station details, etc. Our Android App will request information from Google Map server and our server for prediction. Mike has written the protocol for the communication between our Android App and web server.

Adam was responsible for the initial idea for the app. He worked with contacts at Metrolinx to gain access to lot information data held by Metrolinx as well as their support for the project. He collaborated with Tai to perform experiments to determine the feasibility of using TI sensor tags as traffic counters, which would have allowed the team to collect higher quality data. He planned and coordinated a station occupancy count that the team used in the forecasting step. He also managed the team's workflow, ensuring all deliverables were submitted on time and contained all required content.

Apper Context

This application is an implementation of a parking information system. Much research has surrounded this type of system, but this application is the first working system we are aware of in Toronto. The benefits of this type of system accrue to several groups. Users benefit by saving time previously spent searching for parking. Lot operators benefit by making more efficient use of their existing infrastructure. Society itself benefits as well, as reduced cruising for parking and more efficient use of facilities reduces congestion and prices over time.

Although this application applies the idea of a parking information system in the context of a commuter rail service, it can be used in many other areas as well. My research is focused on commercial vehicle operations in urban areas. Frequently, commercial vehicles are unable to find adequate parking facilities to make their deliveries. This

problem applies both to couriers that need a space for just a short amount of time and larger commercial vehicles that may require the use of loading facilities. Given the constraint that significant amounts of additional parking facilities will not be constructed in the near term, the problem for commercial vehicle operators begins to closely resemble the problem faced by commuters using the GO system. Instead of planning a destination GO station, commercial vehicle operators would use the app to plan their delivery tours, according to forecasts of where and when they would be able to access parking and/or loading facilities. This application provides a framework for how such a system would be constructed and how it would work. The exercise of building this application also provided valuable information on what some of the challenges of creating such a system would be.

Future Work

There is a large amount of potential for improvement for this application. The current version suffers from long loading times required to collect the travel time information from Google. Additional details may be added to the station detail screen, such as what other facilities (bike racks, EV charging spots, car share spots) are available at the stations. Train schedules could be added, so that users would know if the train will arrive at the station early, on-time, or late. Station maps may be useful for users that are travelling to a station they do not frequently use. If GO implements paid parking at some stations, this app could be expanded to handle these payments. This app would be particularly well suited to this improvement if paid parking is applied asymmetrically in an effort to shift riders from congested stations to the under used stations.