Time Your Trip Design Document

Names: James Zhou, Yixin Zhao, Sirine Trigui,
Navjashan Singh, Johannes Gallmann

Introduction

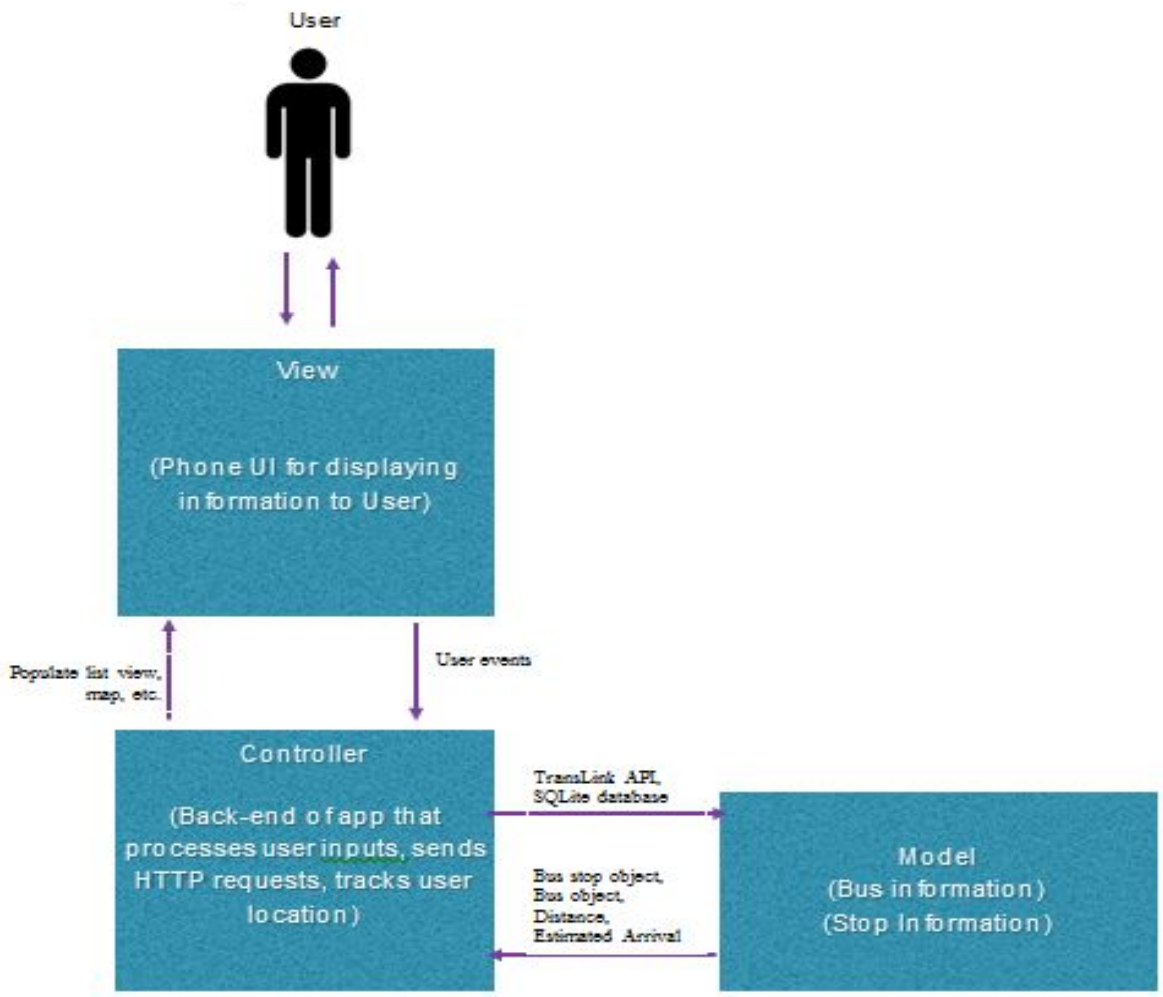
'Time your Trip' is an Android application which notifies the user when he or she has to get off the bus. The rationale behind this application is to increase the user's productivity while taking long bus trips. Instead of focusing on keeping track of their current bus stop while riding (for fear of missing a stop), the user can focus on other activities while the app will send a notification to the user when he/she is near arrival. To achieve this, the app will utilize the TransLink Open API to obtain relevant bus stop information such as GPS coordinates, buses servicing the stop, and real-time bus arrival and departure times around the Lower Mainland. The app will also utilize several other API features such as Google Maps API,Google Distance Matrix API for its tracking algorithm, and utilizes an embedded SQLite database that stores offline TransLink data regarding routes and stops information, to be used for the offline tracking feature.

Architecture and Rationale

The overall application is composed of the following major components:

1. Android application that processes user inputs and implements algorithm for tracking user location.
2. SQLite database that follows a relational model and contains bus routes and stops information.

Because our application focuses intensely on user interaction, we based our system design on the Event-Driven architecture, where the system detects and reacts to various user events. To emulate this architecture, we utilized the Model-View-Controller pattern:



- Data (Model): Bus stops, Bus routes
- An interface to view and modify the data (View): App UI
- Operations that can be performed on the data (Controller): Back-end

The MVC pattern:

1. The model represents the data to be displayed. We have several prominent customized data types, such as *Stops* and *Routes* data types. Information retrieved from either TransLink API or the SQLite database is generally packaged into these two types.
2. The view (App UI) displays the model data in a way that the user can visualize and interact with (i.e. a bus stop marker on the map). This project utilized the Observer pattern to further implement this view process.

3. The controller is the back-end that processes user events (button clicks, user inputs, etc.), retrieves information via HTTP requests or database access, creates the data types based on user inputs, and supplies them to the user via the “View”. The controller interacts with both the view and the model and controls the current “state” of the app (ie. map state, location tracking state, etc).

Android Studio

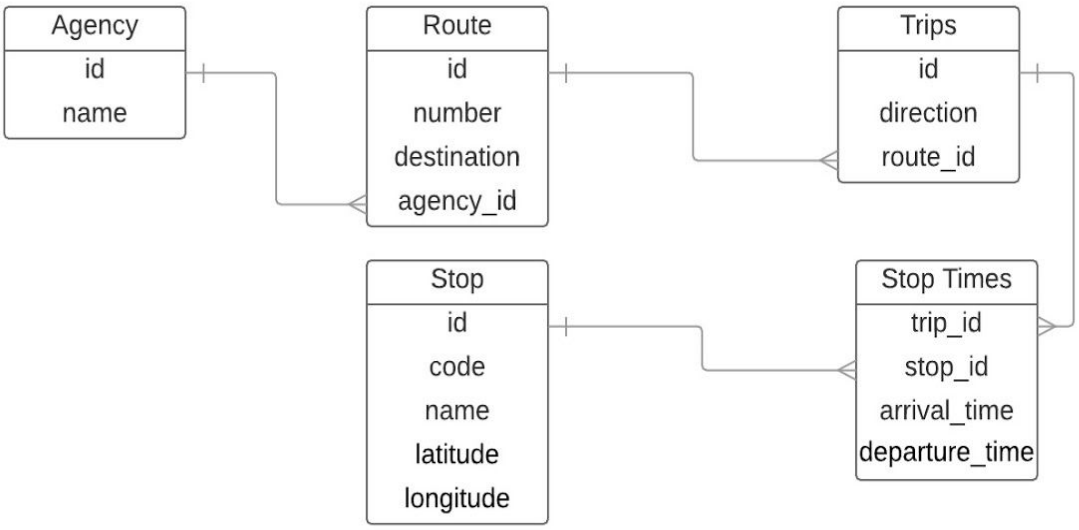
We chose to use Android Studio IDE because it is the environment that will be able to support all of our features for the project. Android Studio utilizes Java as the programming language, a language that is robust and supports Object-Oriented design. Object-oriented principles is crucial for the overall implementation of our system, and will be explained in more detail in the Detailed Design section. Android Studio also utilizes XML to create UI, and we will be using this feature to make the GUI for our app that will be visually pleasing and will guide the user through a series of steps to achieve the overall goal (setting an alarm).

SQLite

We will utilize an embedded SQLite database for storing offline TransLink information. The TransLink website contains static information such as *buses*, *routes*, *trips*, *stops*, and *stop times*, which are available as text files and are updated every week. We download the text files, organize different parameters into tables (ie. stops, buses), and create primary keys/id’s that link tables to one another (ie. link buses to stops). This relational database model forms the structure of our SQLite database, and we simply import it as a new database file into our program. Due to SQLite being an embedded database, it does not require an external server, allowing our program to access information offline.

Data

Our application retrieves relevant information from both the TransLink Open API and the embedded SQLite database. Information retrieved from the TransLink Open API is packaged into a Bus Stop object, with fields such as name, gps coordinate, arriving buses, and arrival times. Information regarding individual bus routes are retrieved from the embedded SQLite database, with parameters such as *name*, *bus number*, *route name*, and *stops in route*. As mentioned in the SQLite section, our group is using a relational database model for storing offline data in the embedded database. The Entity-Relation diagram below represents how our data is organized:

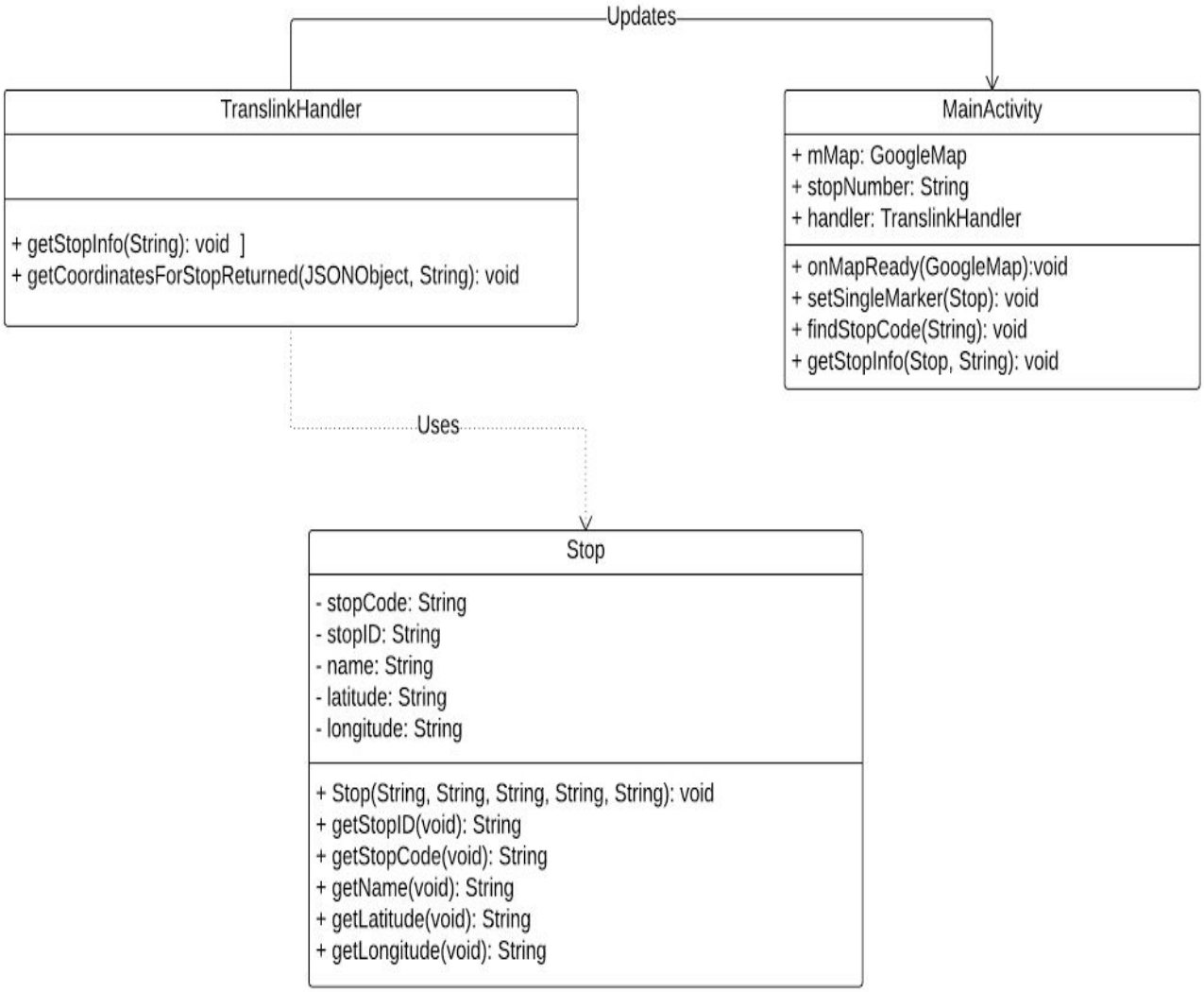


Every block represents a table in the database, and keys are used to link entities within tables together. For example, each trip contains a key value that matches the primary key of a particular route, thus grouping trips together based on routes.

Detailed Design

MVC

As stated above, the overall design pattern is composed of the Model-View-Controller pattern. This pattern allows our application to respond fluidly to user inputs, as the system must prioritize with providing information to the user. One example of the MVC pattern is our map display feature. The Model is the bus stop information being presented, and is represented by the *Stop.java class*. The view is represented by the MainActivity.java class that generates a map fragment. The purpose of this fragment is to display a bus stop on the map based on its GPS coordinates. The TransLinkHandler.java class represents the controller. It changes the data represented by the Model class based on user input (ie. generate a new Stop object based on the user's stop code input, and create a marker for said stop), and controls the View by creating new bus stop markers on the map and telling the View class to zoom into a bus stop marker chosen by the user. The class diagram for the MVC pattern is presented below:



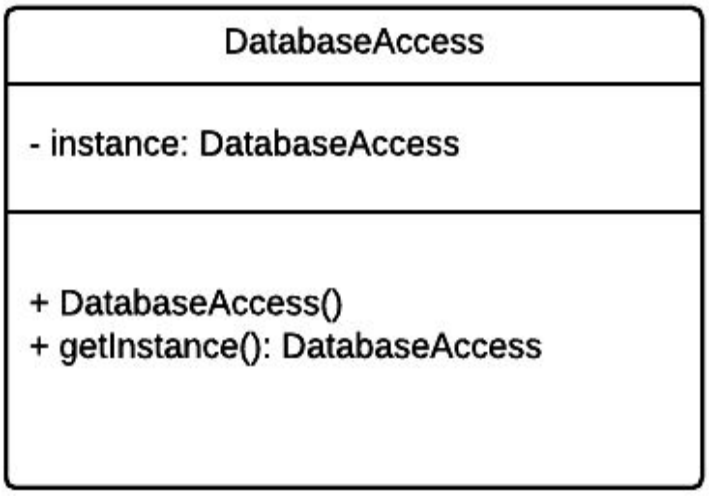
In this feature, the TranslinkHandler class retrieves information via HTTP requests to the TransLink Online database. The getCoordinatesForStopReturned() function retrieves a set of information in the form of JSON objects, and the function uses the Stop model format to create a Stop object, which is passed to the getStopInfo() function in MainActivity.java.

The MVC pattern is also present in features such as: generating buses for a selected bus stop, and generating stops for a selected bus route. The only difference is that a different mode (i.e. the Bus.java class), a different controller (i.e. GPSHandler or DatabaseAccess) and a different View (various Activity classes). One weakness in our overall design is that we failed to exploit encapsulation for the various controller behaviours such as DatabaseAccess and TransLinkHandler. If we had more time, our group would've chosen the Strategy pattern for representing our controllers, where a high-level controller interface would be implemented, with

various controller behaviours extending it. Every activity will have a Controller variable that will be assigned a specific behaviour for runtime, exploiting polymorphism and class composition.

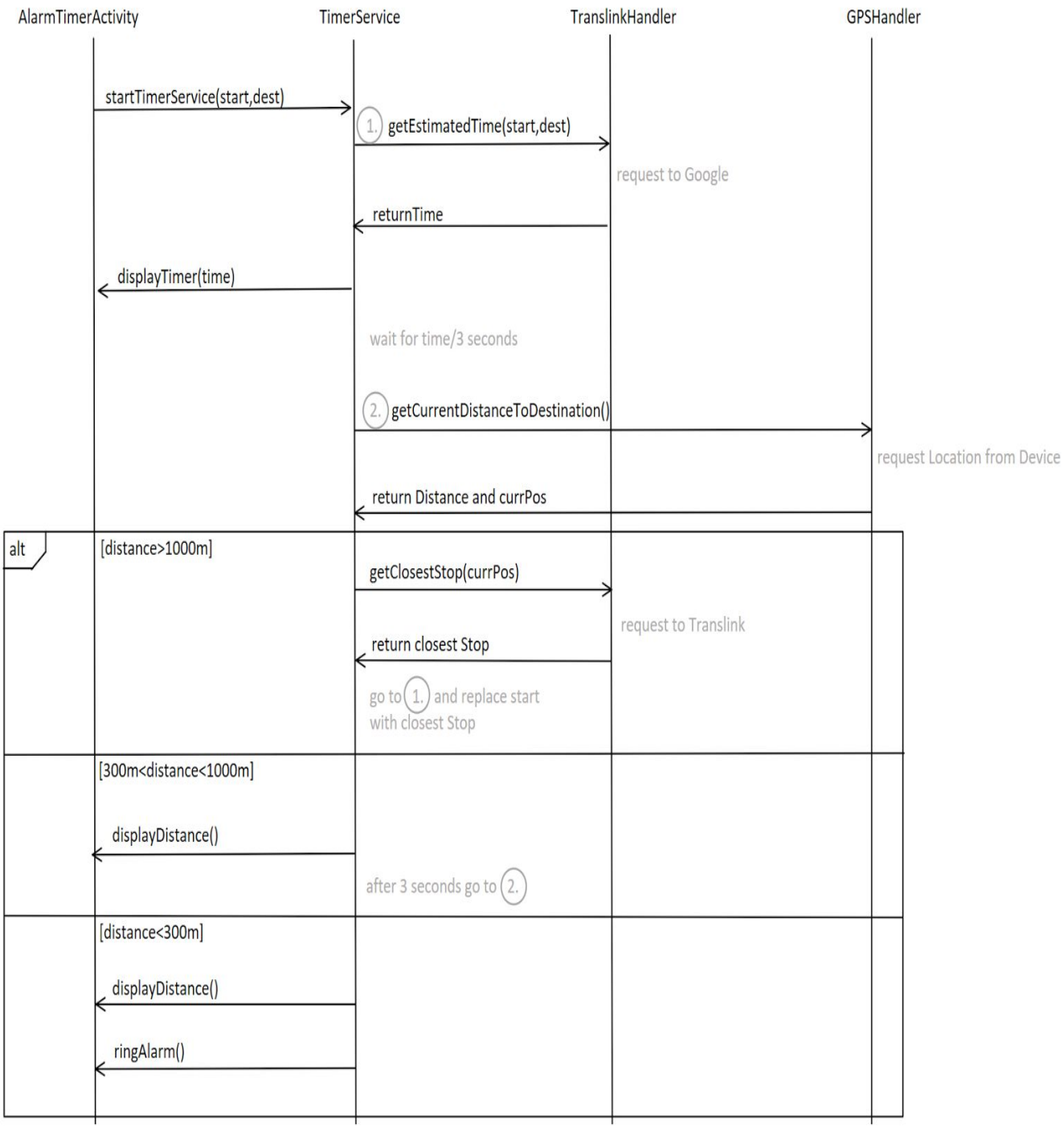
Singleton

Because our app needs to access the embedded SQLite database to return data to the user, we have created a database helper class to process requests from different activities. For this class, the Singleton pattern is implemented to ensure that only one instance of the database helper is present at any time during the system lifecycle. This is due to the fact that creating more database helper classes will create more instances of the database itself, which is a waste of resources considering that the application is only using a single database as an information source throughout all the activities. Furthermore, to connect/disconnect from the SQLite database, the helper class must call open/close operations in order to interact with a writable database; the open operation is called at the start of every activity, and the close operation called at the end. Creating more instances of the helper class will create open/closes imbalances (ie. close operation may not be called for every instance of open), resulting in memory leakage during runtime. The Singleton pattern for the database helper class is represented in the following class diagram:

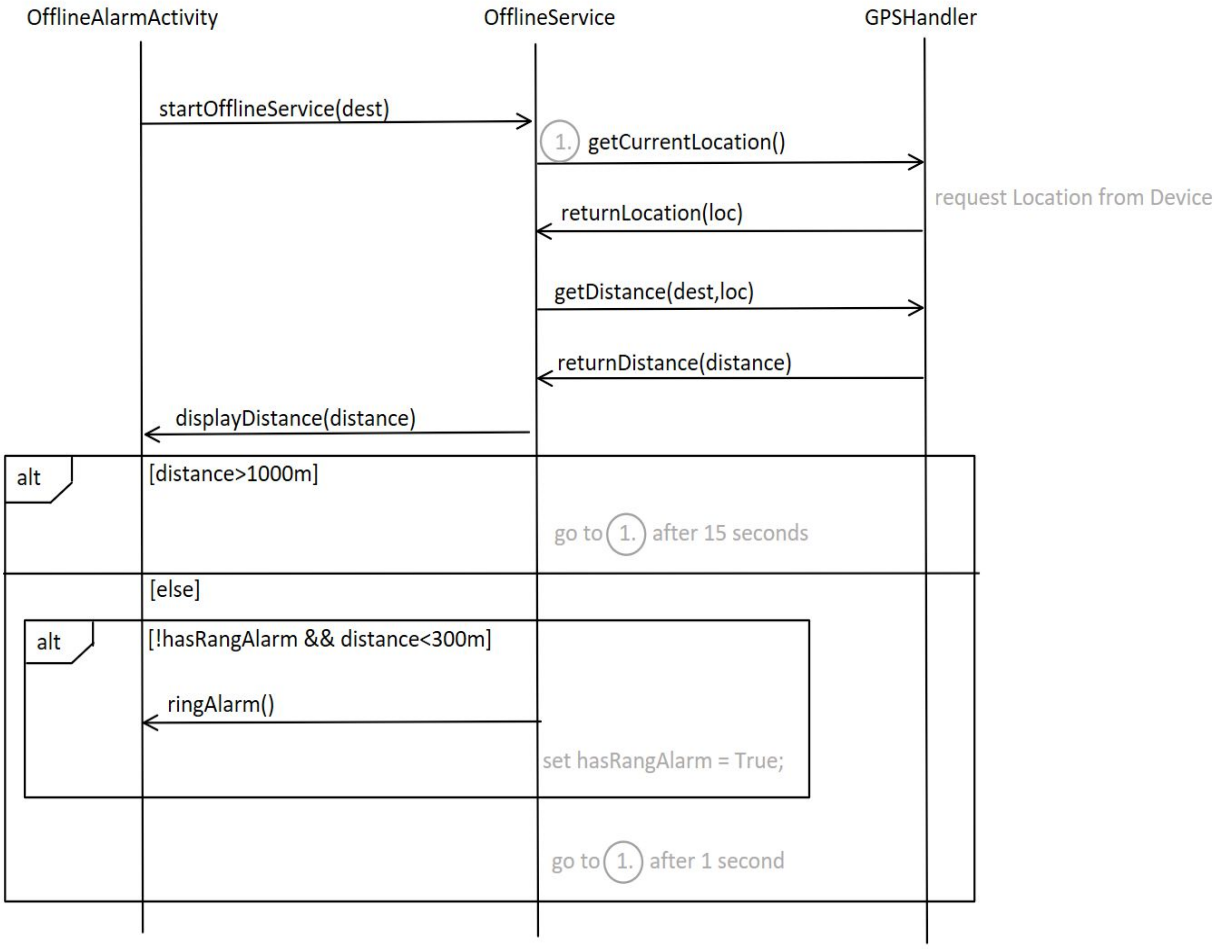


Dynamic Description of System

The following two sequence diagrams provide a slightly simplified dynamic description of the algorithm that keeps track of the user's position, updates the display and triggers the alarm during the bus ride. The first version handles the case where the user has internet connection and thus gets provided with an estimated arrival time.


AlarmTimerActivity
TimerService
TranslinkHandler
GPSHandler
startTimerService(start,dest)
1. getEstimatedTime(start,dest)
request to Google
returnTime
displayTimer(time)
wait for time/3 seconds
2. getCurrentDistanceToDestination()
request Location from Device
return Distance and currPos
alt
[distance>1000m]
getClosestStop(currPos)
request to Translink
return closest Stop
go to 1. and replace start with closest Stop
[300m<distance<1000m]
displayDistance()
after 3 seconds go to 2.
[distance<300m]
displayDistance()
ringAlarm()

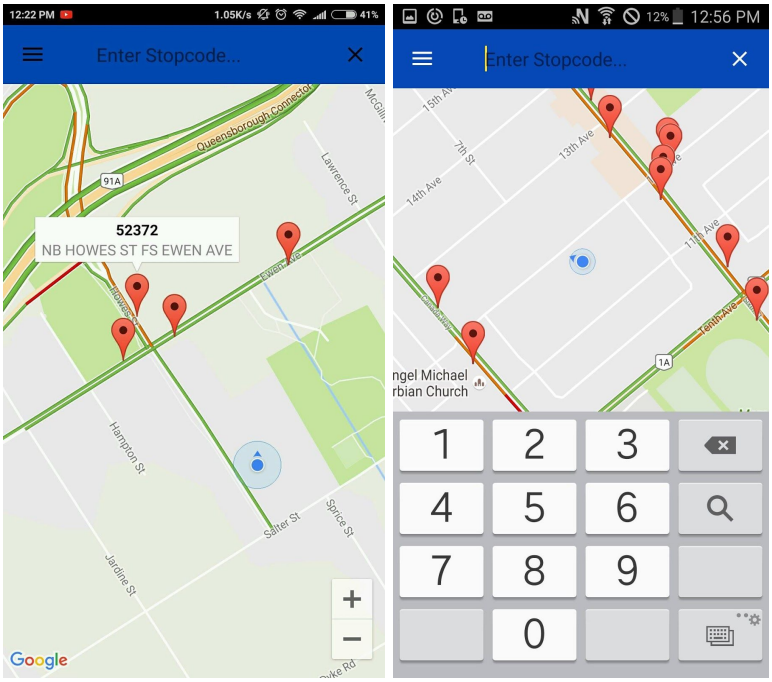
The second diagram handles the case where the app has only access to the GPS module to keep track of the user



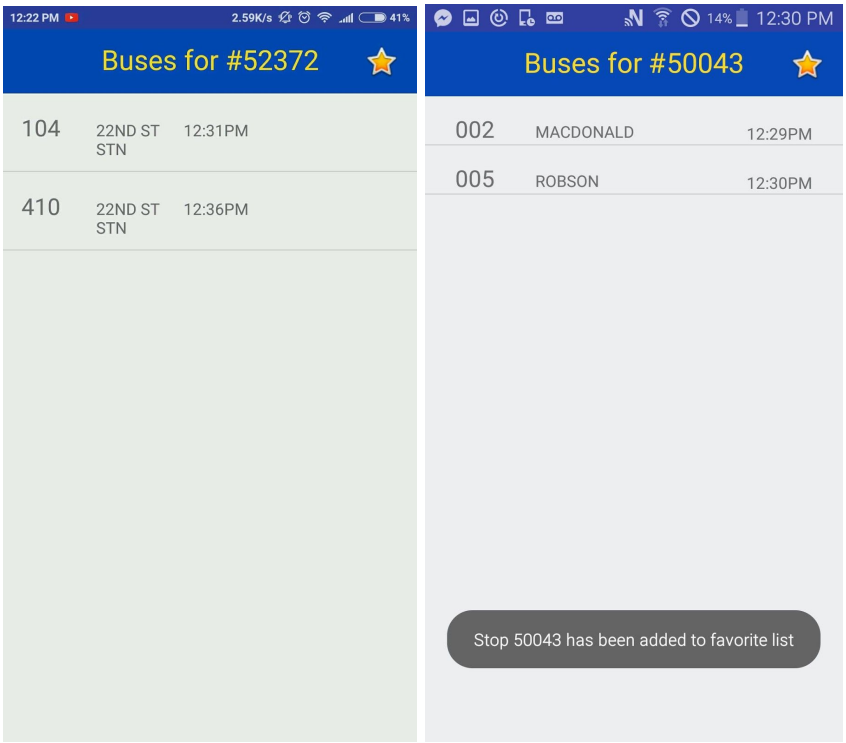
GUI

The user interface is built using Layout Editor of Android Studio, which provides developers with preview of their design.
(https://developer.android.com/studio/write/layout-editor.html)

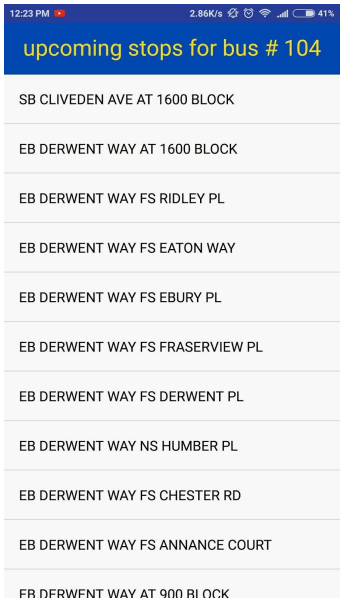
Launcher Page and Inputs:
1.As soon as the user launches the App, the user is going to be asked for selecting a bus stop number from the map, where he/she is at; or the user can search for a bus stop number.



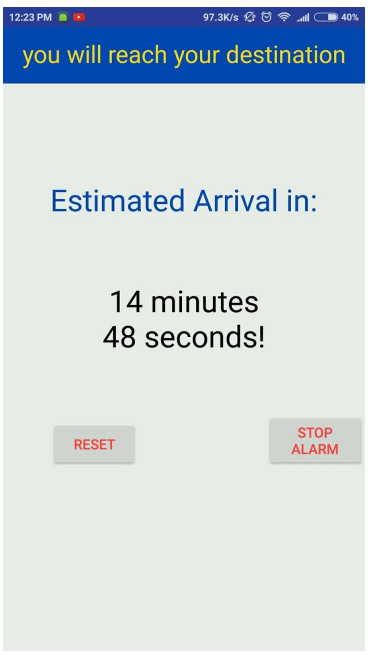
2.Pressing on a bus stop number on the map, the user is going to be directed to a new window where they can select the bus number they would like to take.On this page, the user can also add the bus number as a favorite.

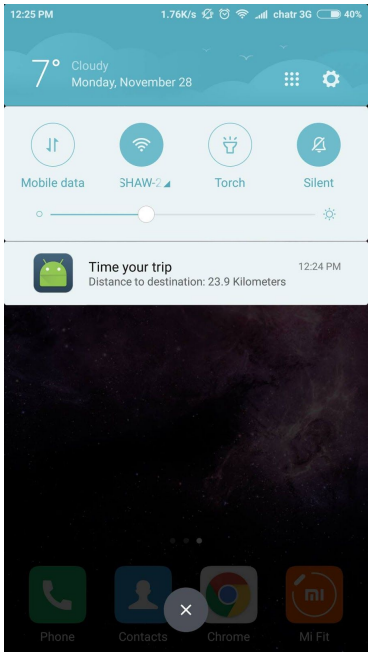


3.Having select a bus, a list of upcoming bus stops will be provided so that the user would be able to select their destination stop .



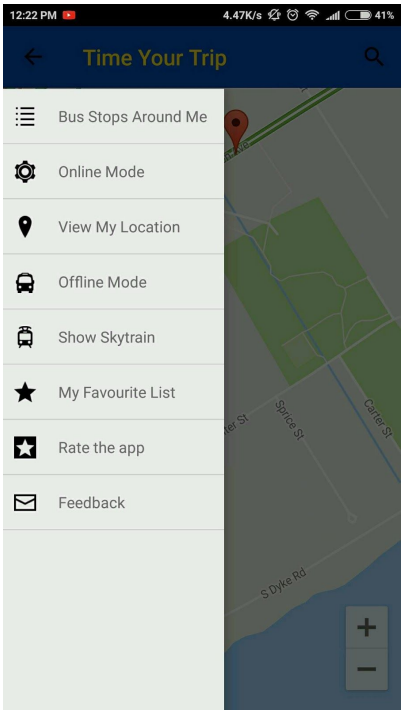
4. Having chosen a destination stop, the alarm will be set automatically and the user will be directed to a window where the estimated arrival time will be displayed with an option of clicking on “Stop Alarm” button if they want to mute the alarm or “RESET” if they want to start over. If the user exit the App, a notification will be shown on the quick access menu.



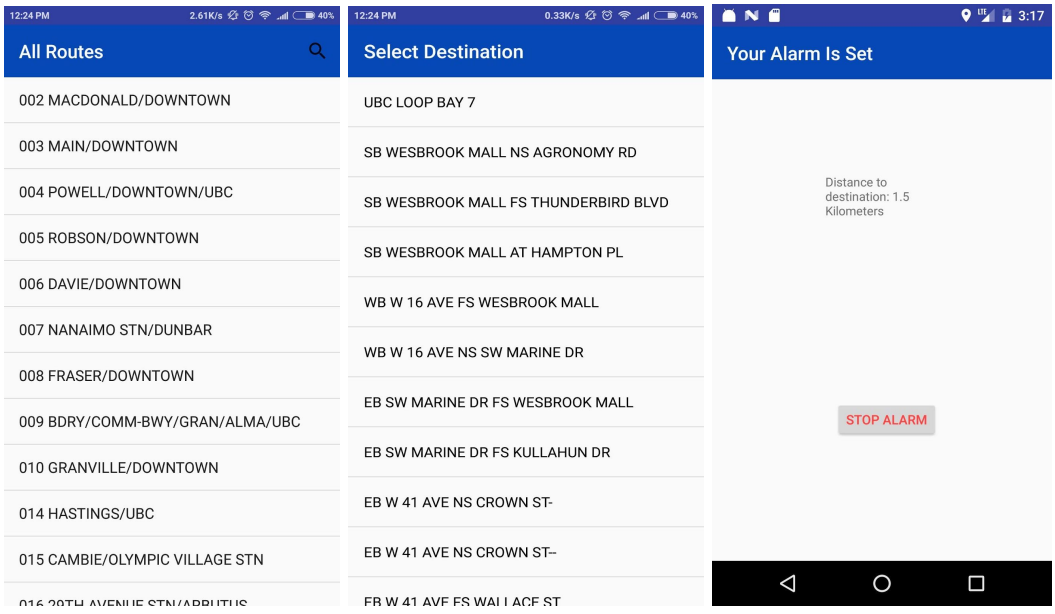


4.a) In case the user clicked on “RESET”:
Then the user will be directed to the launcher page and all the previous input will be deleted
4.b)In case the user clicked on “STOP ALARM”:
The alarm will be muted and stop calculating

On the launcher page, we also have a sliding menu for the user to access our different features.



We added an offline feature in case the user don't have access to internet.
After press the Offline Mode from the sliding menu on the launcher page, the user will be provided with a list of all the routes, and he/she can search for the wanted route by using the search bar on the top. Then, the users will be able to choose a destination stop from all the stops of that bus.



Validation

Architecture

We used MVC(Model, View and Controller) approach as the system architecture for our app. After discussion among the team members, the way we wanted our app to work is that it will take some data from the user and will compute the time for the trip based on that. As we need to retrieve real time data from the translink in order to ensure the correctness of the app, we used Translink API in the model of our app which could be accessed by sending queries and parsing the data returned in the JSON format. Apart from that, as we also need to calculate the time which the user will take to reach his/her destination, we used Google Maps API for it that can return the time that will be taken by travelling on public transit and is also a part of our app's model. We wanted to ensure the reliability and adaptability in our product which is the reason we decided to make both the API's a part of the model of our system. We found that it will be hard for the user to know the stop just based on the stop number. Also, in order to provide the user with an offline functionality, as a lot of transit users don't have internet connection, we also need to have a database which we could use to retrieve information such as bus stop names and to implement the offline functionality for our app. Therefore we added a Translink's GTFS database as a SQlite database in our app which will get installed on user's phone when the user installs the app. This database will serve as the controller for our app as after getting data from the view of the user, we can access our database to get relevant information such as bus stop names. This database could also be used to get the GPS coordinates of the destination stop and an alarm could be set to alert the user that he/she is about to reach their destination. For the view or the user interface, which is one of the most important factor to be considered for the graphic design of the app, is explained below -

User Interface

The UI validation is a continuous process in which the client was involved. The selected theme was discussed with the client and user in the process of the project and was approved. For the view, we required to provide the user with an easy and interactive user interface due to which we used a map with which the user can easily locate and choose a bus stop. For the search fields of the offline feature, we provided the user with an auto complete field feature which allows the user to easily select a field. For choosing and selecting from the information, we provided the user with a list view. The design progress is presented on a weekly basis and client feedback taken into account to design the user interface as user friendly as possible.

Appendix I: Installation Guide

Because our app is currently undergoing testing phase, it is not available publicly on the Google Play store. To install from the Play Store, please follow this URL and follow the instructions to download:
https://play.google.com/store/apps/details?id=com.planmytrip.johan.planmytrip
Time Your Trip is currently only available on Android devices, and is optimized for mobile phones running OS version Lollipop and above.

Appendix II: Developer Section:

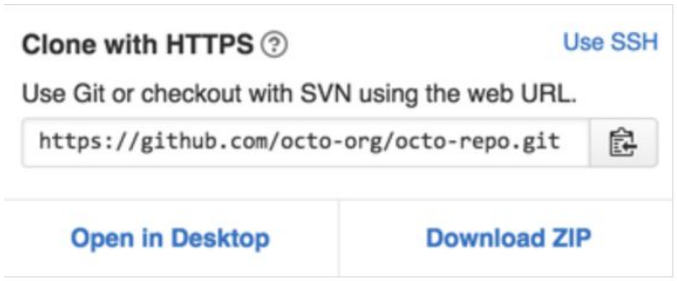
What is Time Your Trip Application:
This Android mobile application notifies bus passengers when they are arriving at their destination stop. It allows the user to spend the bus ride doing other activities, such as studying and sleeping, without having to constantly check the current stop and without fear of missing their destination stop.
The app operates such that when the user gets on a bus, he or she will enter the starting stop he/she at, then the app will allow the user to choose the bus number and the destination stop. After that, it accesses real-time TransLink information to retrieve the estimated time it will take for the user to reach the chosen stop. Then, the app will set an alarm based on the retrieved time.

How to find our source code:
Our source code is on a public github repository, the following is link to Time Your Trip Application:
https://github.com/Farwhyn/cpen321project

How to check it out:
This procedure assumes you have already created a repository on GitHub, or have an existing repository owned by someone else you'd like to contribute to.

1. On GitHub, navigate to the main page of the repository.

2. Under your repository name, click Clone or download.



3. In the Clone with HTTPs section, click to copy the clone URL for the repository.
4. Open Git Bash.
5. Change the current working directory to the location where you want the cloned directory to be made.
6. Type git clone, and then paste the URL you copied in Step 2.
7. git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORY
8. Press Enter. Your local clone will be created.
9.
```
git clone https://github.com/YOUR-USERNAME/YOUR-REPOSITORYCloning into
"Spoon-Knife"...
remote: Counting objects: 10, done.
remote: Compressing objects: 100% (8/8), done.
remove: Total 10 (delta 1), reused 10 (delta 1)
Unpacking objects: 100% (10/10), done.
```
10. The code is under “PlanMyTrip” folder.

How to build the project:

There are two options for building the project:

- Through an emulator:

This procedure assumes that you have already installed Android Studio.
To start an Android emulator such as the default emulator installed in RAD Studio:

1. Tools->android ->Start the Android SDK Manager (select Start. ...
2. In the Android SDK Manager, click the Tools menu and select Manage AVDs.
3. In the Android Virtual Device Manager, select the emulator and click Start.

The app will be uploaded automatically and it can be tested starting by providing whatever expected.

You can download our app from https://play.google.com/apps/testing/com.planmytrip.johan.planmytrip , or use Google Play and search "Time My Trip"

How to run tests:
After opening the project, switch to the Project first. There are two folders that contain different tests for our application:

Application Tests

1. In the project source code, go to PlanMyTrip\app\src;
2. Under the “andoridTest” folder, click on “java”, then the “com.example ….” folder, where the Application tests are;
3. To run the application tests, click “Run” or right click on the file and select “run ‘ApplicationTest’”. Then Android Studio should open the emulator;
4. Choose a virtual device. If don’t have any
5. Click on the “create new virtual device” button;
6. Under the “Category”, select “Phone”;
7. Choose a device then click on “Next”;
8. Click on “Next” on the following windows then, at last, click on “Finish”;
9. Wait for it to be created then it should be seen on the window.
10. Click “OK”
11. The result will show at the bottom “Run:” console.

Unit Tests

1.In the project source code, go to PlanMyTrip\app\src;
2.Under the “test” folder, click on the “java” then “com.example...”;
3.To run the unit tests, right click on the file and select “run ‘NameofTheTests’”;
4.The result should also shown on the bottom “Run:” console.

Description of Classes used in the Project

alarmTimer: this activity is responsible for turning on/off the alarm
AllRoutes:this is used when the offline feature is selected so that it returns a list view of all the buses with an autocomplete option
Bus: this activity is providing the user with infos regarding the bus(route, bus number,estimated leave time and the destination)
ConnectDatabase: generate the list of the upcoming bus stops provided to the user
DatabaseAccess: using the created database to send queries so that a listView can be generated
DatabaseOpenHelper: creating the database
Favourite: this is responsible for the favourite list where the user can save their frequent bus stops.
GPSchecker: this activity is for checking whether the GPS mobile is enabled or not
GPSHandler: is for checking how far the user is from their destination stop
MainActivity: this activity is responsible for the launcher page, where the user is expected to enter the bus stop number he/she is at
NextBusesAdapter: this activity is to generate the list view of the buses provided to the user after he entered the starting bus stop
NextStopsAdapter: this activity is for returning the upcoming bus stops
OfflineAlarm:this activity is for setting the alarm when the offline mode is selected.

OfflineService: this activity is for the offline feature
OfflineStops:this is for generating the bus stops for the offline mode.
Stop: this activity will provide the location of the selected stop
SwipeDetector: this activity is for deleting a bus stop from the favourite list by swipping it left.
TimerService: this activity contains the algorithm of the time, check the dest, and update the time in the online mode
TranslinkHandler: this activity is where we are extracting the infos from Translink API and parsing them to a JSON object or JSON array
TranslinkUI: this activity is for providing the user with the buses that they can take from their starting location

Design patterns used:

Model-View-Controller pattern

Various examples of this pattern is used throughout the system, such as generating a marker on a map for a user-input bus stop, or generating a list of bus stops for a selected bus route.

Singleton pattern

Is used to design the database helper class.