ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE
Department of Informatics – Science and Engineering DISI

BACHELOR'S DEGREE
IN
COMPUTER ENGINEERING
Class L-8

GRADUATION THESIS
in
System Administration Laboratory

A Travel-Guidance Engine for Visually Impaired People

CANDIDATE
Stefano Gasperini

SUPERVISOR
Prof. Marco Prandini

ASSISTANT SUPERVISORS
Dott. Andrea Melis
Dott. Saverio Giallorenzo

Academic Year 2015-2016
Session II

Abstract

Until a few years ago, using public transportations could be confusing and required understanding of the local transit system. Then, with the diffusion of location aware devices, mobile data networks and Google Maps (former Google Transit), everything changed, making possible to plan a trip away from home. Even though Google Maps can provide directions almost worldwide and offers plenty of information, some features like real-time integration are not available in every city, but are based on arrangements with local transit agencies.

In this thesis I present GoGoBus: an Android application that provides transit support for the city of Bologna, Italy. By combining several services, GoGoBus addresses the needs of different kinds of users: offering planning for new or infrequent riders, real-time arrival information for frequent transit users, plus vehicle tracking, voice support and a simple UI for riders with special needs. Specifically designed for visually impaired people, the application's most innovative aspect is its support during the trip, which is also integrated with travel planning and real-time arrival information. The system tracks the bus through the use of the mobile GPS, its location is used both to detect when a stop passes and also to provide useful information like the distance to the next stop, the number of remaining stops, the time before getting off and most of all to notify the user when it is time to get off. The idea behind GoGoBus is to increase the usability of public transit for visually impaired riders, as well as new or infrequent ones, dramatically enhancing their independence, while improving the service quality for commuters and frequent riders.

Abstract [Italiano]

Fino a pochi anni fa, usare i trasporti pubblici poteva essere fonte di confusione e richiedere la comprensione del sistema dei trasporti locali. Più tardi, con la diffusione di dispositivi con localizzazione GPS, reti dati cellulare e Google Maps (inizialmente Google Transit), tutto è cambiato, rendendo possibile la pianificazione di un viaggio mentre si è fuori casa. Nonostante Google Maps disponga di indicazioni stradali più o meno in tutto il mondo e mostri molte informazioni, alcune funzionalità, come l'integrazione degli orari in tempo reale, non sono disponibili in tutte le città, ma sono basate su accordi con le agenzie dei trasporti locali.

GoGoBus è un'applicazione Android per l'ausilio al trasporto nella città di Bologna. Combinando diversi servizi, GoGoBus si rivolge a svariati tipi di utilizzatori: offre la pianificazione per i meno pratici del sistema e coloro che usano i trasporti pubblici raramente, dispone di orari in tempo reale per chi usa i mezzi frequentemente, e in più traccia la posizione dell'autobus, ha un supporto vocale e un'interfaccia semplice per persone con disabilità. Progettata appositamente per ipovedenti, l'aspetto più innovativo dell'applicazione è il suo supporto durante il percorso sull'autobus, integrato alla pianificazione del tragitto e agli orari aggiornati in tempo reale. Il sistema traccia la posizione dell'autobus attraverso il GPS del dispositivo mobile, la cui posizione è usata sia per riconoscere quando una fermata viene superata, sia per mostrare informazioni utili come la distanza dalla prossima fermata, il numero di fermate e i minuti rimanenti prima di scendere, e soprattutto notificare l'utente quando deve scendere. L'idea dietro GoGoBus è incrementare la fruibilità dei trasporti pubblici per non vedenti, ma anche per persone che li usano di rado, aumentando ampiamente la loro indipendenza, allo stesso tempo migliorando la qualità del servizio per chi usa i mezzi quotidianamente.

Table of Contents

Introduction

The first version of a Google's Transit Trip Planner was developed in Portland in December 2005. To be part of the project, the transit schedules and associated geographic information needed to be delivered to Google in a specific format, Google Transit Feed Specification (GTFS). This format became a de facto standard and revolutionized transit planning. It made possible to extend solutions valid for a single operator to any other area in which GTFS data was available. Since then, several mobility services have been developed to support public transit users and traveling in general.

Nowadays, Google Maps planning service has a very wide coverage and seems to offer everything possible to support the travelling. However, some useful features are not available everywhere in the world. Recently in Bologna, when requesting a transit trip, Maps declares "We don't have the most recent timetables for this area". Moreover, real-time ETAs, are not available on Maps in Bologna. In addition, Google Maps support while riding the bus is limited to showing the list of stops the bus will go through and displaying the current GPS location on the map, but there is no tracking for the stops passed, and nothing that notifies the user when it is time to get off.

In order to display on Maps real-time information, transportation agencies must provide to Google live updates on their fleet in a specific format. Despite the local travel agency in Bologna has a real-time tracking system on every bus, the information is not on Google Maps, but is available to the public and to developers.

The main objective of this thesis project is the development of an application, based on the Android platform, to support visually impaired people using public transit. This application, through the integration and combination of several services, must provide all the information that the user needs in order to travel by bus and tram safely and efficiently. It should also increase the usability of public transit, by being useful for every kind of rider.

The target is to create an application with planning features, live updates on delays and, most of all, support while riding the bus. By using local information, the application will work only in Bologna and its suburbs, but it will be tailored for its population and provide in a single package all the available services related to public transits. Then, since the data provided by the local transit agency is in the GTFS format, it will not be too complicated to extend the coverage to other cities, retrieving information from their transportation servers. The application will serve as a model for service integration and trip support.

Requirements Analysis

In order to be an efficient mobility service, suitable for visually impaired people and useful to all kinds of users, the application must meet specific requirements. In order to make public transportation easier and more appealing, riders who are new, infrequent, or vision impaired users would all benefit from an easy-to-use system with planning capabilities. In addition, passengers during their trip could also be guided by a service that notifies when it is time to get off, and some kind of information about the distance to the destination. Moreover, visually impaired people also necessitate of a dedicated interaction method, like spoken texts instead of written ones and views compatible with the Android accessibility service. Finally, also frequent riders can benefit from real-time features. Indeed, they already know what is the best transit path to get to their destination, but they might want to choose the best route considering the current delays. All the mentioned features need real-time updates of buses estimated time of arrival (ETA).

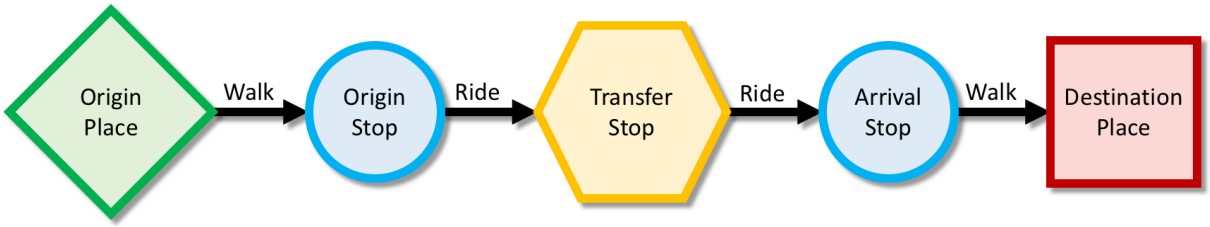

Figure 1. Trip with Transfer: diagram showing the various steps.

Figure 1 shows all the steps and places involved in a transit trip with a transfer. The application should provide solid support from the origin to the destination. Apart from the support during the travel, also the support at the stop is very important. Passengers generally have to wait for buses during their trips. In fact, transit riders spend outside buses a large portion of their total trip time. For this reason, the time spent waiting and transferring between rides is important to provide a better experience to the rider.

In 2009, a collaborative research study between the transportation institutes of University of California Berkeley and University of California Los Angeles analyzed the opinions and perceptions of transit riders about waiting and transferring[1]. The surveys were taken in Los Angeles County, with 750 riders interviewed. The results show that the users ranked the importance of having a short wait almost the same value of having a way to get help in case of emergency, 70% and 74% respectively.

By providing to customers accurate ETA of buses, they can schedule their arrival to the stop according to the time they are willing to wait. An option to do this is to use display signs, however, these signs require not only to build the bus tracking system, but also to install the actual led signs and maintain them. This is why in cities like Bologna, at-stop displays are available only at a limited number of stops, the most frequented ones. As the sign coverage is limited, to reach the rest of the customers, online or text message services are also available.

A research study from the University of Washington, Seattle, analyzed the impact of mobile real-time information on waiting times, by comparing perceived and actual times[2]. The research

results show that riders with static data source perceive on average a waiting time that is 15% longer than the actual time. On the other hand, riders with real-time info, estimate correctly the time spent waiting. Moreover, if real-time ETAs are available, the time spent at the stop decreases as the rider knows the bus arrival time and plans the walk to the stop accordingly.

Another research study also from the University of Washington, examined the opinions of users of OneBusAway, a system which provides real-time arrival information for public transit in Seattle (2009)[3]. The results are based on 488 people interviewed in an initial survey, plus 139 respondents from a follow-up walking survey. Users resulted being more satisfied with transit (92%), 48% of which said they were much more satisfied, because waiting times were reduced and transit trips per week increased using OneBusAway. More than a fifth of the interviewed population perceived an increase of their personal safety (21%), by spending less time at the stop and 78% reported that the system increased their walking, as by knowing the bus ETA, they could walk to a different route or along the same route, either further down or back up. Considering these results, it is clear how a relatively simple system can increase riders' satisfaction, save them time, and increase public transit usage.

To summarize, the main requirements that can be elicited for GoGoBus are:

- to have an accessible user interface integrated with the text to speech function, specifically designed for visually impaired people;
- allow the creation and management of travel profiles, intended as pairs of places (origin, destination);
- show several different bus routes to choose from, given a travel profile;
- show detailed information related to times, stops and lines;
- show and update buses estimated time of arrival obtained real-time through GPS tracking;
- have a real-time tracking system to show previous and next stops, ETA and the number of stops to go by using the device GPS;
- support routes with multiple legs.

In order to provide all these features, it is fundamental to use different online services and combine them. Due to the current services availability, only by integrating data from several sources it is possible to create such application.

Android

Android is an operating system developed by Google[4]. It is based on a Linux kernel and it is designed for touchscreen mobile devices like smartphones and tablets. Founded in 2003, the first release came out in September 2008. Since then it has gone through many updates and improvements that made it the most used operating system in the world with a total of over 1.4 Billion active devices by month in September 2015[5]. Android dominates over the 54% of the OS market, considering smartphones, tablets, PCs and laptops all together[6].

We decided to develop this system on Android because of several reasons. First of all, it is a very successful application environment. Even though publishing apps requires the payment of a one-time fee of 25 USD, developing on Android is free of charge. In addition, it also has features to help people with special needs.

Android also has a built-in accessibility service called TalkBack[7], developed by Google. It helps visually impaired users to interact with their devices. Once activated from the accessibility settings, it helps by giving spoken and vibration feedback when the user touches elements on the screen. Android also has a built-in dictation feature: available in many different languages, it can be used to write in any text field, while online but also offline, prior download of a package.

From a developer point of view, Android code is written in Java and uses XML to define layouts and other graphic elements. XML stands for eXtensible Markup Language and it defines the rules to encode documents in a format that is readable both for humans and machines. However, many external libraries use code written in other languages, but coding using the libraries provided by Google requires only Java and XML competences.

To develop applications in Android, developers can rely on many built-in libraries and then post their apps on the Google Play Store[8] so that users can download them. Moreover, Google provides a powerful and easy to use IDE called Android Studio, which allows developers to build apps for every kind of Android device, from tablets to smart watches. Android Studio has a useful tool that makes possible testing and debugging of the app under development directly on a real Android device connected through a USB data cable. This is very useful, instead of using an emulator, also available, because it can quickly show how the app performs in its real ecosystem dealing with real network issues, GPS, memory and other problems that can occur. Testing on a real device also allows the developer to interact with the app using the device touchscreen as the final user will do.

GoGoBus uses many Android technologies and components spacing from the simple Activity to ServiceIntent, SharedPreferences, TextToSpeech and many others.

Related Work

GPS has been integrated in cell phones since 1999, the first full internet service on a mobile device was provided also in the same year. Since then, both technologies and also the mobile devices went through extreme renovation and improvement, giving people all new possibilities while traveling. The interest of modern societies in this kind of technologies has been relevant and is still increasing, attracting huge investments, that made possible such development.

Several systems have been developed to aid public transportation riders using mobile technologies. Most of them are the results of university research projects, that were later abandoned and never updated to the current technologies.

BusView, developed at the University of Washington, in Seattle, is one of the first real-time information systems for public transit[9]. The system was able to track the buses and show their location on a map. Developed in 2000 only as a computer application, it was not available on mobile phones.

Opportunity Knocks was also developed in Seattle at the University of Washington. It was an automated transportation routing system for people with cognitive disabilities[10]. Made in 2004 to increase independent and safe use of public transportation, the system works on cell phones, has an associated GPS sensor to carry around and is able to predict the future trips based on previous history. The system had a learning algorithm and asked the user how to get to a chosen place if different travel options were used in the past to get to the same destination.

Travel Assistant Device (TAD) is also for people with cognitive disabilities[11] [12]. Developed for a couple of years at the University of South Florida and presented in 2006, the system works by setting the travel profile remotely and then prompting on a cell phone simple and short instructions to tell the user that the bus is coming at the stop or that the destination is close.

Also for people with cognitive disabilities, Mobility Agents was developed in 2006 by a company named AgentSheets in Boulder, Colorado[13]. The idea is again to give on a mobile device short and simple instructions only when needed. It has the possibility for the caregiver to track real-time the rider with a 3D visualization.

All these projects seem to be abandoned and not further updated, however, it is not the same case for the following.

OneBusAway started in Seattle around 2008 as a research project at the University of Washington by Brian Ferris, Kari Watkins and Alan Borning (advisor) [14]. The system, still updated and in use today (last update on September 27, 2016), provides real-time updates for bus ETAs (plus travel planning only in Tampa Bay, Florida). Available on many different platforms, including iOS and Android[15], it covers a few cities spread across North America. First available in Seattle, it then reached the East Coast and is available today in 7 different areas mostly in the United States, but also one in Canada[16]. Its coverage is expanding and the application counts today more than 500K downloads only on Android and has thousands of users every day. OneBusAway shows the stops and their direction on a Google map and integrates real-time updates provided by the travel agencies to show in a single screen all the live delays for the routes passing through a specific

stop, when available. It also has the possibility to star favorite stops and routes. Its features are determined also by surveys and usage stats[17].

Another university project currently updated and developed, originated by the professor Katia Obraczka as a senior design project[18]. The result is a network to track buses real-time moving around campus at University of California Santa Cruz. Available via a mobile friendly web app, a constantly updated map shows real-time the campus buses and stops[19]. The project is called Slug Route and demonstrates in a small scale, what should be applied to every area to improve public transit. Every campus bus mounts a hardware GPS transmitter, while base stations collect the data. The platform on the bus allows to choose the route, so that the web app can show the correct label.

Online Services

SMAll Platform Concept

The core of the SMAll platform is a series of services to build a marketplace of services[20]. Centered on mobility the services space from Planning to Booking and Traveling. Still under development, the key idea is to offer a single system that, thanks to the cooperation of smart mobility operators, provides several services. Smart Mobility is a new concept of mobility based on services able to monitor, store, analyze and understand data. The services can also provide customized solutions to the problems derived from the data analysis. SMAll includes standard interfaces to access data and use its services, an infrastructure for service development and deployment, toolboxes for statistical usage analysis, the definition of a model specific to handle the services, plus a set of basic services. Regarding public transit, SMAll provides interfaces to retrieve many information about public transportations and can also perform actions such as tickets booking and payments. For example, by analyzing the user's transportation history, the system could suggest to buy a subscription if the travel habits would make it convenient. GoGoBus can be somehow seen as a simplified example of this idea of services combined to offer a more complete service.

All the online services presented below have in common the format of their data. All the server calls related to the services return a JSON, which stands for JavaScript Object Notation and is an open standard format for the data exchanged between client-server applications[21].

Travel Planner

There are many planning services available to choose from, Google Maps also provides a third party developer interface to retrieve all the information they show. However, GoGoBus uses an open source service that can provide reliable and up to date schedules and locations.

The chosen service is a travel planner named Viaggia Bologna created by the Bruno Kessler Foundation in Trento. This service provides a list of different itineraries starting from fields like date, starting time, starting place, ending place, mean of transportation, etc. Some of the important information returned by the service are the starting and ending bus stops, the times of departure and arrival, the walking minutes, the bus line, the legs information, etc. This is all needed to provide a planning feature.

Real-Time ETAs

Tper[22] is the public agency of transportation in Bologna. Tper provides an online and a text message service called Hello Bus[23] to get the ETA of a bus line given the interested time and stop. There is also available a developer version of this service which can be used in order to show the bus ETA while waiting for it and estimations during the ride for the trip ending time.

This service tracks buses using their GPS position, collects the fleet data and by considering driving times and current traffic information it gives an estimation of their arrival at every stop. However, two problems originate from the use of this service. First, the estimation provided is not very accurate: the system tends to associate to a bus a delay larger than its actual one.

Second, in order to track their location, all the fleet buses must have an installed communication device that works in pairs with the GPS and regularly sends location data to the agency server. In Bologna all the buses have this device, but the system relies on the fact that the bus drivers have turned it on. Obviously, every bus with the communication device or the GPS disabled, cannot be tracked by the system, which is unable to give real-time updates for those vehicles. For these reasons, the real-time system is not always reliable and must always be integrated with the scheduled times obtained from the travel planner.

Bus Stops Location

Regarding the travel support, the key is to have all the intermediate stops and associated locations. Then by comparing them to the device GPS position, it is possible to tell at any time where the bus is in relation to its route. This is fundamental in order to know during the trip how many stops are left before getting off and the names of the previous and next stops. Integrated with metric distances between the user and the next stop and a system that recognizes when a stop has just passed, this is the foundation of the support that the application provides during the trip.

A service of SMAll allows to get all the stops of a line and their location, given starting time, date and stop, ending stop and a route id.

Design

Model

Together with my colleague, Francesco Fiacco, we designed the core model of GoGoBus. In order to deal with public transportation, the app needs to have classes that model entities like travel profiles, places, stops, lines, legs and routes. These are all interlaced between them and in the idea of a route, made of multiple legs, in which a line goes across a series of stops identified by places.

Specifically, the main information contained by these entities are the following:

- Profile: name, start place, end place;
- Place: name, location;
- Stop: name, code, site, location, bus departure time;
- Line: name, code;
- Leg: start stop, end stop, line, start time, end time, start date, end date, inter stops;
- Route: list of legs, start place, end place, start time, end time, start stop, end stop, total minutes of walking.

In Android the easiest way to store permanently data available across the app, is the use of the SharedPreferences. The preferences work like a map, with key strings and data. Unfortunately, they only allow saving primitive data types (boolean, int, double, etc.) or strings. As the entities described above are much more complex than a single primitive data type, in order to store and retrieve these objects efficiently in the SharedPreferences, we need the possibility to turn them into a String and re-create them from the same String. For this purpose, each of the Java classes related to those entities, have two methods:

- savingString(), returns a String with all the information contained in the object
- getXYZFromString(), statically returns a XYZ object filled with all of its information (XYZ stands here for Profile, Place, Stop, etc.)

The first category of methods simply concatenates the information separating them with a special character like one of these examples: ∫∂Δßŧμ√π. The character must be used only for a single entity and is the same used by that entity in the method of the second category to create a StringTokenizer and separate the original String, previously created by savingString(), into tokens containing the information defining the object, which re-created using its constructor. So for example at the end of these lines of code:

```
Profile p1 = new Profile(…);

String s = p1.savingString();

Profile p2 = Profile.getProfileFromString(s);
```

the profiles p1 and p2 contain the same information. The same works for all the other model entities.

User Interface

The most complicated part of the visual design at first is taking the role of a visually impaired person. The UI, in order to be usable without sight, must be designed properly, for example, an activity (page) with many options and fields to fill up is not easy to use for people with visual impairments. Instead, it is better having multiple and extremely simple activities with big buttons so that they are easier to understand, navigate and use. Moreover, the aesthetic choices are not as relevant as in a standard application, but what governs the UI is its usability and functionality for someone without sight.

Francesco Fiacco has focused his contribution on the special UI design. This is why I will not go over the principles here, but instead just give a general idea about them.

Google TalkBack, the built-in Android accessibility service, is a useful service for visually impaired. Here is how TalkBack explore by touch function works: a single tap triggers the spoken description of a component and half selects it, a double tap anywhere on the screen selects the last component explored, equivalent to a tap with TalkBack disabled.

In addition, a text to speech component should be used in order to provide messages and instructions.

Travel Profiles

In the design of the app, we recognized that it is important to have in the application a travel profile management component. This allows the creation, editing and deletion of travel profiles.

Travel profiles are the other focus of Francesco Fiacco, I will therefore go over only the key ideas. As said in the model description, a profile is a pair of places: an origin and a destination, both with latitude, longitude and a related address. The origin and destination can be entered in three different ways: through GPS location, through the use of previously set favorites or with the address. All the profiles and favorite places are stored in the app shared preferences, and accessed throughout the application itself.

Planning

The route planning is done thanks to the use of the online travel planning service described in sopra. The service is called on demand when the user wants to use a travel profile and it returns several information, useful also for other components of the app.

Everything that deals with planning and traveling is the main focus of my contribution to this thesis project.

There are two different approaches to plan a trip: a static way and a dynamic one. The difference is when the routes are computed. By computing them at the time of the profile creation, the user would have to ride a single chosen line and stop. The other approach, the one adopted, is more dynamic: it originates from the idea that buses do not have a regular schedule, but their times and availability vary a lot day by day and also through the same day depending on the current time. The routes are here determined only when needed and multiple choices are available to the user, typically with different lines, stops and times, depending on the availability.

This allows to catch a larger number of buses and get to the destination in a smarter way. Obviously though, the user may have a favorite stop or line, for this reason the app suggests multiple routes for the same trip, so that the user can choose the one that fits the best.

The app can also distinguish between different legs of a single route, up to three, treating them one by one as single sub-trips. Therefore, in case of routes with bus changes, the app provides support also in between two bus rides.

Going to the Stop and Waiting for the Bus

While going to the stop or waiting for the bus, the useful information that should be displayed is related to the bus arrival time and the future ride. This is why a specific activity gives an estimation of the bus arrival based on its GPS tracking system.

Moreover, while walking, the user may want to know the distance to the bus stop. So the app provides a walking distance in meters from the device to the bus stop calculated through the same travel planning service (see the service described in Travel Planner) and updated every few seconds.

The Bus ETA is provided by the regional transportation agency Tper through the online service described in Real-Time ETAs. By knowing the real-time ETA, the user is more aware of the situation, having an idea of the time left before being ready to get on. A very long ETA can suggest the user to look for a different route if possible, while a very short one suggests to be ready at the stop as the bus should be coming soon.

This feature is particularly important for frequent riders. It is common, especially in urban areas, to have different routes that can take to the destination, following either the same road or different ones. By having an estimation of the bus arrival time, the rider has more useful information to choose a route instead of another one.

The ETA should be retrieved regularly as it may vary a lot because of traffic.

For the reasons described in Real-Time ETAs for which the time given by the service is not always available and sometimes not accurate, even when this service is available, it is always integrated with the scheduled time.

On the Bus

The support provided while on the bus is definitely the most important one. Even for not visually impaired people, the situation is not the easiest to interact with the app, but critical and a potential source of confusion. Therefore, everything in this phase needs to be carefully designed.

On the bus it is fundamental to have an idea of the distance to the destination. In this direction the app computes the time remaining before getting off and also the number of stops left. It is also useful to know the stop that just passed and the next one, also to orient across the city, so the app has a sophisticated algorithm to detect when a stop just passed.

The list of the intermediate stops with the associated locations needs to be downloaded separately as it is not provided by the travel planning service.

During this phase, the user must be updated every time a stop passes with an alert recapping the situation. Moreover, the user must be notified with a vibration when the stop to get off at is the next one, so that visually impaired, but also new and infrequent riders, and frequent users who are distracted, can request the stop, get ready and not miss it.

Stop Passed Algorithm

In order to detect when a stop has just passed, the application has to keep track of the vehicle location across its route.

By knowing the ordered intermediate stops, with the associated latitudes and longitudes, it is possible to compute the air distances between the vehicle (device) and the next stop, and also between the vehicle and the stop after the next one.

A possible detecting criterion is simply based on the control of the current GPS device location in relation to the stop position. If the device appears to be within 20 meters from the stop and its accuracy is smaller than 20 meters, then the stop is considered passed and the app communicates it to the user, both with a text on the screen and, if enabled, vocally using the text to speech component. In order for this criterion to work, the device distance to the stop should be checked regularly.

However, the GPS location is not always precise, especially while moving on a vehicle. Moreover, buses generally skip a stop when nobody requested it and no one is there waiting. Therefore, if the bus is going fast (for example on a big road in the suburbs), the bus driver skips the stop and the GPS location is not precise, the 20 meters criterion may not be triggered and the stop would remain undetected.

For these reasons, we introduced another criterion, to fill the gaps left by this first one. The second one is more complicated: it detects when the bus is distancing from the stop just passed and approaching the one after. It involves multiple consecutive location updates, two stops and air and road distances to both of them.

Here is how the criterion works: by comparing the distance to the stop with the one calculated at the previous location update, it is easy to tell when the bus is distancing from the stop. However, this is definitely not enough to consider the stop passed: this can be caused by bad GPS reception, by the road shape, there may be a curve, a roundabout, or simply by the bus route itself that does not follow the shortest path to get to the stop, for various reasons like road sizes, one ways, buildings, etc. For these reasons it is important to check also the approaching to the next stop. So also the distance to the following stop is calculated and compared with the one previously computed to detect if the bus is approaching it. Even if distancing and approaching are detected at the same time, it cannot be enough yet to consider the stop passed: the distances are still air linear and not the actual road distances. Road distances can be retrieved from the travel planning online service and they should be computed only when needed, not every update as they require two server calls. The calls should be performed only to confirm the distancing and approaching hypothesized using the air distances. If the comparisons of the road distances confirm it, then the stop is passed and the user should be notified.

Crash Detector

Providing constant support requires that the application is always running in the foreground or background, especially from when the user chooses a route until the application is closed. This can require it to run for more than an hour nonstop. Across such a long time frame, while traveling, many aspects can go wrong: GPS, mobile data network, slow online services, active memory size etc.

For example, it is a fact that mobile networks do not always have a high degree of reliability. Either because of network coverage or simply a temporary weak connection, while moving everyone encounters problems with network connection, despite the age and quality of the device. These issues increase when moving on a vehicle as the reception gets worse, alongside with connection speed and reliability.

With all this in mind, the application should have a crash detector component. This should restore the situation whenever a crash occurs, so that the application can keep supporting the user during the trip. Whenever the app starts with a previous trip unfinished because of a crash, the component should recognize it and restore the last opened screen right before the crash. If the crash occurs while on the bus, at the reopening, the system should not restart from the beginning of the trip, but from the point reached earlier.

Testing

When testing the application, it is crucial considering also borderline cases and not only the best case scenario. To avoid spending too much time on buses to test the travel support in critical situations, the use of an external application to simulate the trip is fundamental. It saves time and money and also allows to go through potentially complicated situations, like sharp turns, roundabouts, etc. that can confuse the Stop Passed algorithm.

A utility to help testing is Lockito[24]. Lockito is an extremely useful tool to make the device following a fake itinerary, while controlling speed and GPS accuracy. To use it, mock locations must be allowed from the developer options in the device settings. Once the itinerary is set, also with intermediate spots, cruising speed, accuracy and accuracy offset, the fake trip can be started.

By reproducing with Lockito intricate bus routes like those traveling across the center of Bologna, it is possible to test and stress the Stop Passed algorithm in every aspect. With different accuracies and offsets it is also possible to test singularly either the 20 meters criterion of the algorithm or only the distancing-approaching one, or even a mix of the two as in a real setting.

Obviously tests with real buses must be performed anyways in order to ensure that everything works properly also in a real environment, checking the actual precision and usability of the system.

Technical Description

App Organization

The application is organized in packages: model, activities, adapters, utilities and json. The package scheme is represented in Figure 2.

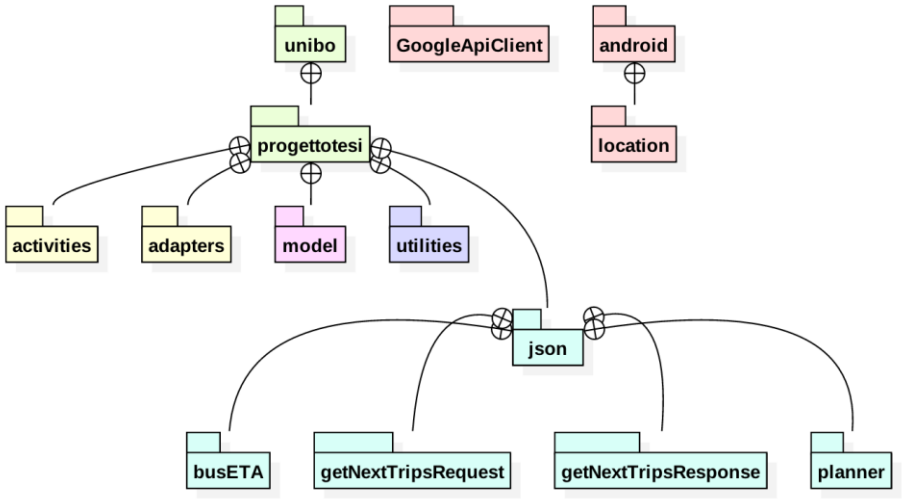

Figure 2. Packages: the application organization.

With reference to Figure 2, the packages with a yellow color contain Java files related to the graphic aspects, the activities and the adapters. These components fill views and lists, making buttons reacting when clicked and retrieving the information about the model from the preferences.

The packages colored in light blue are related to Java files auto-generated to handle the server calls replies. They are compatible with the JSON format and they contain the classes that handle the data downloaded from the server. Specifically, the two getNextTrip are related to the request and the response for the list of stops, which are used for real-time tracking while on the bus. Finally, planner contains the Java classes that handle the data downloaded from the travel planner, similarly the package busETA does for the real-time information service.

The packages colored in red are Android built-in packages: location is used every time the GPS location is involved, while the GoogleApiClient is used for the activity recognition, to detect if the user is on a vehicle (the bus) or walking, in order to show the respective information.

The green packages are just containers. Utilities contains several Java classes: there are those that take care of the actual communication with the servers, there are also other classes to manage the GPS location and services for the activity recognition.

The core of the data managed by the application is contained in the model package. Inside it there are all the classes representing the relevant entities involved in public transportations. Figure 3 is a representation of the relations between the model entities.

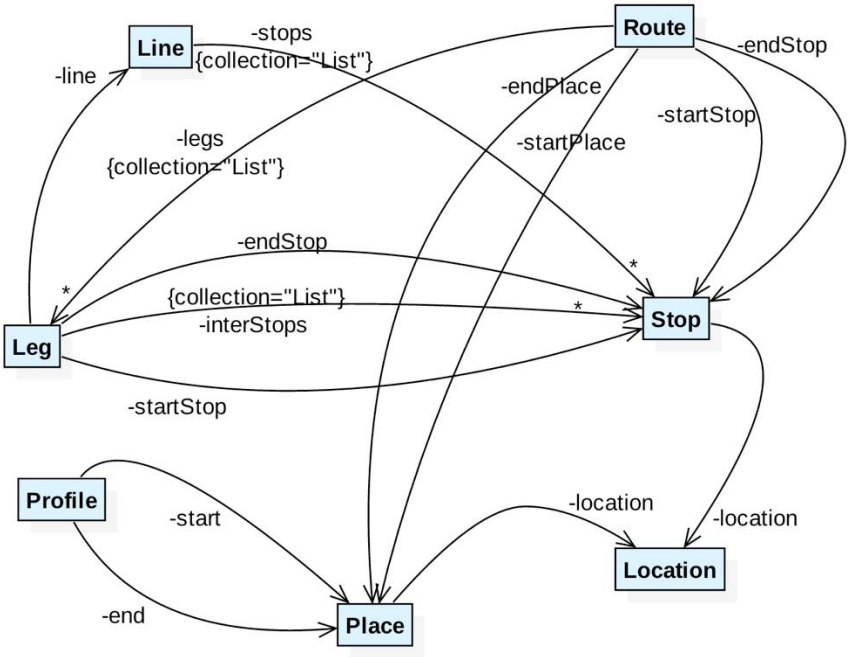

Figure 3. Model Relations: the dependencies across the model package.

From the diagram, it is clear the amount of relevant dependencies between the entities. The model UML diagram in Figure 4 is more detailed and shows also the main methods of the various classes.

The UML diagram of Figure 4 does not show the get and set methods related to all the classes private fields. It does not show all the constructors available either, but only the main and most used ones. Also, in order to simplify the view, classes like Time (from the package utilities) are omitted here.

Each class contains methods to save to and retrieve from the SharedPreferences. As every class has these methods, it is very intuitive storing and retrieving also complicated objects like a Route.

Figure 4. Model: UML diagram.

Activities Implementation

This section is dedicated to the description of the relevant components of the application. The following descriptions address and explain the main choices and issues encountered while developing the application, both regarding interface and technical aspects.

Travel profiles and the associated UI are the focus of my colleague Francesco Fiacco, I will therefore give only a general idea about it.

On the other hand, planning and traveling are the main focus of my work of this project. The idea here is showing and motivating the main implementation choices. The focus will be mostly high level, with the key components also detailed with code excerpts.

All the layouts presented below have buttons and components designed for visually impaired people. Specifically, they are suitable for the use of Google TalkBack. Moreover, a vocal message is associated to each button: the message is read aloud on click by the TextToSpeech component, but only if the user selected this option.

The description that follows is split into the different activities in order to present their layouts and what happens behind them. An Activity in Android is an application component that provides a screen and is associated with a window with which users can interact.

Home

This is the activity that opens when the application is launched from the app drawer or the home screen.

Interface Description

Characterized by a simple and easy to use interface, the Home activity (see Figure 5) has three big buttons for profile management, settings and starting a new trip. The first time ever that the app is launched, it asks if the user wants to activate the text to speech support, this option can be later modified in the settings.

Technical Description

Except for the basic methods to handle the creation of the view and button clicks, this activity (called MainActivity in the code) presents fundamental instructions for components like the crash detector and the activity recognition (walk, in a vehicle, etc.). As the MainActivity is never terminated unless the user closes it, it is the ideal place for the activity recognition module. The detection of walking and motion in a vehicle, in order to save battery power, are used only when needed, i.e., while at the bus stop and also on the bus, only when a few stops are left. The component is started from the respective activity and managed inside MainActivity. For the crash detector, since this is always the first activity that is created whenever the app is used, before creating its view, it checks if the app previously terminated without the user's consent and in case it launches directly the last activity that was open before the crash occurred.

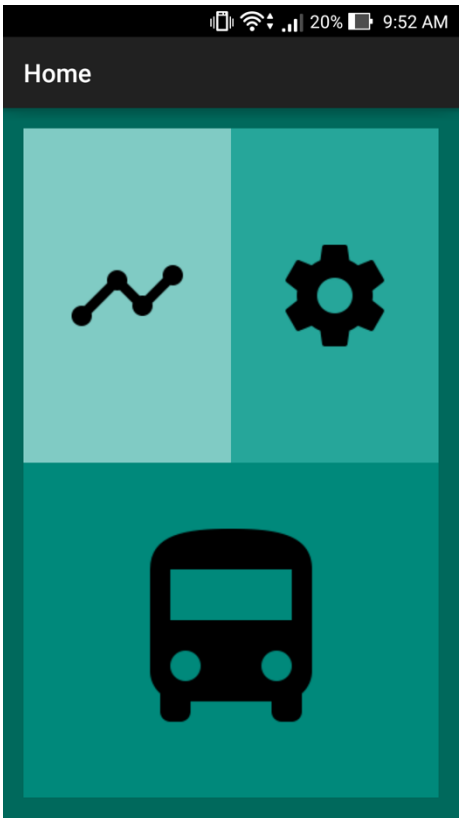

Figure 5. Home Page: screenshot.

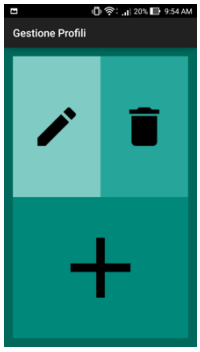

Figure 6. Profile Management: screenshot.

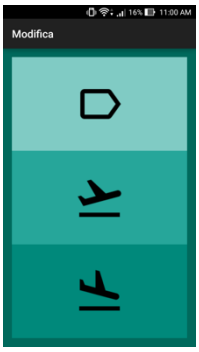

Figure 8. Profile Editing: screenshot.

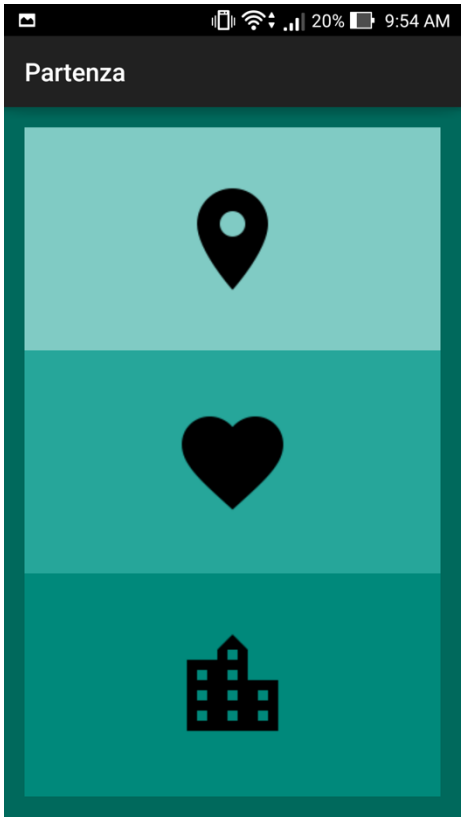

Figure 7. Travel Profile Creation: screenshot.

Travel Profiles

Travel Profile Management

This activity, called ProfileManagingActivity, opens when the user clicks the top left button in Home and it redirects to other activities for editing, deleting and creating travel profiles.

Interface Description

The travel profile management activity (see Figure 6) has the same layout as Home, it has buttons for profile editing, deletion and creation. Creation and editing will be detailed below. Regarding deletion, if the bin button is clicked, a list of the saved travel profiles is shown (EditDeleteActivityB) and the clicked one from the list is eliminated after requesting a confirmation.

Travel Profile Creation

This is shown at the click of the plus button in the previous screen. It is reused twice in the creation of a new profile, both for the selection of the profile origin and its destination. The origin selection is shown in Figure 7.

Interface Description

Here there are three buttons, from top to bottom, they allow to select origin or destination through the use of GPS, favorite places previously set, or with an address input (see Figure 7). If the user selects GPS, a confirmation dialog of the nearest address found pops up, so that the user is sure that the GPS located the device accurately. If the user selects GPS or address input, a dialog pops up asking to save the new place as a favorite, if yes, then another activity is shown to insert the name. If the favorites are chosen, then another activity with a simple list to choose from is shown. Whenever the user confirms the GPS, choses a place from the favorites list or confirms an address, a new activity like this appears to choose the destination. The destination view is the same (may not show the GPS button if chosen as origin), but the confirmation of the destination place shows another activity to input the travel profile name, which will then be saved.

Technical Description

This activity, called NewProfileActivity, is the key for the profile creation flow, shown in the diagram in Figure 9. It guides the user through the steps for the creation of a new travel profile. This and the other activities related to the profile creation deal with GPS location, geocoding and reverse geocoding, alert dialogs and SharedPreferences.

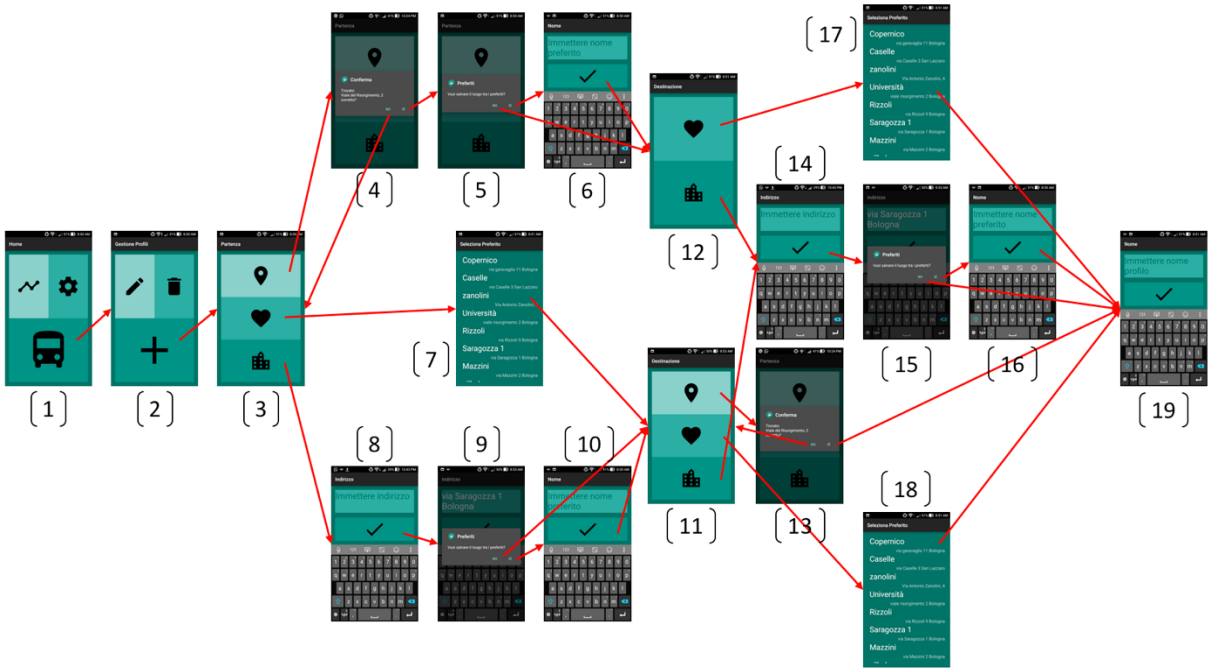

Figure 9. Profile Creation: diagram showing all the possible steps and activities involved.

Starting from Home (1), Figure 9 shows a diagram of the flow of the profile creation. Passing from profile management (2) and the selection of the origin (3), the creation can continue selecting a favorite (7) or choosing the GPS: in case, after asking confirmation of the current location (4), it asks if the user wants to save it as a new favorite (5), in which case a new screen is shown to input the name (6). If address is chosen, then an input screen is shown (8), the user is asked to save it as a new favorite (9, 10). Then the destination selector with all three input methods is shown (11), or one without the GPS option (12) if it was previously chosen as origin. The

destination part is the same as the origin (13, 14, 15, 16, 17, 18), with screenshots numbered (17) and (18) representing the same choice. At the end, the user is asked to input the new profile name (19).

Travel Profile Editing

This activity, EditActivityB in the code, is shown after choosing (from the home view) profile management, editing and having selected a profile to edit. The activity that opens when the pencil button in profile management is clicked and leads to this one is called EditDeleteActivityB in the code and it is the same used to display the list of profiles both to edit and delete.

Interface Description

With the same layout as the Travel Profile Creation one, this activity allows the user to rename a travel profile, modify its origin or its destination (see Figure 8). By clicking the name button (with the label icon), a text input activity is shown and, if confirmed, the new name of the profile is modified. On the other hand, by clicking origin or destination (taking off and landing planes), the same activity as the profile creation is shown (with a different title), so that the user can choose the new input method.

Figure 10. New Trip: screenshot.

Figure 11. Route Selection: screenshot.

Planning and Traveling

The following components are the main focus of my work in this project. Here, I go over the key features and ideas that governed the development, with an eye on the most critical aspects, the problems encountered and the related solutions provided.

New Trip

The new trip activity is displayed when the big button with the bus icon in Home is clicked.

Interface Description

Composed by a list and three buttons, it gives all the options needed to start a new trip (see Figure 10). The list is filled with the travel profiles, displaying the profile name, origin and destination addresses, smaller and aligned to the right.

When possible, the travel profiles are ordered by the current distance of the origin place from the user. This is done in order to facilitate the user's research, so that the travel profile needed will always be at the top of the list or close to the top, without having to scroll through all of them to find it.

The buttons at the bottom of the screen allow to start a one-time trip, look for trips starting in the future, or arriving at a specified time. The click of the left darker colored button opens a new activity to choose the destination. The two buttons on the right are used for time options. The darker one of the two triggers between "leave at" and "arrive at", while the lighter colored one displays a dialog to choose a time to depart or arrive. By default, they are set to leave at the current time.

When a travel profile is selected, a new screen appears with several different possible itineraries to choose from.

Technical Description

Called NewTripActivityB in the code, when this activity is created it loads all the previously saved profiles from the SharedPreferences. At the same time, it computes the distance between the device last known location and the origin of the travel profile, then it saves it inside the associated profile. The Profile class implements the interface Comparable for sorting purposes, so that the method compareTo(Object otherObj) simply compares the different distances. The profiles are shown thanks to the use of an ArrayAdapter called ProfilesAdapter, from the adapters package. ProfilesAdapter, used also to display the profiles also to pick one to be edited or deleted, is what creates the view of every item related to a profile and fills the list with them. Moreover, ProfilesAdapter has an instance of the activity that created it, obtained through the constructor. This instance allows the adapter to have a valid application Context, this is used in the adapter both to access the SharedPreferences and create a TextToSpeech component. It is also used as a parameter for a static method of NewTripActivityB, called when a profile is selected. The static method saves in the preferences the chosen profile for later access and starts a new activity passing the information related to the arrival or departure time selected by the user. Time that the user can select clicking the time button, which shows a TimePicker, an Android stylish built-in component.

Google TalkBack is the accessibility service that Android provides for visually impaired users. Already described in previous sections (Android and User Interface), I want to describe here its behavior when a list is displayed. TalkBack tells the user that there is a list and the number of items currently displayed out of the total in the list. Then every time the user touches an item of the list, TalkBack reads the content of it. In case of long lists, this can be very long and also potentially frustrating. For this reason, we decided to order the profiles by distance of the origin when possible, so that the most likely ones to be used are at the top, easier and faster to find and select. A further improvement is the integration with Nicola Ferroni's thesis project, which provides the most likely next trip (see the section Possible Improvements).

In Android it is often complicated, problematic and critic dealing with the transferring of information between different activities and between classes in general. Same for figuring out how to do actions that are not permitted in the "environment" where they are needed. For example, many actions like displaying a Toast message, accessing the preferences, starting a new activity, etc. all require the application context. However, this context cannot be retrieved from a class not derived from Context itself. In this application, these issues were solved for ProfilesAdapter (and others) by having an activity instance, passed from the constructor. Another approach is to use handlers: a Handler is an object that receives a message and runs code to handle it. Every Handler is associated with a single thread. An unconventional and not thread-related way of using them is having some code executed by an activity without having a reference to it. For example, in order to save power, handlers are used in this project to start and stop the component that detects if the user is on a vehicle or walking (MainActivity).

Route Selection

The Route Selection screen is shown when the user has selected a travel profile to start a new trip. The selection of one of these routes opens the next activity.

Interface Description

This activity is dominated by a list of routes (see Figure 11). As the server call can take a little time, a progress bar is displayed before the routes are available. The routes are listed with departure and arrival times, total travel time, the starting bus stop, the bus lines to take, the walking time included and the arrival stop. Further information about a route is shown in the next activity.

Technical description

Called SelectRouteActivityB in the code, it uses the class RouteFinder (from the utilities package) to retrieve the routes, RoutesAdapter and a class named Filler to fill the list.

RouteFinder performs the actual server call, all it needs are origin, destination and departure or arrival time. The call is made with the use of Retrofit[25] an open source library for Android and Java to handle synchronous or asynchronous HTTP request to a remote webserver. The main components needed by Retrofit are the url, an interface with the declaration of the parameters, a converter and a Call with a related onResponse method to handle the response. The response needs to use a class, or series of classes in this case, that has specific annotations. This particular kind of class can be generated with a tool starting from the server response. The response is written in the standardized JSON format. We used jsonschema2pojo, a very useful online open

source tool to convert from JSON to the plain Java classes[26]. All the generated classes are inside the package planner inside JSON.

Once the request is successful, the routes need to be cleaned from all the data not needed. To accomplish this, all the routes, stops, legs, transport info, etc. are saved into the respective Route, Stop, Leg, Line, etc. objects from the model package. Specifically, Route has a static method called getRouteFromPlanner(json.planner.Response response) which converts the route retrieved from the server call to a Route object. To do so it goes over the main fields and saves them reconstructing a full route with legs, stops, places, times, etc. Unfortunately, at the moment the planning service used is not able to distinguish between a bus 19 and a 19C, which is shown as 19 and this can be source of confusion.

Filler, from the utilities package, simply fills the view for each item given the information and attaches them a click listener to start the next activity. RoutesAdapter, from the adapters package, does the equivalent job of ProfilesAdapter in the previous activity.

Figure 12. Bus Waiting: screenshot.

Figure 13. On The Go: screenshot.

Bus Waiting

The Bus Waiting activity is the one showing when the user has selected a route, typically before walking to the bus stop. It serves as guide for the user to take the bus and give live information about its arrival.

Interface Description

The view shows plenty of information related to the bus route chosen by the user (see Figure 12). At the top we have the general information about the whole route, with the same layout as seen in the Route Selector, this creates continuity throughout the trip and gives always an immediate overview of the route. Then we have a scrollable view with the current leg information, starting with the walking distance to the bus stop updated in real-time, continuing with origin stop, scheduled bus time, bus ETA and delay determined with satellite tracking, bus line, destination stop and scheduled arrival time. This information is repeated for another leg in case of multiple ones. The application supports up to three legs, so three different bus lines in the same route. The real-time walking distance can be useful if either the user or the bus is late, so that the user can tune the walking speed accordingly. At the bottom of the screen there is a button with a bus icon, it should be pressed when getting on. In case the user forgets to press it, the app detects it and notifies.

Technical Description

Called BusWaitingActivity in the code, this activity uses the components for the ETA requests and the current walking distance. It starts setting up a location updates request, then it loads the chosen profile from the SharedPreferences, after filling the information, it asks for the bus ETA update. This activity relies on RealTimeTracker for the real-time walking distance and the bus real-time ETA. By implementing the two interfaces HelloBus and Walking, it can easily share with other activities the use of the same methods of RealTimeTracker, from the utilities package. RealTimeTracker has static methods to make server calls to different services depending on the information requested. The bus estimated time of arrival is requested every minute, while the walking distance every two seconds. The server calls are performed with the use of Retrofit as seen above. Specifically, the bus ETA is not always available as it relies on the bus driver turning on a system present on the bus. If not available, only the scheduled time is shown. Moreover, as the service requires only the bus route and the stop code, it is possible that the ETA shown does not match the scheduled time and needs to be discarded. Regarding the walking distance, its duration, which is also shown when selecting a route and in the top part of this screen, happens to be overestimated: distances which require 5-6 minutes of walking at a normal pace, are computed as 10 minutes, lacking in accuracy. This is why the first leg information is a distance and not a walking time. Due to this overestimation, when the user requests a new trip, the bus routes that leave in the next few minutes are not shown as the planning system considers a longer walking time than the actual one. For this reason, it is more convenient inserting the addresses of the stops instead of the real origin and destination places.

In order to detect when the user gets on the bus without pressing the Get On button and prevent the loss of support during the trip, as soon as the user opens this screen, an activity recognition component is started. This component, made by Google and part of the APIs, at a given interval can detect the probable current activities and associate a confidence to each of them. By enabling this service and using an IntentService, this information can be read. The IntentService in this application is in the utilities package, called ActivityRecognitionService. It is used to detect that the user is on a vehicle, in case the rider is notified and asked for confirmation. If confirmed by pressing the dedicated notification button, the next activity is shown. It is important to have the

next activity opened within the first two stops of the ride as in the current version, after two stops are passed there is no way to provide the tracking support during the trip.

On The Go

The On The Go activity is the core of the application, where the support comes useful for someone who is visually impaired. In this activity, displayed when the user gets on the bus, we have the most innovative aspect of the whole application: traveling support with real-time tracking.

Interface Description

The view shown in Figure 13 contains all the information needed while on the bus. At the top of the screen there is, as before, an overview of the current trip. Then there is a scrollable view showing the stops information, while at the bottom of the screen there is a get off button. The central view shows in order the current bus line, the previous stop name (with a progress bar dismissed as soon as the stops are available), the distance to the next stop, the next stop name, the number of remaining stops, the scheduled and estimated times before getting off, and the scheduled and estimated arrival times at the final destination. In particular, the distance updates real-time, as well as the GPS estimated times, shown only if available.

While the bus is moving, the application is aware of its location, thanks to the device GPS. This allows it to monitor its position in comparison to the bus stops and the whole route. This information is used to give the user an idea of the time remaining, indicate the next bus stop, show the number of stops before getting off and display an arrival time estimation.

Technical Description

This activity, called OnTheGoActivity in the code, is the most complicated of the app, it includes the algorithm to detect the overcoming of a stop, described in the next sub-section.

It starts requesting location updates to be sent every second; then it retrieves the current route from the SharedPreferences; fills up the top view using a static method of Filler; fills also the information of the scrolling view; downloads the inter stops and sets a timer to request ETA updates every 30 seconds.

All the server calls are done using RealTimeTracker (utilities package) which uses Retrofit to perform the actual calls. The list of the bus stops along the route is downloaded by the method getStopsFromWeb of RealTimeTracker. In this case, the server call is an HTTP POST and not a GET as in the other cases. To get a response from a POST request, an object in a JSON format needs to be sent to the server. Retrofit can handle also this kind of calls, that require the generation of more data container classes for the communication with the server. The information needed by the server to retrieve the stops are the current date, the bus line direction and name, the starting stop code, the number of trips to display, the ending stop code and the starting time. On the other hand, the response contains information like a unique trip ID that can be compared to the one downloaded for the planning, to be sure that the stops will be correct. It also contains all the bus line stops from its origin to its terminus. Every stop comes with its location, name, code and the time at which the requested bus is supposed to pass. These stops are generally too many for the trip requested, so they need to be processed to filter only those that match the request. This is done by the method stopsToInterStopsConverter(List<json.getNextTripsResponse.Stop>

stopsR) of RealTimeTracker, which notifies the activity of the successful download with a method called setNewLeg(Leg currentLeg). The new leg contains all the stops that will be processed throughout the trip.

OnTheGoActivity deals with both air and road distances. Air distances are used both to display the meters to reach the next stop and to consider the next stop passed. They are calculated using a mathematic function from LocationToolbox (also inside the utilities package). The static function distance of LocationToolbox computes the distance in meters between two given points using their latitude and longitude alongside with trigonometric functions. Instead, road distances are used only in the stop passed algorithm to confirm that the bus is distancing from what is considered the next stop and is getting closer to the stop after. Linear air distances are mainly used because road distances require to perform a specific server call, while air distances simply need some Math calculations.

Every time the activity receives a location update (around every second), it calls locationUpdated. After checking that the next stop is not the last one, this method controls also if the stop passing condition is valid (passingCondition method). PassingCondition checks the overcoming of a stop, in case it calls stopPassed, then it updates all the views, showing the new distances and times. While every minute it asks RealTimeTracker to get the new ETA, in order to show the updated time to get off at and also to calculate a time estimation to get to the final destination. This estimation is done thanks to the scheduled time information contained in the server response to get the stops.

If the remaining stops are less than 3, then the activity recognition component is started, so that it will be able to detect when the user will get off and start walking. If the system detects that the rider got off at a wrong stop, the activity recognition information could be further used to ask the user for a route recalculation.

If the stop to get off at is the next one, then the user is notified with a vibration and a message, the vibration will persist until the user confirms a pop up alert.

All this support is available also while the app is running in the background or the device is locked. However, while testing the application during the development, we hardly encountered a crash. In those rare cases it was related to memory usage issues. In Android every application has dedicated a discrete amount of RAM, if it goes over a certain threshold, then it crashes. As this Exception cannot be caught in any way, we decided to implement a crash detector component. After the user clicks “ok” in the crash alert dialog, this component restarts the activity that was open at the time of crashing (typically this one), setting as previous stop the last one that passed before the crash (always saved in the SharedPreferences alongside with other information).

This activity and its passing stop algorithm have been tested with the use of Lockito, set at different speeds, with different accuracies across common and critical routes. It performed very well as the stop were all recognized as passed even on critical routes with many turns and stops.

Destination

This screen is shown once the user gets off the bus of the last leg. The idea is displaying a final screen while getting to the destination.

Figure 14. Destination: screenshot.

Interface Description

This activity shows the name of the final stop, the distance in meters to get to the actual destination and the destination address (see Figure 14). The distance represents the walking distance and it is updated real-time every two seconds. On the bottom of the screen there is a button to close the activity and terminate the application itself.

Technical Description

The activity DestinationActivityB is simple. All the information shown are retrieved from the SharedPreferences, while the dynamic walking distance in meters is calculated with a server call made by the utilities class RealTimeTracker. As seen in other activities like Bus Waiting, the location updates are provided by the device GPS system.

Stop Passed Algorithm

The algorithm to detect the overcoming of a stop combines two different criteria. The first one will be called “20 meters”, the second one will be called “distancing - approaching”. Here presented separately, they are actually combined.

20 Meters

High Level Description

Designed for a city environment, with many stops and slow speed due to traffic and limitations, “20 meters” is simple and straightforward. It considers the device location, its GPS accuracy, and the next bus stop location. When the device is within 20 meters from the bus stop, with an accuracy smaller than 20 meters, the stop is considered passed. The distance is calculated with Math functions and the internet connection is not used.

Visual Description

The following drawings will show how it works and point out its weaknesses.

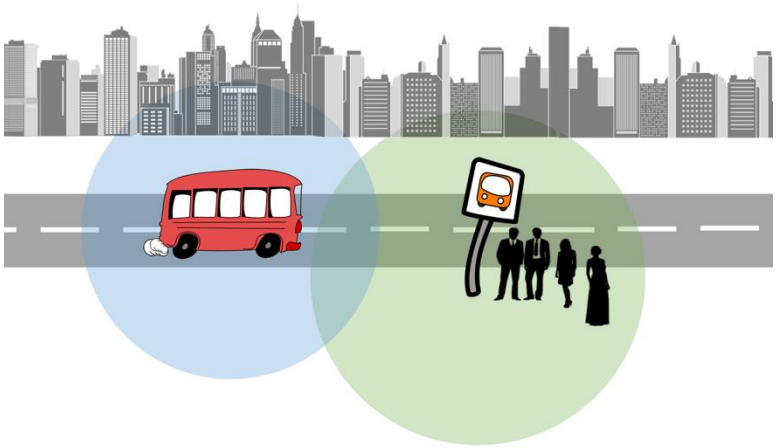

Figure 15. 20 Meters: diagram showing a blue accuracy circle.

Figure 16. 20 Meters: diagram showing the bus approaching the stop.

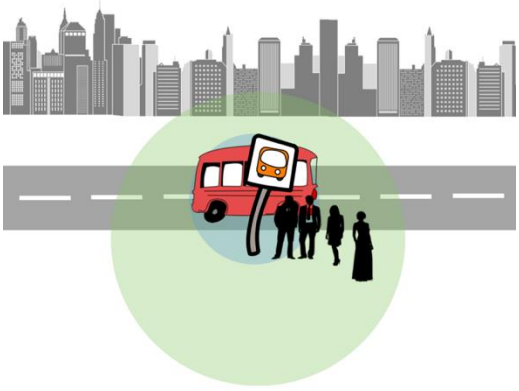

Figure 17. 20 Meters: diagram showing the bus at the stop.

The blue circle in Figures 15, 16 and 17 represents the current device accuracy related to the GPS signal, while the green circle represents the 20 meters radius activation circle of the bus stop.

Whenever the device (bus) is inside the 20 meters circle, then the stop is considered passed.

So in the situation represented in Figure 16 it is considered passed (accuracy is around 14 meters), even if the bus has not reached the bus stop yet.

Same in the situation in Figure 17. The bus has stopped, during the time of the stop the device can improve its accuracy (here around 10 meters) and has enough time to receive many location updates. Typically, at least one of these updates will meet the accuracy requirement and the stop will be considered passed.

Figure 18. 20 Meters: diagram showing a high accuracy that prevent the stop to be detected as passed.

However, the stop is not considered passed in the situation represented in Figure 18, where even though the bus is within the green circle, the GPS accuracy is too high (around 30 meters).

The problem of this algorithm is that it relies on the GPS precision, which can be unreliable while moving on a vehicle.

Technical Description

Here is the code to detect when a stop is passed with this criterion. The first method, locationUpdated(Location location) is called every time there is a location update from the GPS. The method passingCondition(Location location) checks if the distance d0 from the device to the stop is less than 20 meters, then in case the method calls isClose(int acc), which will then call stopPassed() only if the accuracy is smaller than 20 meters. The method stopPassed() sets as previousStop the one that just passed and removes the same one from the list stopsToGo, those left before getting off. The method updateViews(Location location), here not shown, simply updates the information displayed on the device screen.

```
private Stop previousStop;
private List<Stop> stopsToGo;

private void locationUpdated(Location location) {
  if (stopsToGo != null && stopsToGo.size() > 1) {
    passingCondition(location);
    updateViews(location);
  } else{
    if (stopsToGo != null && stopsToGo.size() == 1) {
      //notify the user to get off
    }
  }
}

private void passingCondition(Location location) {
  int d0 = ((int) LocationToolbox.distance(location.getLatitude(),
       stopsToGo.get(0).getLocation().getLatitude(), location.getLongitude(),
       stopsToGo.get(0).getLocation().getLongitude(), 0.0, 0.0));

  if (d0 < 20) {
    isClose((int) location.getAccuracy());
  }
}

private void isClose(int acc) {
  if (acc < 20) {
    stopPassed();
  }
}

private void stopPassed() {
  previousStop = stopsToGo.get(0);
  stopsToGo.remove(0);
}
```

Strengths and Weaknesses

The strength of this criterion is that it does not need an internet connection to work. This makes it usable also in conditions without internet signal. In fact, using the mobile data network can be source of problems because its speed, difficulties increase in an old device and while moving in a vehicle. This criterion is robust to this kind of internet issues.

However, this heavily relies on the GPS signal precision. The GPS position can be unreliable, especially in the following situations: moving inside a vehicle, staying between buildings, using an old smartphone, etc. For these reasons, it is possible that the precision requirement for this criterion is never met. Especially if the bus does not stop, there is a chance that this criterion cannot recognize the overcoming of the stop. Therefore, it is clear that an additional criterion is needed.

Distancing - Approaching

High Level Description

Designed for more suburban and border line situations, this criterion is more robust and can detect that a stop is passed even with a weak GPS reception. This criterion involves multiple sequential location updates. More complicated than the “20 meters” one, “distancing approaching” considers the location of the next bus stop and also the one after, as well as the device GPS location. It deals with two different kinds of distances: the air distances to the two bus stop and also the respective road distances. Air distances are computed with Math functions (as for “20 meters”), while road distances are calculated with a server call. Moreover, to be able to tell when the device is distancing or approaching the stops, it needs to keep track of the previous distances.

The algorithm works as follow. Whenever the bus has distanced the first stop (S0) and approached the next one (S1) for two updates in a row (second and third, compared to first and second), then the respective road distances are computed (related to the third and fourth updates). If the road distances confirm that the bus is actually distancing the first stop (S0) and approaching the second one (S1), then the stop (S0) is considered passed.

Four sequential location updates are considered and the server calls for the road distances are performed only when needed, in order to confirm the supposed distancing and approaching from the air distances. Computing the road distances every update would give a faster detection, but it would also be an overhead (as the calls are also two every time). In the current implementation, detecting that the stop is passed takes around 3 seconds.

Visual Description

The following drawings show and describe how the algorithm works. In the situation represented below, the bus did not stop since there was nobody at the stop and it was not requested. The bus is going at a speed of almost 55 km/h, so two consecutive location updates have a distance of around 15 meters. The stops are 3 km far from each other. In addition, in order to simplify the comprehension, the GPS reported location are accurate.

At time 0 (Figure 19), the bus passes by the stop. See Table 1 for the distances in meters.

	Air 0	Air 1	Road 0	Road 1
Time 0	0	1000	/	/

Table 1. Time 0: distances for Stop Passed algorithm.

At time 1 (Figure 20), the bus is distancing the stop, while it is getting closer to the next one. The associated distances in meters are shown in Table 2.

	Air 0	Air 1	Road 0	Road 1
Time 0	0	1000	/	/
Time 1	15	985	/	/

Table 2. Time 1: distances for Stop Passed algorithm.

Only the air distances are computed. By comparing the new air distances to the stops with the ones computed at the previous update, the application detects distancing and approaching for the first time.

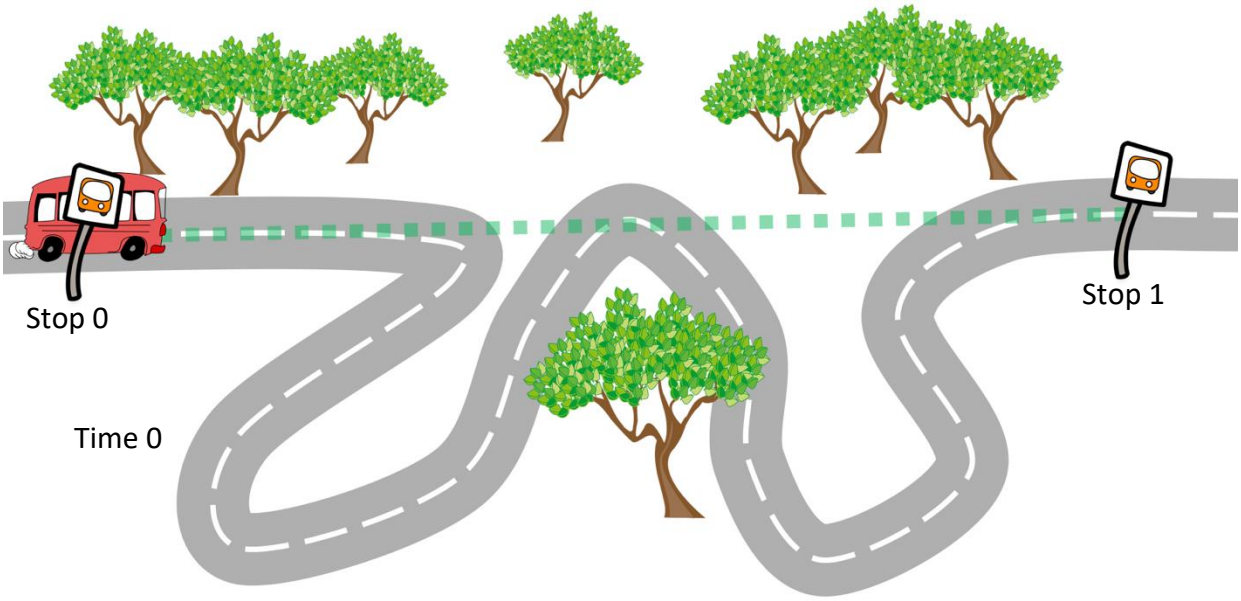

Figure 19. Distancing Approaching, Time 0: diagram showing the bus passing by the stop. The air distance to stop 1 is shown.

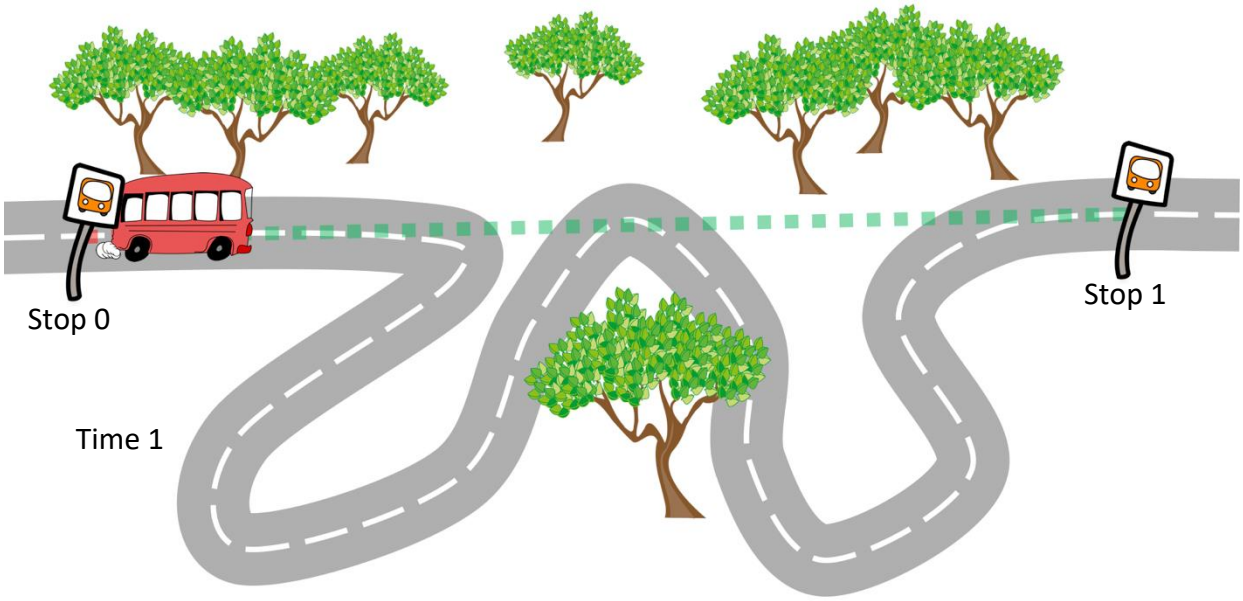

Figure 20. Distancing Approaching, Time 1: diagram showing the bus distancing the stop 0. The air distances to stops 0 and 1 are shown.

At time 2 (Figure 21), the bus is continuing on its route. The related distances in meters are shown in Table 3.

	Air 0	Air 1	Road 0	Road 1
Time 1	15	985	/	/
Time 2	30	970	?	?

Table 3. Time 2: distances for Stop Passed algorithm.

Here the app detects distancing and approaching for a second time in a row. In order to confirm this hypothesis, road distances are computed with two server calls. The results obtained are shown in Table 4.

	Air 0	Air 1	Road 0	Road 1
Time 1	15	985	/	/
Time 2	30	970	30	2970

Table 4. Time 2 with road distances: distances for Stop Passed algorithm.

The road distances are computed for the first time (Figure 22), at the next location update the algorithm will be able to confirm that the stop is passed.

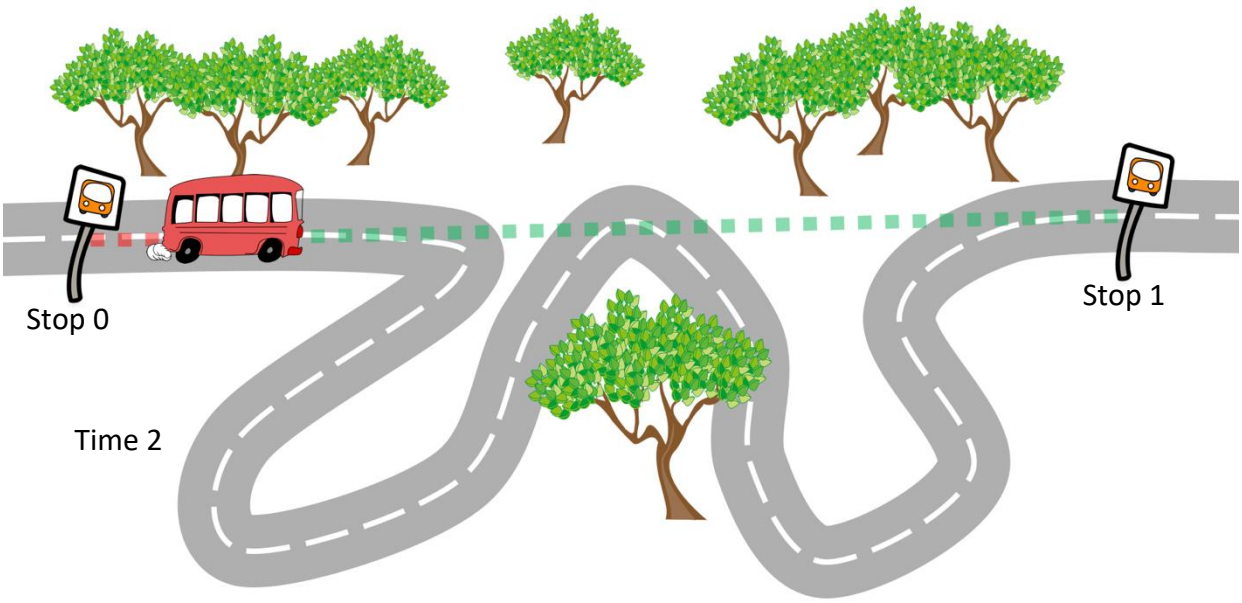

Figure 21. Distancing Approaching, Time 2: diagram showing the bus at another location update distancing the stop 0. The air distances to stops 0 and 1 are shown.

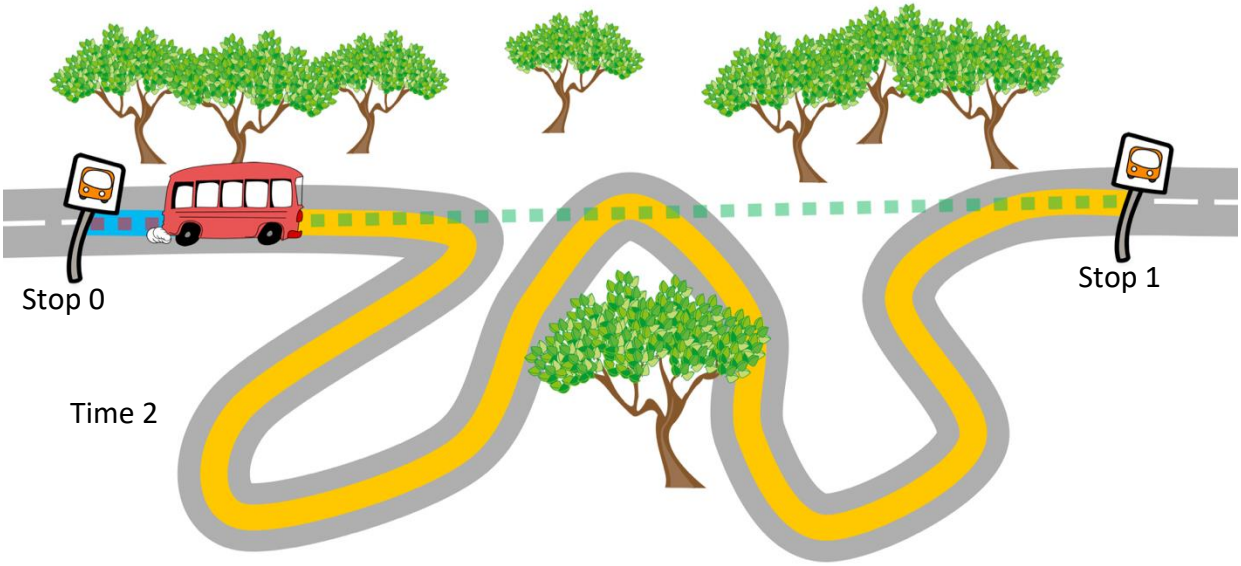

Figure 22. Distancing Approaching, Time 2 with Road Distances: diagram showing the bus distancing the stop 0 with road distances to stops 0 and 1. Also the air distances to stops 0 and 1 are shown.

At time 3 (Figure 23), the bus keeps going and the new distances are shown in Table 5.

	Air 0	Air 1	Road 0	Road 1
Time 2	30	970	30	2970
Time 3	45	955	?	?

Table 5. Time 3: distances for Stop Passed algorithm.

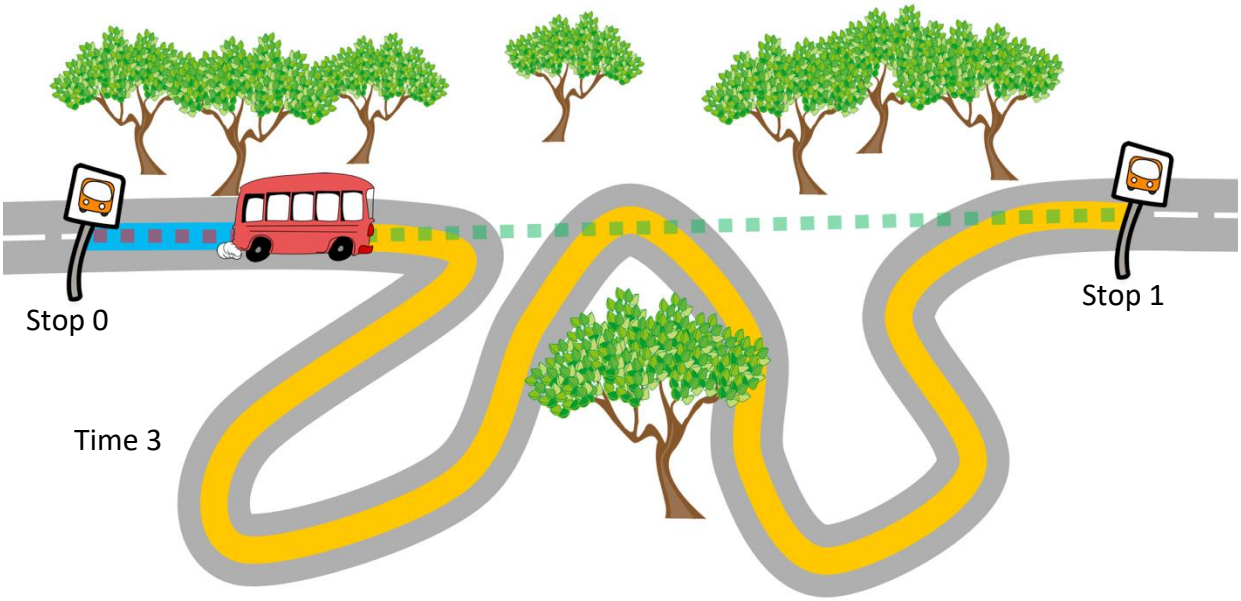

Figure 23. Distancing Approaching, Time 3 with Road Distances: diagram showing the bus distancing the stop 0 with road distances to stops 0 and 1. Also the air distances to stops 0 and 1 are shown.

As the air distances keep confirming that the bus is distancing the stop and getting closer to the next one, then the road distances are computed a second time (see Table 6).

	Air 0	Air 1	Road 0	Road 1
Time 2	30	970	30	2970
Time 3	45	955	45	2955

Table 6. Time 3 with road distances: distances for Stop Passed algorithm.

Finally, as the comparison between the newly calculated road distances and the ones computed at time 2 confirm that the bus is distancing from the stop and getting closer to the next one, the stop is considered passed.

The case shown above presents road distances that are coherent with the air ones. However, in cases that are more border line, the algorithm works anyway.

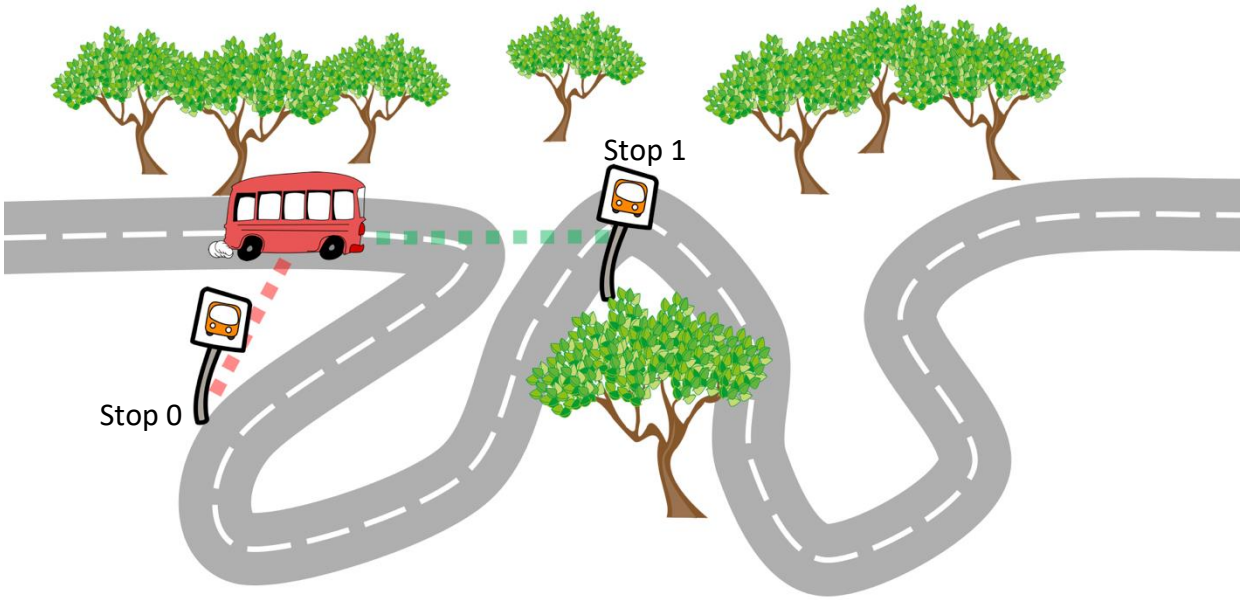

Figure 24. Distancing Approaching, Critical Case: diagram showing why road distances need to be computed. The air distances to stops 0 and 1 are shown.

In the situation represented in Figure 24 for example, the only analysis of the air distances would lead to a mistake. While the bus approaches the turn, the red distance is increasing while the green one is decreasing. However, the comparison between the road distances computed at the second and third consecutive updates with the air distances following this trend, will prove that the bus is getting closer to both stops. So that the following stop can not be considered passed.

Technical Description

This criterion requires a higher number of controls and variables. It involves a maximum of four different consecutive location updates and only if the 3 last ones are consistent with the distancing approaching trend, then the stop is considered passed. Specifically, road distances are computed in the third and fourth updates, only if the second and third ones show distancing and approaching from the linear air distances. At the fourth update, the road distances are finally compared to either confirm or invalidate the hypothesis that the stop is passed.

Three boolean variables are used to keep track of what was the situation at the previous updates. Their meaning is written in the code extract below. All three variables are associated with an event that must happen two times; their value starts false and is true when the related event has already happened once. The other relevant fields for this algorithm are all int variables to keep track of the distances. In particular, those with the name starting with “old” have the value of the distance in the previous update and they are used for comparisons.

In the following extract there are three methods seen also in the code above: in the actual Java class, those are combined as the two criteria are working in parallel. The method updateViews of this class and the static calculateDistances of RealTimeTracker (utilities package) are not shown.

Whenever there is a location update, the passingCondition method is called: the two air distances to the stop 0 and 1 are computed. Then the method gettingCloseNext(Location location, int d0, int d1) is called. This method compares the newly computed distances (d0 and d1) with the ones from the previous update (oldd0 and oldd1), if distancing from the stop 0 and approaching to the stop 1 are detected, then a boolean is set. At the next update, when this method is executed, if the distancing approaching trend is the same then the method roadDistance(Location location) is called. This method calls the static RealTimeTracker.calculateDistances, which performs the two server calls to retrieve the road distances. Whenever a road distance is computed, the respective method setRoadd#(int roadd#) is called. These two twin methods, setRoadd0 and setRoadd1, simply save the distances and call compare(). The method compare() when both are computed sets a boolean. At the next location update, when the air distances will keep following the same trend and the road distances will be computed again with the server call, the method compare() will be called again. When both road distances will be calculated for a second time (roadd0 and roadd1), the method will compare them with the ones computed at the previous update (oldRoadd0 and oldRoadd1). If the distancing and approaching trend is confirmed by the road distances then the stop is considered passed and stopPassed() is called, otherwise the hypothesis will be discarded. The stopPassed() method resets the variable used, sets stop 0 as the previousStop and removes it from the list.

```
private Stop previousStop;   // previous stop
private List<Stop> stopsToGo;   // list of stops before getting off
private int oldd0;   // previous distance to the stop 0
private int oldd1;   // previous distance to the stop 1
private boolean gettingClose1;   // noticed distancing approaching with air
    distances also at the previous update
private boolean roadDistance1;   // computed road distances also at the previous
    update, they can be compared
private int roadd0;   // road distance to the stop 0
private int roadd1;   // road distance to the stop 1
private int oldRoadd0;   // previous road distance to the stop 0
private int oldRoadd1;   // previous road distance to the stop 0
private boolean calculatedd1;   // one of the two road distances has been computed

// called every time there is a location update: around every second
private void locationUpdated(Location location) {
  if (stopsToGo != null && stopsToGo.size() > 1) {
    passingCondition(location);
    updateViews(location);
  } else{
    if (stopsToGo != null && stopsToGo.size() == 1) {
      // notify the user to get off
    }
  }
}

// checks if the condition for the stop to be considered passed is valid
private void passingCondition(Location location) {
  int d0 = ((int) LocationToolbox.distance(location.getLatitude(),
      stopsToGo.get(0).getLocation().getLatitude(), location.getLongitude(),
      stopsToGo.get(0).getLocation().getLongitude(), 0.0, 0.0));
  int d1;
  if (stopsToGo.size() != 1)
    d1 = ((int) LocationToolbox.distance(location.getLatitude(),
        stopsToGo.get(1).getLocation().getLatitude(), location.getLongitude(),
```

```
            stopsToGo.get(1).getLocation().getLongitude(), 0.0, 0.0));
    else
      d1 = -100;

    if (d1 > 0)
      gettingCloseNext(location, d0, d1);
  }

  // checks if the bus is distancing the stop 0 and getting closer to the stop 1
  private void gettingCloseNext(Location location, int d0, int d1) {
    if (d0 > oldd0 && d1 < oldd1) {
      oldd0 = d0;
      oldd1 = d1;
      if (gettingClose1) {
        calculatedd1 = false;
        roadDistance(location);
        return;
      }
      gettingClose1 = true;
      return;
    }
    gettingClose1 = false;
    roadDistance1 = false;
    oldd0 = d0;
    oldd1 = d1;
  }

  // calls the method that performs the server calls to retrieve the road distances
  private void roadDistance(Location location) {
    RealTimeTracker.calculateDistances(this, location,
        stopsToGo.get(0).getLocation(), stopsToGo.get(1).getLocation());
  }

  // called by RealTimeTracker when the road distance to the stop 0 has been
      computed, it saves it
  public void setRoadd0(int roadd0) {
    oldRoadd0 = this.roadd0;
    this.roadd0 = roadd0;
    compare();
  }

  // called by RealTimeTracker when the road distance to the stop 1 has been
      computed, it saves it
  public void setRoadd1(int roadd1) {
    oldRoadd1 = this.roadd1;
    this.roadd1 = roadd1;
    compare();
  }

  // called every time a road distance is computed, checks that both were computed
      and if they were computed also at the previous update, then it checks if the
      bus is actually distancing the stop 0 and approaching the stop 1
  private void compare() {
    if (!calculatedd1)
      calculatedd1 = true;
    else {
      if (roadDistance1) {
        if (roadd0 > oldRoadd0 && roadd1 < oldRoadd1) {
          stopPassed();
          updateViews(null);
        }
      } else {
```

```
            roadDistance1 = true;
            calculatedd1 = false;
        }
    }
}

// called when the stop is considered passed, updates what needed
private void stopPassed() {
    previousStop = stopsToGo.get(0);
    stopsToGo.remove(0);
    gettingClose1 = false;
    roadDistance1 = false;
    oldd0 = 0;
    oldd1 = 0;
}
```

Strengths and Weaknesses

This algorithm fills the gaps left by the “20 meters” criterion. Its strength is that it works despite the quality of the GPS signal: its updates do not have to be extremely accurate. Moreover, the road distances are computed as walking ones and not driving ones, mostly because of bus lanes and U turns. In fact, as the walking path is the most similar to the bus route, it also provides the most accurate distances to the bus stops. In addition, in case of road work sites, it happens sometimes that the stop is moved forward or backward. In some cases, the stop is even moved to a nearby road, slightly changing the bus route. In these two situations, especially in the road change one, the “20 meters” criterion would probably not be able to detect that the stop passed as the location of it retrieved online would not be the actual temporary one but the usual permanent one. Instead, this distancing approaching criterion, would still be able to detect it eventually. Even if the bus goes on a parallel road, one or any blocks far, the distances will still increase and decrease the same way.

However, if the bus goes through a remote area, then it is possible that the overcoming of a stop is not detected. In fact, this criterion relies on the use of the internet connection to compute the road distances.

Considerations

By comparing strengths and weaknesses of both criteria, it is clear how the combination of both is the best solution to the problem of detecting that a stop is passed. The weaknesses of a criterion are the strengths of the other one and vice versa. Therefore, both are included in the algorithm. Their combination results in an anticipated prediction by a few seconds if “20 meters” works fine, or a few seconds delayed feedback, which is the ideal, if “distancing approaching” is involved. “20 meters” can detect that a stop is passed even before the bus has reached it, or while the bus is right at the stop. This can be source of confusion, especially if the next stop is the last one, as the user would be notified to get off when the bus is still at the previous stop. Unfortunately, there is no other accurate way to detect the actual overcoming of a stop without the use of internet.

Possible Improvements

This section is dedicated to the discussion of possible further implementations for this system, with a focus on its weaknesses, going over possible solutions.

Walking Directions

This application has been designed to be everything needed while using public transportations. Unfortunately, some useful features could not be implemented with the tools used. For example, Bus Waiting and Destination activities could display walking directions to better support the user while going to the stop or the actual destination. However, the online service in use does not provide directions, but only a series of points resulting in an unreadable text string. This feature would definitely be a good addition to the app. Even though Google Maps provides good and reliable walking directions that could be integrated, supporting them for a visually impaired user would be much more complicated.

Stefano Mattoccia and Matteo Poggi, from the University of Bologna, developed a working prototype of a wearable mobility system that can recognize objects and obstacles in front of the user and provide tactile and sound feedback[27]. This is based on 3D vision and machine learning, it has long battery life and can also classify the detected objects, like parked cars, walls, etc. In combination with Google Maps walking directions set on a smartphone, this system would be the ideal walking partner for visually impaired people.

Personal Data Vault

Another further implementation involves the use of a personal data vault to store the user's travel profiles and all the personal data like travel habits. This improvement is already in the pipeline, Francesco Fiacco is working on it, and will be provided in the next version of this application. The idea is having a layer of security and privacy protection to access sensible data. Each application that needs access to its data, has an associated authorization that is valid only for the data it strictly needs. The Personal Data Vault collects all the data and provides to every application only the data it is authorized to handle.

Remote Profile Management

Related to the creation of a Personal Data Vault, it would be useful to set the profiles by using a web browser. This would mean using a computer, probably easier for a visually impaired user. Plus, the ability to have the profiles remotely set by someone else. After authenticating, the website could also show data related to the previous trips, providing statistics, and other useful information.

Most Likely Next Trip

The integration with Nicola Ferroni's work on the most likely next trip is also in the pipeline. This would mean both feeding his algorithm with travel movements and getting back the trip that, based on previous history, the user is most likely to do[28]. This trip could be suggested when the

user shows the will to start a new trip, so when New Trip activity is opened. Or even better is suggesting the trip with a notification also when the app is not running.

New Layouts

Another possibility is creating a new set of layout views that are more compact. Despite the fact that they were designed for visually impaired people, the current layouts are definitely appropriate for sighted users as well. However, some aspects could result redundant or an overhead for a user without special needs. For example, the creation of a profile could be collapsed into a single screen instead of having multiple and potentially confusing ones. Other views could be reviewed as well. A new version of this system could be customized on the user's specific needs.

Tourism

Another improvement could consist in optional features specifically designed for tourists. Taking public transportations in another city, especially abroad, has always been source of confusion and uncertainty. As it is already usable for tourists as well, better after giving a language choice, an option could be adding preset trips to get to the main sights and move between them. Also having preset tours could be useful. A tourist would have the choice to use the app with travel profiles, or use preset trips. A further implementation could be providing also information about the sights, like historical facts, opening times, prices, etc.

Voice Controls

Voice controls would be another addition, especially useful for visually impaired users. The idea here is having the choice to control the app with the voice. So creating, editing and deleting profiles, but also choosing a travel profile and selecting a route, getting on and off the bus, etc. The user could be able to select a profile and set a departure time without going past the home activity. This addition is technically possible, but the usability could be limited: while moving, it is often hard to be surrounded by a quiet environment, suitable for voice recognition, therefore it could be source of misunderstanding. It would definitely be useful to manage travel profiles though, as the user would probably be home or in another relatively silent environment.

Map Integration

A better support for sighted people can also be the one provided by the use of a map. By showing information on it, displaying routes with origin and destination, a new or infrequent rider would have a much better idea when choosing a route. Also showing the position of the bus on the map can be helpful, both while on it and while waiting for it. It would be possible by using Google Maps and overlays with highlighted stops and routes. This would mean integrating the application with something like SlugRoute.

With the services in use, estimating the current bus location and showing it on a map would mean using the real-time ETA to go backward and estimate its position. This could be done by using also the stops prior to the one the user is getting on at (discarded in the current version), then making server calls to the planning service (bus routes between the stop the user is at and some

of the other stops the bus will go through before getting there), by comparing the travel times included in the routes downloaded and the bus ETA, it would be easy to get an estimated position. However, its accuracy would be very limited and fundamental for this implementation is that the planning service takes the current traffic into account and also that the ETA service is available and running for every bus around.

New Profile Concept

Another improvement regards the use of a travel profile: using it in both directions. If a travel profile takes from A to B, then it can take from B to A as well. This would save time creating profiles, but also reduce the list of profiles and make them more intuitive. When selecting a profile, the list would still show origins and destinations, but displayed accordingly to the current location. They could consider origin the closest of the two and then have an inversion button for the other way around in case the user wants to explore at a different time.

Service Notification

Having a service notification with the bus ETA while waiting it would also be useful for the user. As well as a notification displayed while on the bus showing the next stop, the distance to it, the stops and time left before getting off. These notifications would be updated regularly and by showing the most relevant information, the user would not have to open the app to know them. Instead, by pressing the device unlock button and checking the notifications, he or she would be informed straight away.

More Disabilities

Several systems have been developed over time to provide transit support to people with cognitive disabilities. The key is the ability for the caregiver to remotely set travel profiles, select routes and check the user's real-time location, by logging into a website. Regarding the travel support, the application should give very short and simple vocal instructions like "The bus is coming, get ready" or "Get off at the next stop, request the stop". Then if the system detects that the user missed the stop or got off at a wrong stop, then the caregiver would be contacted and the route recalculated. GoGoBus would have to work in the background and give only the few essential vocal instructions. It could be integrated with the ideas of TAD, Mobility Agents or Opportunity Knocks to support more people with special needs.

Other Platforms

Even though Android is the most common operating system, many use iOS by Apple and some others Windows Phone by Microsoft. The extension of this system to the iOS and WP platforms would increase its usage and reach a larger public. As for accessibility, iOS has a sophisticated service called VoiceOver, available across all the built-in Apple applications.

Route Selection with Delays

Particularly useful for commuters and frequent riders is having in a single screen all the relevant bus lines available at the stop with the associated delays. By integrating delays together with the

suggested routes, the user would always know if it is more convenient to wait a few minutes more to get a faster bus line (less stops), or simply take the first one (more stops). This means taking to an upper level what OneBusAway provides, as it would not only show all the routes and delays, but instead, display just the buses that take to the user's destination.

Home Screen Shortcuts

Useful for quick and easy access, being able to create shortcuts to specific travel profiles on the home screen could save time both to visually impaired riders and transit commuters. The click of the shortcut widget would bring straight to the route selection screen of the application, showing the next suggested routes.

Coverage

Finally, despite all these interesting additions and improvements, the most important one in my opinion is the extension to a larger area. Even though the application already supports the GTFS standard format, in its current version the app works only in Bologna. Due to the use of services provided by the local transportation agency, it provides transit support only across the city and its suburbs. For the planning as well, another service would have to be used in order to extend the coverage of the system. The application is developed this way because it is part of a local project related to the Region. However, further implementations should include and integrate the information coming from different local agencies. Integrating Google Maps to fill the gaps between the local agencies would make GoGoBus global, but it would not make sense as its support would be more limited and would never be an improvement from the actual Google's service.

Conclusion

I have presented a working system called GoGoBus which utilizes GPS sensor information and mobile data network to access existing online services and provide public transportation assistance. The main function of the system is to guide a user to a chosen destination, but unlike existing systems, its features address the needs of different kinds of users, including visually impaired people. For frequent riders it provides real-time arrival information; for new users, it has a travel planner. Most of all, for vision-impaired, it has a specifically designed UI and a tracking system to notify the rider when it is time to get off.

Developed on the Android platform and tested on real bus rides, the system has accurate routing and tracking information and is able to detect the overcoming of every bus stop, even in critical cases. Due to local infrastructure issues, the real-time arrival information is limited: not always available and reliable, it must be associated with scheduled times. The application is currently local as it relies on a series of online services developed for the city of Bologna and its suburbs. However, the system is able to process data in the standardized GTFS format, so it can potentially integrate also new and different areas.

I have described the application details and provided results from research studies showing that transit agencies should be interested in such mobility services to increase transit ridership and their customer satisfaction. There are plenty of possible improvements to push the GoGoBus concept even further, bringing the idea of mobility service to a completely new level.

References

Application code on GitHub at:
https://github.com/sgasperi/TesiSF

[1] Taylor, B.D., H. Iseki, M. A. Miller, and M. Smart. Thinking Outside the Bus: Understanding User Perceptions of Waiting and Transferring in Order to Increase Transit Use. *California PATH Research Report UCB-ITS-PRR-2009-8*, Univ. California, Berkeley, 2009.

[2] Watkins, K., B. Ferris, A. Borning, S. Rutherford, and D. Layton. Where Is My Bus? Impact of mobile real-time information on the perceived and actual wait time of transit riders. *Transportation Research Part A: Policy and Practice*, Vol. 45, No. 8, 2011, pp. 839–848.

[3] Ferris, B., K. Watkins, and A. Borning. OneBusAway: Behavioral and Satisfaction Changes Resulting from Providing Real-Time Arrival Information for Public Transit. *90th Annual Meeting of the Transportation Research Board*, Washington, D.C., January 23-26, 2011.

[9] Dailey, D.J., S. Maclean, and I. Pao. Busview: An APTS Precursor and a Deployed Applet. *Report No. WA-RD 467.1*, Final Research Report, NTIS No. PB2001- 100471, Prepared for Washington State Transportation Center, Seattle, 2000.

[10] Patterson, D.J., L. Liao, K. Gajos, M. Collier, N. Livic, K. Olson, S. Wang, D. Fox, and H. Kautz. Opportunity Knocks: A System to Provide Cognitive Assistance with Transportation Services. Proceedings of the International Conference Ubiquitous Computing (UbiComp), 2004.

[11] Barbeau, S., P. Winters, R. Perez, M. Labrador, and N. Georggi. Travel Assistant Device. Aug. 11 2006. US Patent App. 11/464,079.

[13] Repenning, A. and A. Ioannidou. Mobility Agents: Guiding and Tracking Public Transportation Users. In *AVI '06: Proceedings of the Working Conference on Advanced visual interfaces. ACM*, 2006, pp. 127–134.

[14] Ferris, B., K. Watkins, A. Borning. OneBusAway: A Transit Traveller Information System. *International Conference on Mobile Computing, Applications and Services*, MobiCase, 2009.

[16] Barbeau, S.J., A. Borning, and K. Watkins. OneBusAway Multi-Region – Rapidly Expanding Mobile Transit Apps to New Cities. *Journal of Public Transportation*, Vol. 17, No. 4, 2014, pp. 14-34.

[17] Ferris, B., K. Watkins, and A. Borning. OneBusAway: Location-aware tools for improving public transit usability. *IEEE Pervasive Computing*, Vol. 9, No. 1, 2010, pp. 13-19.

[27] Poggi, M., and S. Mattoccia. A wearable mobility aid for the visually Impaired based on embedded 3D vision and deep learning. *First IEEE Workshop on ICT Solutions for eHealth* (IEEE ICTS4eHealth 2016) in conjunction with the *Twenty-First IEEE Symposium on Computers and Communications*, June 27-30, 2016, Messina, Italy.

[28] Ferroni, N. Un sistema di previsione degli itinerari per applicazioni di smart mobility. 2016.

Online References

[4] Android. Retrieved in September 2016 from: https://www.android.com

[5] Wikipedia: Android. Retrieved in September 2016 from: https://en.wikipedia.org/wiki/Android_(operating_system)

[6] Wikipedia: Usage Share of Operating Systems. Retrieved in September 2016 from: https://en.wikipedia.org/wiki/Usage_share_of_operating_systems

[7] Google Play: Google Talkback. Retrieved in September 2016 from: https://play.google.com/store/apps/details?id=com.google.android.marvin.talkback

[8] Google Play Store. Retrieved in September 2016 from: https://play.google.com/store/apps?hl=en

[12] University of South Florida: Location-Aware Information Systems Laboratory. Travel Assistance Device. Retrieved in September 2016 from: http://www.locationaware.usf.edu/ongoing-research/projects/travel-assistance-device/

[15] Google Play: OneBusAway. Retrieved in September 2016 from: https://play.google.com/store/apps/details?id=com.joulespersecond.seattlebusbot

[18] Shores, A. (2016, April 26). UCSC News: Campus launches mobile app to track shuttles in real time. Retrieved in September 2016 from: http://news.ucsc.edu/2016/04/slug-route.html

[19] Slug Route. Retrieved in September 2016 from: http://slugroute.com

[20] GitHub: SMAll. Retrieved in September 2016 from: https://github.com/small-dev/SMAll.Wiki/wiki

[21] Wikipedia: JSON. Retrieved in September 2016 from: https://en.wikipedia.org/wiki/JSON

[22] Trasporto Passeggeri Emilia-Romagna (Tper). Retrieved in September 2016 from: https://www.tper.it

[23] Tper: Hello Bus. Retrieved in September 2016 from: https://www.tper.it/hello-bus

[24] Google Play: Lockito. Retrieved in September 2016 from: https://play.google.com/store/apps/details?id=fr.dvilleneuve.lockito&hl=en

[25] GitHub: Retrofit. Retrieved in September 2016 from: https://square.github.io/retrofit/

[26] JSONSchema2Pojo. Retrieved in September 2016 from: http://www.jsonschema2pojo.org