EECS589 Final Project Report:

"Blue Bus Tracker" App

JIASHUO WANG 98293600
JUN YIN 66408000
PENG SUN 84224689
INSTRUCTOR: Z. MORLEY MAO

Abstraction

We build a "Blue Bus Tracker" app, together with the server, to help users track the blue bus based on the user's schedule and finally notify users so that they will not miss the bus. We also test the performance of this app with the battery saving, response time and accuracy, which proves our app works well under network scenario.

Key Words:

Android Wear, Server-Clients, Energy Saving, Google Cloud

1 Introduction

As a student of University of Michigan, you may forget the exact arrival time of Blue bus and have to rush towards the bus station. Someone may use the smart phone app to track the real-time blue bus information, but he or she needs to focus on the smart phone screen and could not concentrate on anything else, which is quite inconvenient and annoying for Umich students.

In this paper, we would like to introduce an app----Blue Bus Tracker, which we developed under Android Wear on smart watch. Our main motivation is to allow the wearer to have hands-free access to information in real time. With the assistance of our app with smart watch, Umich students are guaranteed to catch the bus timely in most time.

Specifically, the user opens the app on the watch, puts in the desired arrival time and destination, and choose a routes from those provided by the server, and then let the watch go into sleep mode. On the other side, a server which works with watch app, is responsible for retrieving, analyzing data, and push notification to the app.

2 Android Wear Implementation

Android Wear is a version of Google's Android operating system designed for wearable devices such as smart watches. In smart watch like LG urbane, it supports both Bluetooth and Wi-Fi connectivity, with which the smart watch could pair with a phone. By this means, Android Wear on smart watches could use some feature or hardware from the phone, even if smart watches do not have. For example, most of smart watches do not have GPS in them, but Android Wear could get GPS location information based on the GPS in the paired phone via the connection between the smart watch and the phone.

Software development on Android Wear is much like the development we are doing on phones with general Android. We create several activities to handle the interaction with user based on the status of user decision and the information collected by the device. The relationship among those activities and related actions are shown in Figure 2-1.

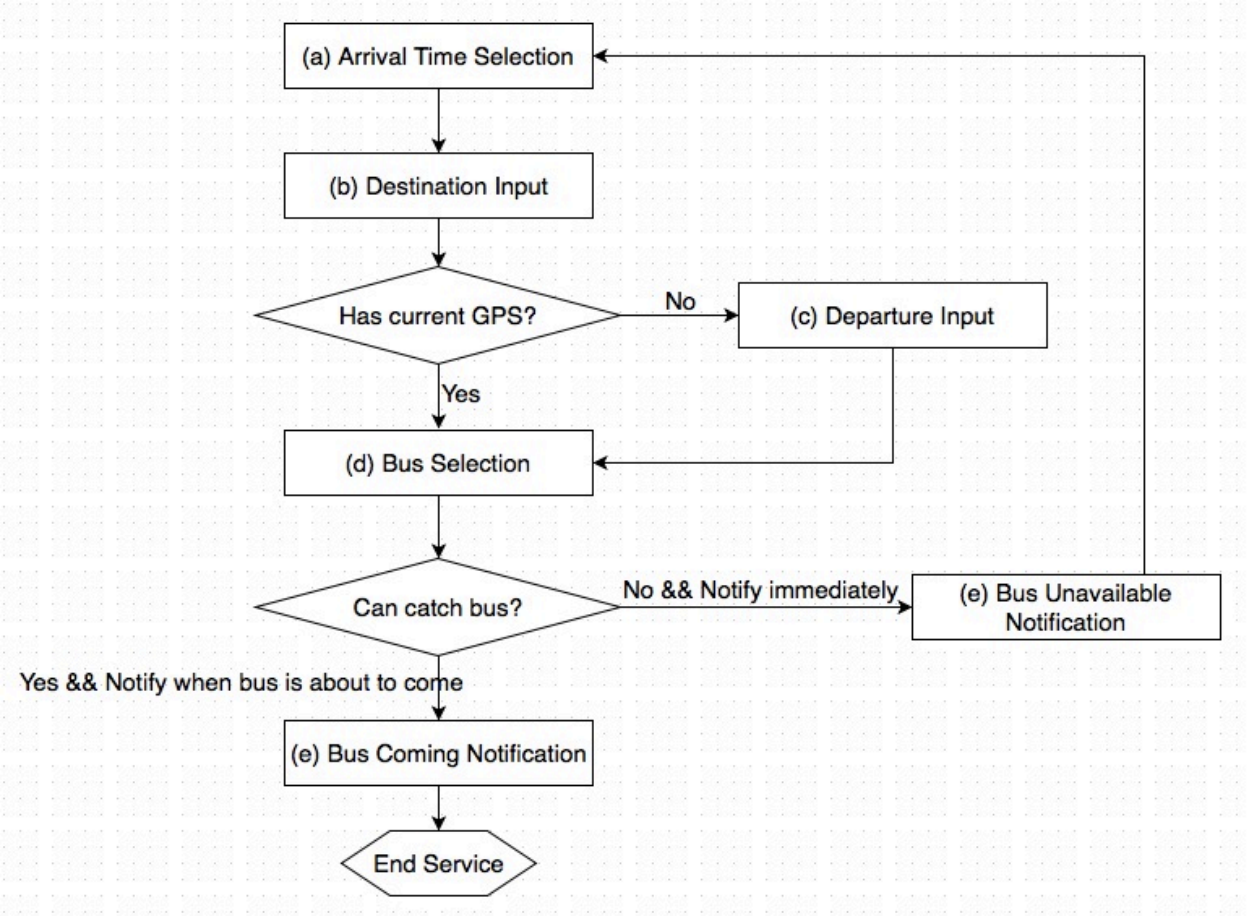
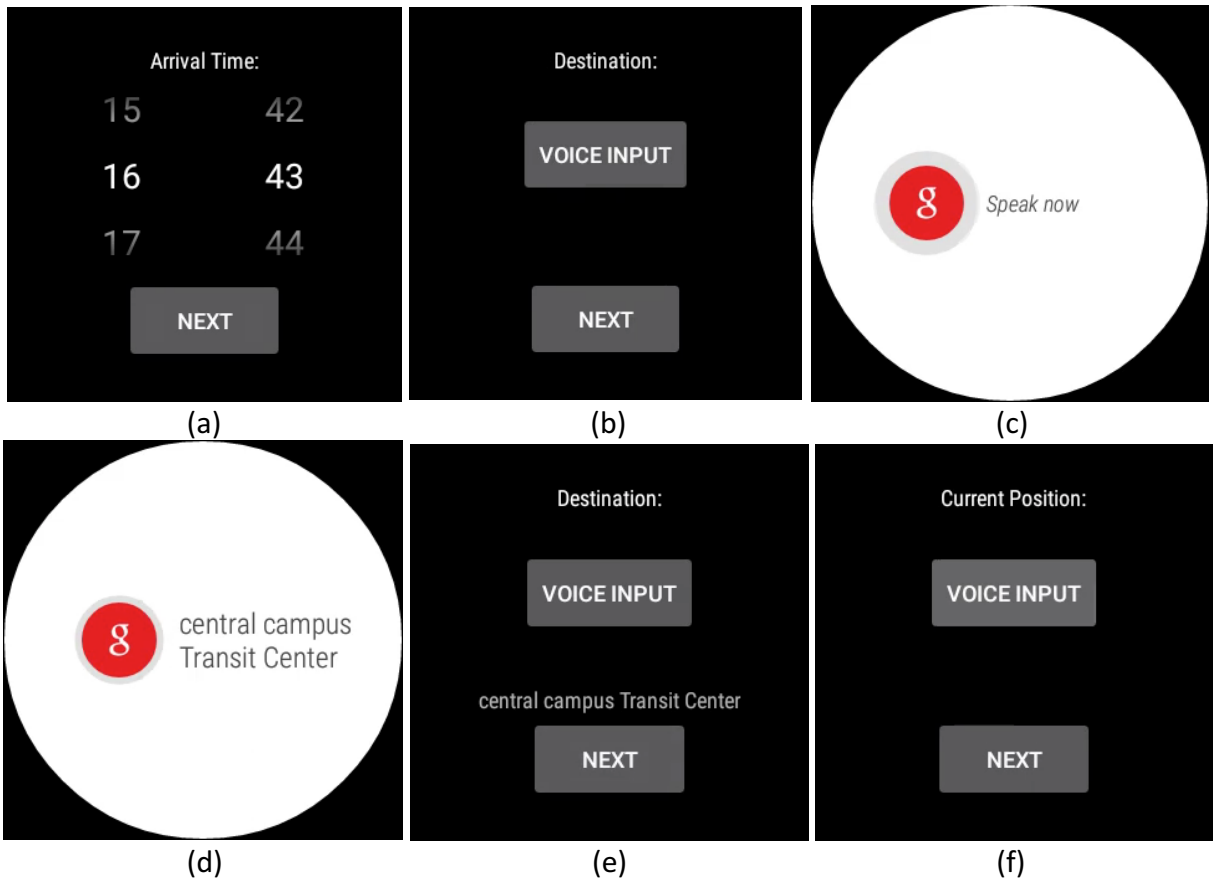


Figure 2-1 Android Wear Activities Workflow

2.1 Arrival Time Selection

Suppose we have the following situation: Two hours later, a user wants to go to the central campus from north campus where he or she is having a very heated discussion. This user doesn't want to be late for his or her next schedule, but he or she is afraid to miss the time because of this heated discussion. Here comes our app!

When he or she opens our "Blue Bus Tracker" app, the first Android Wear activity is to select the arrival time (Figure 2-2a), which provides the UI for the user to choose what time he or she needs to arrive at the position in Umich campus, even if that arrival time is kind of far away from current time.



(a) (b) (c)

(d) (e) (f)

Figure 2-2 All UIs of Blue Bus Tracker App (Part 1)

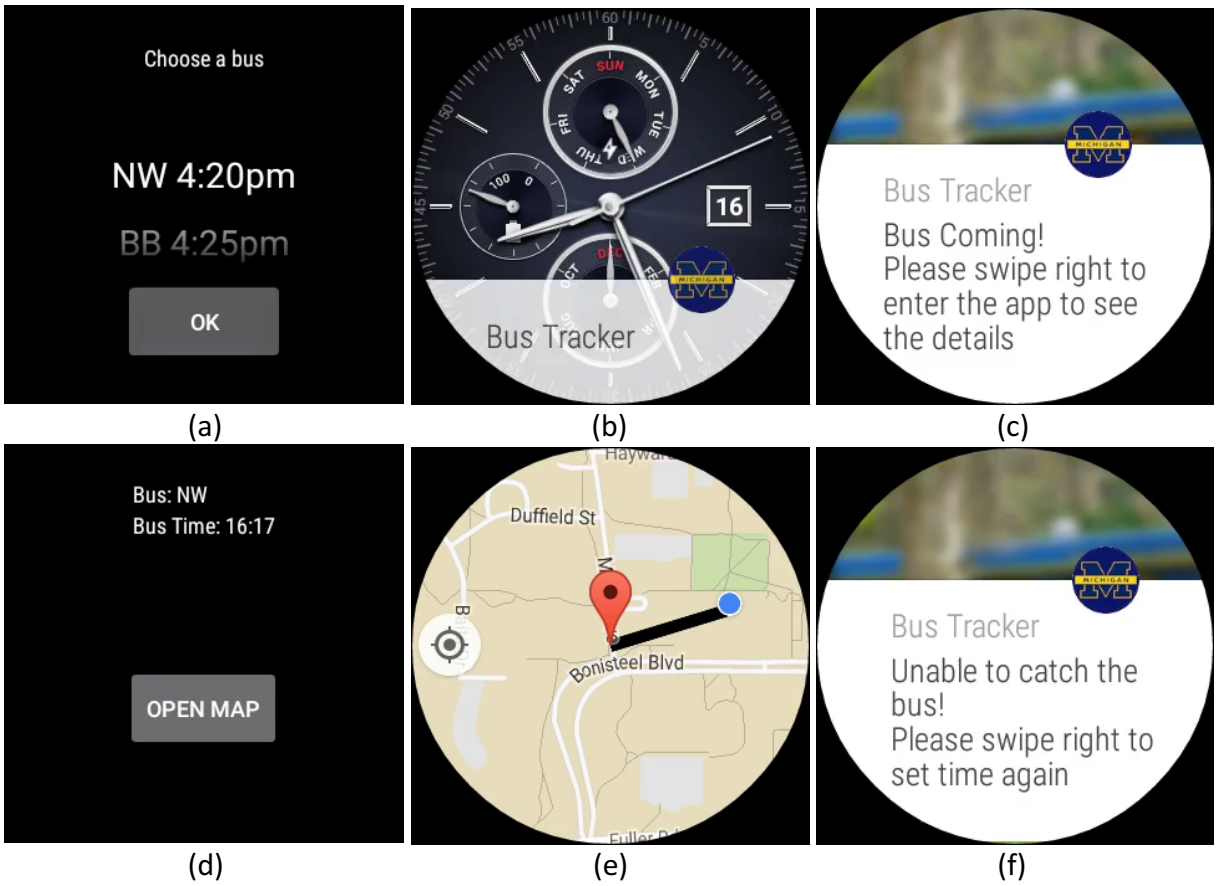
This will involve a NumberPicker widget in the activity. After clicking the "Next" button, we will jump to the next Android Wear activity.

2.2 Destination and Departure Input

In this activity, we want the user to use voice to input the destination (Figure 2-2b-e). Meanwhile, in the background, the device will try to collect the GPS information for current location. If the GPS information is unable to be obtained, the user will get the chance to voice input the departure location via "Next" button (Figure 2-2f).

To implement this functionality, we use both GPS and voice services provided by Google Android. For GPS, we use Google API Client and Location listener to get the current GPS information, which contains both latitude and longitude updated every time when GPS value is changed. For voice, we use Google voice application provided by the default Android operating system. This needs us to use intent provided by Android to start activity for the Google voice.

After clicking "Next" Button in destination input activity (When we could get current GPS) or in departure input activity (When GPS location could not be obtained), smart watch will then send those user inputs and current GPS location to server. Server will send back some bus options for the user to choose in the next activity.



(a) (b) (c)

(d) (e) (f)

Figure 2-3 All UIs of Blue Bus Tracker App (Part 2)

2.3 Bus Selection

After receiving the bus options from the server, the app will then jump to the bus selection activity (Figure 2-3a), where the user could choose which bus is the one he or she prefer to take, together with the estimated departure time for him or her to leave that heating discussion.

When the user clicks “OK” button, the app will send this choice back to server so that the server could track that kind of bus for user, and then will simply quit so that the smart watch could sleep to save energy. Meanwhile, the user could just continue the discussion and wait to be notified by vibration (and sound) of the watch.

2.4 Bus Coming Notification

When the server could arrange a proper bus for the user, and when that proper bus is coming based on the algorithm we are using (We will discuss that in Section 3), the server will push a notification to the watch (Figure 2-3bc).

We could get into the app through the notification, where the bus name and the time for the bus to arrival at the departure stop will show up (Figure 2-3d). If clicking the “Open Map” button, we will be able to see where the stop is on the Google Map, which could help guide us to the departure stop (Figure 2-3e). When the user getting on the bus, this service will be done.

The thing behind this scenario is to implement notification and opening Google Map. For notification, we could simply start the activity from the listener Service, which is used to get the information from server (We will discuss this in Section 4). For opening Google Map, this needs more work about Android Wear map activity and also register the Google Map API key from Google service.

2.5 Bus Unavailable Notification

As mentioned in Section 2.4, if the server can't arrange a proper bus for the user, the server will push a notification immediately so that the server could adjust the inputs, or change to another transportation method such as Uber to be able to get to the destination on time (Figure 2-3f).

3 Server Implementation

There are two phases of the server end. We will introduce these two phase in the following Chapter.

3.1 First phase (Wait for request):

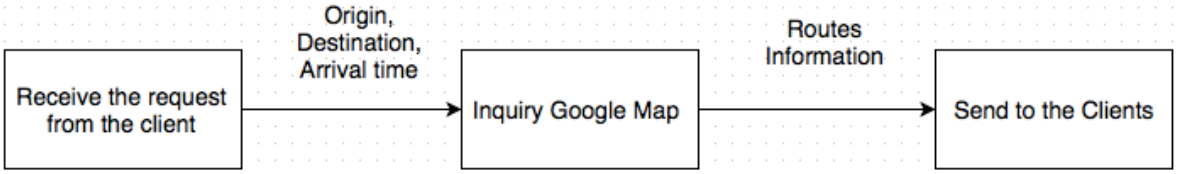


Figure 3-1 First Phase Workflow

In this phase, the server waits for the request from client. Once received the tracking request, server uses the origin, destination, arrival time information provided by the client to inquiry Google Map API. Then it sends back the routes' choices to client and wait for the client to choose. On the backend, the server needs to store the geographical coordinates of departure bus stops, route during time, walking time, and bus name for second phase analysis.

3.2 Second phase (Wait for bus coming):

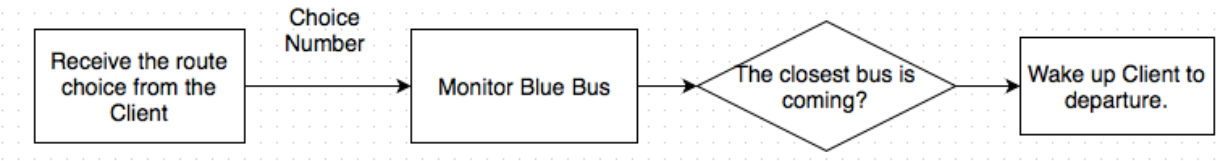


Figure 3-2 Second Phase Workflow

Once the server received the choice from the client, it begins to monitor the chose route. Firstly, it uses the geographical coordinates of departure bus stop to find the stop id used in the blue bus database. Then, it keeps using the stop id and route id to inquire blue bus real-time information until the closest bus is coming.

How do we define the closest bus? It means that when the arrival destination time of next bus exceeds the input, we call the previous bus is the closet bus.

When we find the closest bus and the arrival departure stop time of this bus is larger than the walking time for a certain value, we should notify the client in order to catch this bus in time with smallest waiting time. At this time, we send the departure stop's geographical coordinates, bus name and bus arrival time to client and wake up the client.

4 Communication

There are several ways to be able to communicate between the server and the smart watch. We could use socket, which could easily use TCP connection based on IP and port of the server. We first implemented this kind of way to communicate. But then we found the problem: Sometimes they could not communicate. We realized that in this project, we use our laptop to be a server, which connects the internet through the Umich Wi-Fi. In this case, sometimes the server and the watch will not be able to stay in the same subnet, so that the watch cannot find the IP of the server.

Then we found another way, which uses data layer of the watch to communicate between devices. Again we implemented it, and it worked well. But this kind of technology can only connect them through Bluetooth. Because of this restriction, we finally abandoned this Bluetooth based technology.

So in the finished product, we use Google Cloud Message (GCM) to communicate between Android smart watch and the server. To do this, we registered the API key for Google Cloud Message on the Google service website, which also gave us the Sender ID and configuration file so that we could use that to send and receive.

Google Cloud Message is a simple and reliable messaging service which could reach over a billion devices. The GCM service handles all aspects of queueing of messages and delivery to client applications running on target devices. Server could distribute messages to the client app in any of three ways — to single devices, to groups of devices, or to devices subscribed to topics. Also, GCM provides a reliable and battery-efficient connection channel for clients to use downstream and upstream messaging.

4.1 Server Communication

In order to receive the instant message from the Google Could, our server needs to maintain a permanent connection between our backend and one of Google's endpoints. And Google has

chosen XMPP as the protocol of choice for communicating between its servers and our server. And we use the Smack library for XMPP protocol.

4.1.1 Receive messages from Google's Cloud:

First, our server registers a PacketListener object with the permanent connection. This object is responsible for all incoming traffic. The PacketListener has a method : processPackage() which takes a Packet as argument. And once the server receive the packet from the Google Cloud, we could get the desired information and do the correspond operator.

4.1.2 Send messages to Google's Cloud

To send the message to Google's Cloud Connection Server, we do follow these steps.
1.Create the JSON payload.
2.Create a Smack packet for the message.
3.Send the XML message to Google's cloud.

Google server provides some functions to support these steps, we could easily assemble the packet we desired.

4.2 Android Wear Communication

It is kind of easy to send from Android Wear. There is a GoogleCloudMessaging package provided by Google. We need to use that to register on the Google Cloud Message with the Sender ID. And then we will create an AsyncTask to handle the transmission through gcm.send().

In order to receive information pushed from the server, the app needs a WearableListenerService running in the background to listen to the push message from the server. As long as there comes a message, it will activate the onMessageReceived() method, which could handle the message receiving.

4.3 Packet Definition

We have 4 kinds of messages to send and receive between server and clients (Figure 4-1).

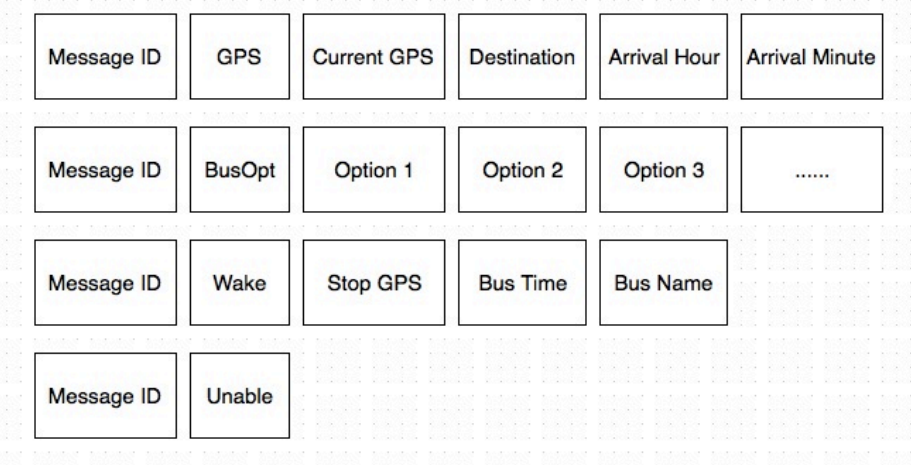


Figure 4-1 Four Kinds of Messages Format

5 Evaluation

5.1 Battery usage

To better illustrate the energy efficiency of our app, we use the app “Wear Battery Stats” provided by Google Play Store to monitor the real-time battery usage on smart watch. Besides, we run Google Maps app on the watch to provide a comparison.

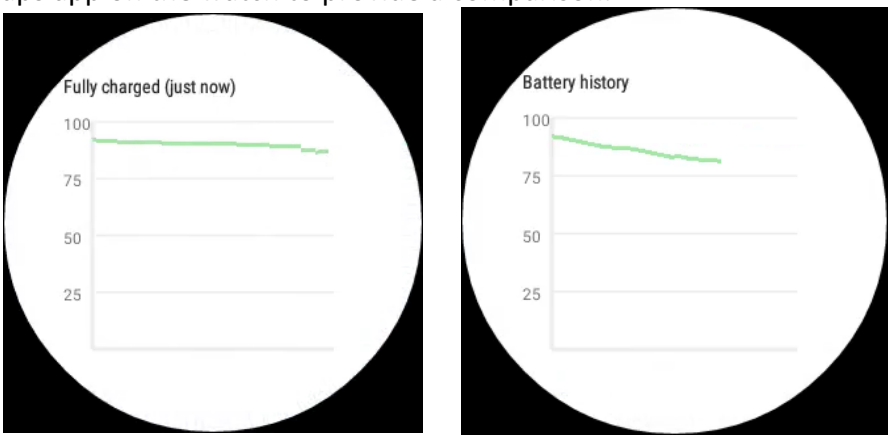


Figure 5-1 Battery Status by Blue Bus Tracker(left) and Google Maps(right)

From the figure above, we can easily find that there is about 5% battery drop during 20 minutes of usage, which is quite small. There is a little drop at the end because of the vibration notification and launching the Google Map. In fact, except the parameters input at the beginning and the server pushes the notification to the watch at the end, the smart watch is in sleep mode and there almost no battery usage of our app in most of the time. As comparison, the Google Maps consumes much more energy due to real-time navigation and GPS information.

5.2 Response time

The interaction between user and smart watch is quick and responsive due to optimizations on the hardware of the smart watch. In consequence, the slowest part during the use is the communication between the server and smart watch.

Because we achieve communication between server and watch via Google Cloud Message, and this push messaging protocol is quite slow. On Android with GCM, Google servers will batch notifications either 2 or 3 minute intervals. What this means is if there was a notification that was just pushed to your device, and the watch just gets another notification, Google's servers could delay that notification for a few minutes in order to batch other incoming notifications. This is done as a power optimization.

Other possible reason may be that Wi-Fi enters a low power state with degraded performance when the screen is off and the device is in sleep mode, or the Internet connection is poor and unstable, which makes the GCM response slow.

5.3 Accuracy

To test whether our Blue Bus Tracker can actually notify the user in time and let user catch the bus successfully, we did several experiments with trip plans. Specifically, we conducted 3 experiments in the morning around 10-11am, 4 in the afternoon during 3-5pm, and 2 in the evening.

Table 5-1 Experiment Result

Time period	Number of success / experiments	Reason for failures
10-11am	3 / 3	/
3-5pm	3 / 4	unstable data from Blue Bus API
11-12pm	1 / 2	too few bus during night

From the table above, our app can guarantee that the user can catch the bus in most of time. However, there are still several cases when failure occurs. The reason for the failure during 3-5pm was caused by the unstable bus arrival time retrieved from Blue Bus server. The time was stuck there and then has a huge jump after a while, so our algorithm failed to track this particular bus and did not notify the user in time. Another failure in the evening was due to too few bus in the late night so that the user cannot get to the destination ahead of the desired time. Our app pops out a notification card and shows the message of “Unable to catch the bus”. Under that condition, the user has to reschedule the travel plan.

6. Future work

Although our app could work in most of the cases, there are still several failures. And our app is just a rough model for blue bus tracking, which has some limitations. Considering wider users in the future and much more complicated scenarios, there are still several aspects that can be done the in future: In our app implementation, we retrieve travel duration time and walking time from Google Maps to locate roughly the blue bus to track and provide accurate time to user by Blue Bus API. However, the bus arrival time provided by Blue Bus API is not stable sometimes. In the future, we may combine more information from Google Map, or other fault tolerance algorithms to deal with these unexpected conditions.

(Please check the link for our demo: https://youtu.be/XNhtbomvssI)