HandiGo: Enhancing Rideshare for the Visually Impaired

Shishira Bhavimane, Omayr Farrag, Jake Bilbro
University of Chicago
CMSC 23400: Mobile Computing

INTRODUCTION

While technology is rapidly becoming more inclusive, many popular apps and services are still highly inaccessible for the visually impaired. Current rideshare apps like Uber and Lyft make it extrememly difficult for the visually impaired to navigate to the correct car and confirm that they are in the correct car with the correct driver because they cannot verify the license plate number by simply looking at it. Due to this lack of accessibility, visually impaired people are susceptible to bad actors that might pretend to be their driver. Because typical rideshare apps make navigation difficult, the visually impaired often take longer to get to the car, and they may not be able to see any messages coming from the driver. This leads drivers to grow impatient and, in many cases, has led to discrimination against the visually impaired. We aim to eliminate these problems with our own rideshare app– HandiGo. HandiGo is a rideshare service specifically for visually impaired people. While anyone may use the app as a passenger, the drivers know that it is likely that their passenger is visually impaired, so they will be much more likely to be kind, patient, and understanding towards all types of passengers. HandiGo also uses bluetooth low energy and iBeacon technology to guide the passenger to the correct car and do an additional driver-passenger verification step by just touching phones.

BACKGROUND

To find potential solutions for using Bluetooth Low Energy (BLE) as a measurement of proximity, a promising candidate was found in iBeacon. Others have shown iBeacon's efficacy and potential use as a tracking device for luggage. [2] Limitations were revealed by the approach, with determining the actual distance to the luggage taking upwards of 26 seconds. The displayed distance was also inaccurate. Yet, the environment in which this was tested presented many barriers between the beacon and the user's phone; much more than would exist between a passenger near the curb and a driver. This paper was also written in 2014, and many improvements in Bluetooth technology were likely achieved since then.

To determine the feasibility of using iBeacon for such a task, a more modern paper was sought out. [1] On average, the accuracy of the distance calculated by iBeacon is on average accurate between 0.79 and 2.28 meters. Since the average length of a car is around 4-5 meters, we were confident that this would be able to guide a passenger to the immediate vicinity of their driver's vehicle. In relation to our project, it was worth considering that metal has a high propensity to block the BLE signals emitted by iBeacons. This meant that our approach may have trouble accurately depicting the distance between a driver and the passenger if the car is interfering. In an ideal situation, in addition to the driver's phone, the driver's car could also be outfitted with a dedicated iBeacon transmitter to mitigate interference. Having two beacons could also allow for some sort of triangulation between the devices.

The body of literature gives the impression that iBeacon is a tool that has many potential uses, but with concerns about the accuracy and reliabilty of BLE signals. However, it has been shown to be useful in locating and ranging an object of interest, which is precisely what is required for the implementation of HandiGo.

SOLUTION

Simply put, our solution involves many of the same user interface features as any typical rideshare app such as passengers requesting a ride by giving their name, pickup, and drop off location and drivers looking at a list of passengers and choosing which ride they would like to accept, but it also adds additional features like navigating the passenger to the correct car by vibrating and making a sound when the passenger is going in the correct direction as well as verifying the correct driver and passenger by clicking the confirm ride button that appears when the phones touch together. We used SwiftUI and XCode for the app because many of the packages we wanted to use were easy to integrate with iOS app development. Our solution can be broken down into the frontend, backend, and iBeacon implementations, which are described below.

Frontend

In SwiftUI, you have the ability to create different views that the users will see in the app itself. The `HandiGoApp.swift` file is what the app runs when it is opened. In this file, you can see that it shows the view in the `ContentView.swift` file. This essentially functions as the home screen for the app where the user has the ability to press a button that describes if they are a driver or a passenger. Ideally, there would be a login/registration page where we can keep track of who is a registered driver/passenger, but we chose to do a simple user interface implementation for our app because of our time constraints and lack of familiarity with the platform.

The app starts running with the `ContentView` view, but the rest of the views in the app are accessed through navigation links. Essentially, whenever the user presses a certain button it takes them to a new view. We will walk through all of those views by looking at the app through the driver and passenger perspectives.

Driver Perspective

Once the driver clicks on the "Driver" button on the main page, they will be taken to the `DriverView` view where they can see a navigation view of all the active passengers in the database. In the navigation view, the driver has the ability to press on any one of the passengers. Every passenger is stored in the database with their name, pickup location, and drop off location, and that list is fetched from the database with this view. The `DriverViewModel.swift` file contains the code to actually fetch a list of the passengers stored in the database.

When the driver clicks on a certain passenger, they see the `AcceptRideView` view where they have the ability to click a button to accept the ride of the passenger. If they are not sure, they have the option to go back and select a different passenger. Once they press the "Accept Ride" button, a UUID is generated for the driver and is then stored (along with the passenger details) in the "rides" collection of the database, which tracks the active rides. In the `RideViewModel.swift` file, there is code to add a ride to the database. A "ride" contains the driver UUID, the passenger name, the pickup location, and the drop off location. A UUID is a Universally Unique IDentifier. This UUID is what will be used to make the driver's phone a beacon and allows the passenger to see how far away they are from it.

After the driver has accepted the ride, they are taken to the `BeaconView` view where they can see an "I've arrived" button. Ideally, we would use some sort of navigational feature to get the driver to the passenger and then the drop off location, but for our more bare bones application, we decided to leave that out. Once the driver arrives at the pickup location, they can press the "I've Arrived" button, which starts emitting their phone as a beacon using the generated UUID that was passed in from the `AcceptRideView` view.

Passenger Perspective

Once the passenger clicks the "Passenger" button on the home screen of the app, they are taken to the `PassengerView` view where they can fill out a form with their information (name, pickup location, and drop off location). You can see a request ride button at the bottom, but that is grayed out (and cannot be pressed) until there is something in each of the fields. Once that button is pressed, the code in the `PassengerViewModel.swift` file adds that passenger to the database.

After the passenger has requested the ride, they can see a navigation view of all of the active rides, which is fetched from the database using the code in the `PassengerRideViewModel.swift` file. Their own ride will not appear until the driver has actually accepted the ride. Ideally, the passenger would not be able to see all rides and only the ride that pertains for them, but for our bare bones implementation, it was easier for us to implement it this way.

After the passenger sees their ride appear (they will need to refresh by pulling the screen down), they will be able to click on their ride. Once they do that, they will see the distance they are from the beacon and a "Start Monitoring" button. As a default, the distance is just the string "Driver has not arrived yet." Once the passenger presses the "Start Monitoring" button, they will start monitoring for the beacon with the driver's UUID. If the driver has not pressed the "I've arrived" button, the same default string will appear because the driver has not advertised their phone as a beacon. Once both the driver has arrive and the passenger has started monitoring for beacons, the passenger will see the distance in meters from the beacon appear on the screen. If the passenger gets closer to the beacon (the distance decreases), their phone will vibrate and make a faint sound. The goal is for the passenger to follow the vibrations/noises to the car.

After the passenger navigates their way to the car, there is an additional verification step that they must perform before the ride begins, which is touching their phone with the driver's phone. After their phones touch (the distance from the beacon is below 5cm), a green "Confirm Ride" button appears on the passenger's screen that they can click for the ride to begin!

Backend

In order to communicate between passengers and drivers using different devices, it was necessary to use a backend. To accomplish this, a free MongoDB database was used. Basic information is added to the database to share crucial information between users. There are two data models used: driver, containing a name and UUID, and passenger, containing a name, pickup location, and dropoff location.

To connect to the backend, a server was run on localhost using a Swift web framework called Vapor. This handled all requests between the MongoDB database and the application. However, an app built on an iPhone cannot directly connect to localhost. To work around this, the service `ngrok` was used. This creates a globally accessible link to access localhost from devices outside the network. This enabled seamless transfer of data between users at zero cost.

iBeacon Implementation

The main instance of the iBeacon implementation occurs in the iBeaconTestingView file of our project. The code in this file implements a system for measuring the distance between two devices (phones) using Bluetooth beacons. It utilizes the CoreLocation and CoreBluetooth frameworks in conjunction with SwiftUI to create the functionality.

The code consists of two main classes: LocationManager and BeaconManager. The LocationManager class handles location-related operations. It requests user authorization for location services and monitors the beacons to calculate the distance between devices. The distance is determined by ranging the beacons and using the accuracy property of the closest beacon. The distance information is stored in the distanceString property, which is updated using the @Published attribute.

The BeaconManager class manages the Bluetooth beacon-related functionality. It initializes a CBPeripheralManager to handle peripheral operations and advertises a Bluetooth beacon using the provided beacon UUID. The beacon is represented by a CLBeaconRegion object, which contains the beacon UUID and identifier. The startAdvertising() method is responsible for starting the beacon advertising process.

In terms of the user interface, the code provides two SwiftUI views: BeaconView and DistanceView. The BeaconView is displayed on the device acting as a beacon and provides a simple UI with a button to start advertising the beacon. The DistanceView is displayed on the other device and shows the distance to the beacon. It also provides a button to start monitoring the beacons. When the monitoring starts, the LocationManager initiates ranging beacons and updates the distance value as the closest beacon is detected and the distance is calculated.

In terms of the navigation between the passenger and the driver, we implemented a method of vibration to aid the passenger to the driver's car using the calculated distance between the iBeacon device and the passenger device. As the passenger is approaching the driver, the passenger's phone vibrates and makes a faint noise to inform the passenger that they are walking in the correct direction. If it happens that the passenger is not getting closer to the driver anymore, the phone stops vibrating, our way of informing the passenger that they are walking in the wrong direction.

Lastly, our solution to help ensure the passenger is connected to the correct driver is to have the passenger and driver phone touch, and thus, only in that instance is the ride able to be executed further. To implement that, we added a threshold of 5 cm, such that once both phones were in a distance of 5 cm of each other, that would trigger a button on the passenger phone appear to continue with the ride.

Overall, the app creates a basic system where one device acts as a beacon and the other device measures the distance to the beacon using Bluetooth technology. It showcases the usage of CoreLocation and CoreBluetooth frameworks in conjunction with SwiftUI to implement this functionality.

EVALUATION & RESULTS

We tried testing our app in many different situations and scenarios to best understand the strengths and weaknesses of our app. We noticed that the app successfully emits the distance from the passenger to the iBeacon driver. However, there may be some accuracy issues and time delays in the distance calculations. Most of the inaccuracies that we experienced was when the passenger and driver were more than 15 meters apart; this means that the iBeacon emission does not work best for long distances. These position inaccuracies can happen for a couple of reasons:
1) Signal Interference: The accuracy of Bluetooth-based distance estimation can be affected by signal interference from obstacles, such as walls, trees, other cars, or other electronic devices. These obstacles can cause signal reflections, attenuation, or distortions, leading to inaccurate distance measurements.
2) Signal Strength Variations: Bluetooth signal strength can fluctuate due to environmental factors like electromagnetic interference or radio frequency noise. These variations can impact the accuracy of distance calculations.
3) Beacon Placement: The position and placement of the iBeacon can also affect accuracy. If the beacon is not positioned optimally or is obstructed, it may result in inaccurate distance measurements.

However, we did notice that the distance measurement when the passenger phone was in about a 3 meter radius of the beacon of the driver, it became extremely accurate. Additionally, when it was within 1 meter of the driver, it was able to give distances accurate down to the centimeter.
Another element of our app the we noticed needs to be addressed is the fact that the correct distance that gets printed takes a couple of seconds to register, thus giving the passenger a slight time delay in terms of his/her position in real time. Some reasons for this might be that:
1) The code in the provided implementation uses the startRangingBeacons(satisfying:) method, which relies on ranging beacons at a specific frequency. This frequency may introduce a delay in updating the distance value, as the ranging process might not occur continuously or instantly.
2) And/or the code may perform some form of distance averaging over a period to reduce fluctuations caused by signal variations. This averaging process could introduce additional delay in updating the distance value.

Thus, to help address these issues, it might be smart to figure out ways to emit a stronger beacon signal from the phone of the driver, if that is possible. Perhaps, one solution might be to implement signal strength filtering techniques to reduce the impact of signal fluctuations and improve accuracy. This can involve techniques such as moving average filtering to smooth out the distance measurements. Another solution could be to optimize the placement of the iBeacon to minimize signal interference and obstructions. Conduct a thorough analysis of the environment where the beacons are deployed and make adjustments accordingly.

CONCLUSION

HandiGo offers a safe and optimal solution to those who are visually impaired and are looking to use ride-share services. Although our current implementation is not fully expansive, it offers a great basis and starting point of achieving those goals of becoming a great option to those seeking safe and optimal solutions for ride-share services. Our technology offers a great way to help navigate those who are visually impaired to the correct driver, ensuring that the passenger is indeed connected with the right driver. To fully ensure that fact, one of our favorite and safest features of the app is the action of the passenger phone having to touch the driver phone to continue executing the ride. Only when that happens does a button appear on the phone of the passenger to continue the ride.
Looking forward, in a more ideal and effective version of this app, we would like to implement a more audible version with speech aid to help the passenger with his navigation to towards the passenger, that way the passenger gets a more informative experience with their position with respect to the driver. Additionally, to help with the accuracy of the iBeacon detection, a more feasible solution would be for the driver to have an actual iBeacon device, thus the emission of the signal from that device would be much more accurate than that of the phone, ultimately increasing the accuracy and experience of the passenger in their navigation towards the driver. Lastly, we would like to implement additional safely features for the passenger, specifically once the passenger actually starts the ride. If the passenger is visually impaired, then they might not be able to correctly identify if the driver is driving along the correct path according to the GPS or not. Thus, if the driver is veering off of the GPS path, we can inform the pas-

senger, and perhaps contact emergency contacts, or any other precautionary measure can be conducted.

REFERENCES

[1] M Fachri and A Khumaidi. 2019. Positioning Accuracy of Commercial Bluetooth Low Energy Beacon. *IOP Conference Series: Materials Science and Engineering* 662, 5 (nov 2019), 052018. DOI: http://dx.doi.org/10.1088/1757-899X/662/5/052018

[2] Markus Kohne and Jurgen Sieck. 2014. Location-Based Services with iBeacon Technology. (Nov 2014), 315–321. DOI:http://dx.doi.org/10.1109/AIMS.2014.58

GROUP MEMBER CONTRIBUTIONS

Shishira worked on the frontend implementation, the project presentation slides, and the introduction and frontend sections of the report.

Jake worked on the backend implementation and the backend and background sections of the report.

Omayr worked on the iBeacon implementation and the iBeacon, evaluations and results, and conclusion sections of the report.

While we all started with these individual assignments, to connect everything together, test, and debug, we all ended up working on and collaborating on all three parts of the application. Multiple people even worked on some of the same things before we combined everything into one end-to-end project so that we all could try and implement everything on our own and try out multiple ideas.