Name: William Gong

Student Number: 61248415

Team: 11

Title: Sabotage

Contributions/Detailed Design:

General:

I was tasked with the app portion of our project (frontend and backend), which is our main UI, alongside the VGA touchscreen. The app was coded using Java on Android Studios. It's connected to the rest of the project through the server, as well as the Wi-Fi module.

UI Design (Figma)

I started by creating draft designs, using Figma. I discussed this with my group to create a design that would fit our project specifications. I would mainly use these designs as reference, when creating the actual designs in Android Studios; however, as certain issues arise, or changes in specifications were made, certain aspects of these drafts were modified, or completely unused. These drafts (with descriptions of what will happen on each screen and a flow diagram) can be viewed in our Github or with this link:

https://www.figma.com/file/olH5i8Xk7tbdFAkaufjcTU/Untitled?node-id=0%3A1

Figure 1.1: Start Screen

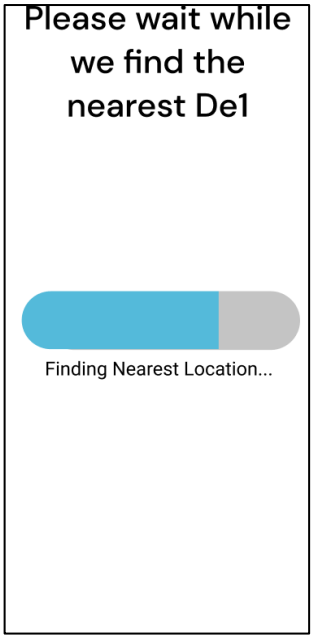

Figure 1.2: Loading Screen

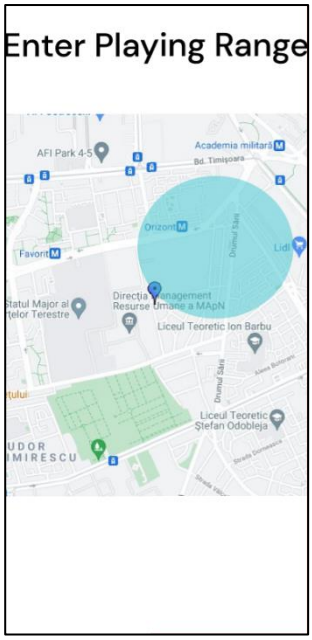

Figure 1.3: Maps Screen

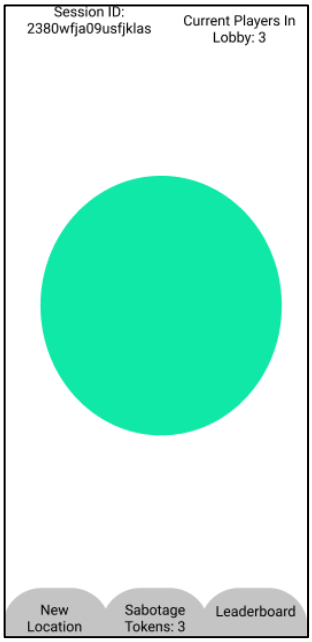

Figure 1.4: Hot or Cold Screen

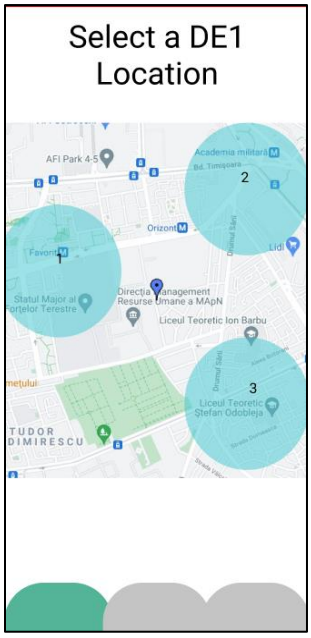

Figure 1.5: Other Locations Screen

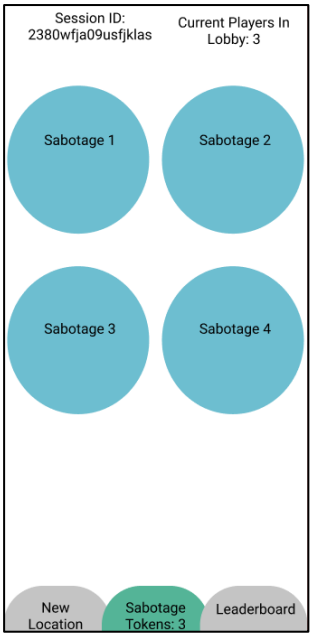

Figure 1.6: Sabotage Screen

Figure 1.7: Leaderboards

Figure 1.8: Winner Screen

GPS and Wi-Fi State Permissions

Players will begin on this start screen:

Figure 2.1: Actual Start Screen

From here, pressing the START button will ask for GPS fine-location (accurate within ±10m outdoors) permission from the user. It may also ask for Wi-Fi state permission, although it depends on the API level of the user's Android phone. These permissions allow us to track the user's current GPS location, as well as see available Wi-Fi signals nearby, both of which are necessary for implementing the later "Hot or Cold" feature. If the user denies access to these permissions, they will be given an error, saying these permissions are necessary for the app to be used, and the user will not be allowed to play the game, until they have given permission. However, once they do give permission, they're permission status will be saved, and as long as they do not change their permissions in their Android settings, they will no longer be prompted to give permissions to play the game and will immediately start upon pressing the start button. I implemented this feature myself but adapted the code from **1)**

GPS Tracking

Once players have pressed the start button, and they have given permission to GPS access, GPS tracking will be in effect. GPS tracking will use fine-location tracking which is accurate within ±10m, when outdoors (basically the same kind of GPS tracking Google Maps uses). This is done using the LocationListener interface and LocationManager class to use the Android phone's built in GPS capabilities to return the location of the phone in a latitude, longitude (LatLng) object. This location is only updated if the location of the player moves a minimum distance from their previous location in a minimum time interval. For example, if the minimum distance is set to 5m and the minimum time interval is 1 second, if the user moves 6m, then the location will be updated at the 1 second interval. However, if the user moves only 2m, then at the 1 second interval, the location will not be updated. The minimum distance and minimum time interval are constants MIN_DIST, and MIN_TIME respectively, and they are set as 5m and 1 second in the constants class.

Within the LocationListener interface, we can override the public void onLocationChanged(Location location) function to not only update the location, but also to check certain conditions, such as if the player is still within the play radius, and if they are within a valid session. This function is written using a try catch statement to avoid crashes and report errors, as GPS signal cannot be guaranteed all the time. The catch block will print the stack trace if location was not found, which was a useful feature for debugging.

The GPS tracking itself was not too difficult to implement, but because many screens relied on the GPS tracking and had specific conditions about the user's location, each LocationListener interface had to be created carefully to meet these conditions, which took a lot more effort to maintain. For example, the Hot or Cold Screen (Figure 1.4), the Sabotage Screen (Figure 1.6), and the Leaderboards Screen (Figure 1.7) are all specific to whichever session a player is currently in. That means if a player's updated location is outside a session's play radius, they must be kicked out, back into the Maps Screen (Figure 1.3) to let the user know they have exited the session, and then update the session's player count accordingly. I implemented this feature myself, with the use of the Android studio Docs on LocationListener **2)** and LocationManager **3)**.

HTTP POST/GET Requests to Server

The POST and GET requests were mostly done using the Retrofit API. This API allows POST requests to be created in Java classes (with proper formatting and naming), and then converts that Java class into a JSON object which is sent to a URL. Similarly, it can send a GET request to a URL, receive the returned JSON object, and convert it into a Java class (with proper formatting and naming), which can then be parsed for useful information. There are two types of GET requests and one type of POST request. A Sessions GET which receives data about all available sessions, and a Lobby GET that receives information on the current lobby a user is in.

The URL is a constant located in the constants class, so that if our URL changes, we only need to make one change within our code to update the URL.

The interface that allows the app to make GET and POST request has two functions to override. The onResponse function and the onFailure function. When the GET/POST request makes it to the server and we return with either a success, or an error code (e.g. error 400), we use the onResponse function. Here is the pseudocode for onResponse:

```
If(!successful) {
Display error code in unused textbox
} else {
Extract data from JSON object and/or update values
}
```

When the request fails due to: syntax error, null object, incorrect formatting, cannot find server, invalid values, we will use the onFailure function, which will display the failure message, and print to the console the failure cause. This allows for very convenient error checking.

Each POST and GET request are done on a queue that runs on a separate background thread as to not interfere with the main thread. In most cases, the only areas where this could cause race conditions are when a POST request is meant to be executed right after a GET request or vice versa. This could be remedied by running POST and GET request on the main thread, but I found it was easier and smoother if the second request is called in the onResponse function when the first succeeds (elaborated in challenges).

I wrote all the POST/GET code on the app side. I used this YouTube series **4)** to learn how to use Retrofit, but all the code was written by me. The POST/GET requests were not too difficult to implement, the main difficulty came from trying to figure out where a request was necessary and what information was required.

Basic Login Page

This was a very simple login page that checks a user ID.

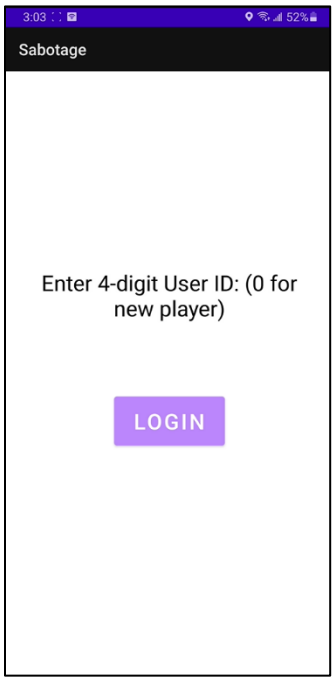

Figure 3.1: Actual Login Screen

This screen would come up right after pressing the Start button in Figure 2.1. It prompts the user for their user ID. It will then check to make sure if the ID is a number. If it is not a number, it will display an error message letting the player know only numbers are accepted. If a player is new, they enter 0, which will send a Sessions GET request to the server which will return a randomly generated unused 4-digit user ID, display it, and then send a POST request, letting the server know that this user is now playing the game. If the player remembers their previous 4-digit user ID, they may enter that instead, which will only send a POST request to the server. If the server does not find the user ID in the database, it will return with an error code which will be displayed for the user to see. All app-side code is written by me, however my code simply communicates with the server, I did not write any of the server code that handles the POST/GET requests.

Map De1 Locations

This screen would be displayed right after a user logs in from Figure 3.1, or if a user exits a session (described later).

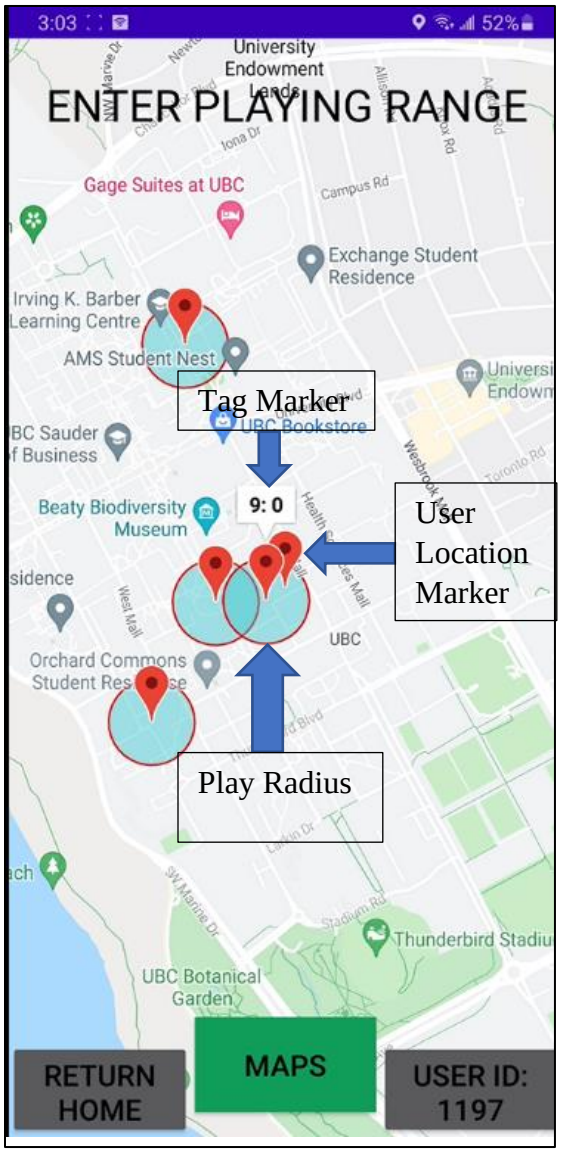

Figure 4.1: Actual Maps Screen

The user is shown each active session, a blue circle, which represents a De1-SoC with a GPS module, VGA touchscreen, and a Wi-Fi module. This data is obtained from a Sessions GET request. The user's location is shown as a red marker, that will display the user's current location using GPS tracking. The play radius is the blue circle, which a player must be within to be able to start playing hot or cold (Figure 1.4). If the player is within the radius, they can press on the blue circle to start the game. Joining a session will send a POST request which updates the player count of that lobby. The radius can be adjusted in the constants class and is set to 100m. Each play radius will have a marker at its center. Pressing the marker will reveal information in the format:

Session ID: Numbers of Players in Session.

In this example, the session ID is 9, and there are 0 players in that lobby.

Note, the marker does not represent the actual location of the De1-SoC. I use a random distance that is less then the radius (which is 100m), and a random direction that is less than 360 degrees, and the center of the circle is the offset of that random distance/heading from the actual De1-SoC

The conversion code of this offset into a new latitude and longitude value pair, was adapted from this source **5)**. This adds randomness but confirms our De1-SoC is still within the play radius. We add randomness so that players cannot simply walk to the middle of the circle and start searching from there. The randomness is seeded based on the session ID, so that the randomness

is consistent among all players, and players cannot share their maps to try and triangulate the location (elaborated in challenges).

A player can return to the start screen (Figure 2.1) by pressing the return home button on the bottom left.

I implemented this feature myself using the Google Maps API **6)**, as well as the Circle Class **10)**.

Note: From here on, these features require that the user is within a session. If the user exits the session ID by leaving the play radius, pressing return home, or pressing maps, they will not have access to these features until they re-enter the session. Furthermore, when they leave a session, a POST request will be sent to the server indicating that the player has left the session and to decrement the player count.

Hot or Cold Button (using GPS + Wi-Fi RSSI level)

This is where the main hot or cold game is played by pressing the big green button.

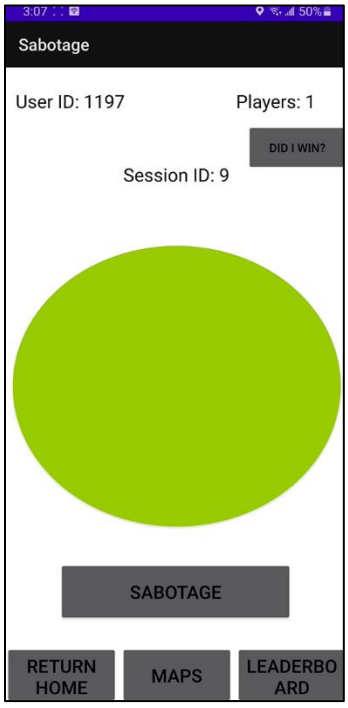

Figure 5.1: Default hot or cold Screen

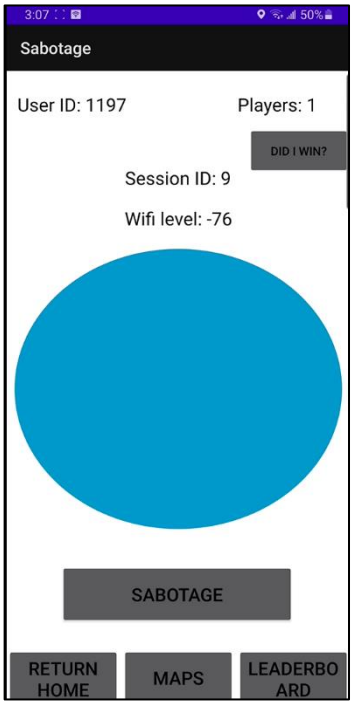

Figure 5.2: Cold Screen

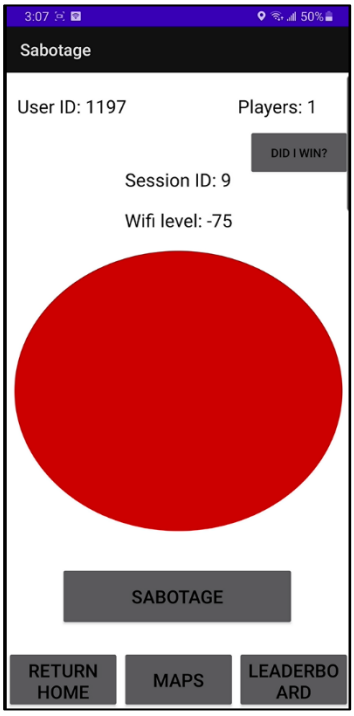

Figure 5.3: Hot Screen

Figure 5.4: Not Within 20m Screen

The user can press the big green button (Figure 5.1), to check whether they are closer to the De1-SoC (hotter = red) or further (colder = blue). This is done in two different ways.

1. Using GPS location
2. Using Wi-Fi RSSI level

If the user is outside of the SMALL_RADIUS (a constant in the constants class) which is set to 20m, the app will take their previous location, and compare it to their current location and see whether they are further or closer, then it will display the corresponding color and vibrate with the corresponding duration. Once pressed, the player's previous location will be updated to be the location at whenever they pressed the button, and the button will be on a 3 second cooldown, where it cannot be pressed again to avoid players spamming the button. Once the 3 seconds are up, it will change back to green and be able to be pressed again. If a player is just entering a session, since they have not yet pressed the button, their previous location will be set to the location at which they joined the session. The app will also let the player know they are not within the SMALL_RADIUS as shown in Figure 5.4.

Once a player enters the SMALL_RADIUS, the concept is the same as before, but instead of using location, it will use the Wi-Fi RSSI level to determine if the user is hot or cold. (Note RSSI is measured in dBm, where all values are negative, but a less negative value is a stronger signal). Similar to the location based hot or cold feature, the first time the player enters the SMALL_RADIUS, since they have not pressed the button yet, the previous Wi-Fi RSSI level is set to -100, a signal strength that is too weak to be found. To further aid the player, the Wi-Fi RSSI level is also displayed (Figure 5.2 and Figure 5.3). It is possible for the app to not pick up a Wi-Fi signal despite being within 20m for multiple reasons. As a fail-safe, if this happens, the app will display a darker red button, upon button press, and the text will let the player know they are within 20m.

I implemented this feature myself. For the Wi-Fi RSSI scanning I used the ScanResult class and WifiManager to locate the Wi-Fi RSSI value of the De1-SoC Wi-Fi module **7)**. It started very easy to implement but as more and more features were added to the hot or cold screen it became more difficult (elaborated in challenges).

Winning

As shown in Figures 5.1 – 5.4, the top right has a “DID I WIN?” button. When pressed it will send a Lobby GET request to the server, check the response, and display the corresponding message.

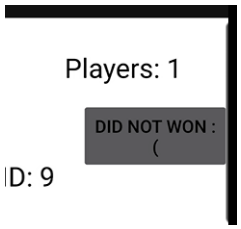

Figure 6.1 Did not win

Figure 6.2 Winning

Sabotage

This feature allows players in the same session to sabotage one another.

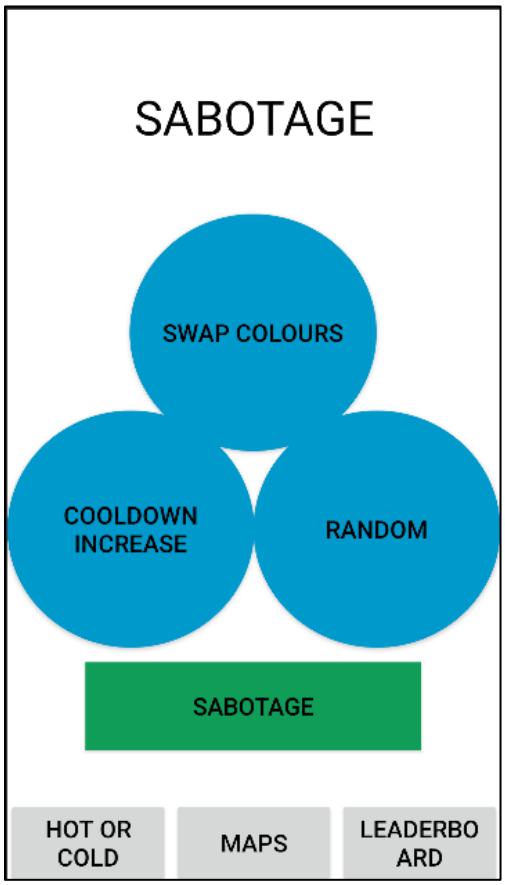

Figure 7.1 Actual Sabotage Screen

A player can sabotage other players in the same session by pressing one of these three buttons. Before being able to press them, it will check if there is at least one other player in the session, if the saboteur has at least one sabotage token, and if the session is currently not being sabotaged. If any of these fail, a corresponding text message will let the user know why their sabotage failed.

If a sabotage goes through, it will use up one of the saboteur’s sabotage tokens. Each sabotage requires one sabotage token. New players start with 3 and winning a game will reward the player with 2 more tokens.

Victims will be alerted that they are being sabotaged.

The sabotage will affect everyone in the session except for the saboteur for 30 seconds. In that time, there are 3 different sabotages that may affect other players in the lobby.

1. Cooldown Increase: As stated before when a player presses the hot or cold button, their button will go on a 3 second cooldown. This sabotage changes that 3 second cooldown to 10 or the rest of the duration of the sabotage, whichever is shorter. For example, if there is 20 seconds left on the sabotage, then the cooldown will only be 10 seconds. If there is 5 seconds left on the sabotage, then the cooldown will be 5 instead of 10.
2. Swap Colors: This simply reverses the color of the buttons. If the player is hot, it will display blue instead, and if the player is cold, it will display red.
3. Random: This has either the possibility to do the same thing as swap colors or do nothing. The randomness is seeded on the saboteur's user ID, so it is consistent throughout the lobby. This was added so that players will not be sure if their colors have been swapped or not. If they knew their colors were swapped, they can simply follow the blue button instead of the red one. Now players will have to check whether they are still going in the right direction or not.

Implementing new sabotages in the future is easy, as each sabotage only requires a new button, and a sabotage ID, that is sent as a POST request. This sabotage ID will then be received by all other users in the lobby and read in the hot or cold screen, where a corresponding sabotage will be performed.

I implemented all the sabotage code on the user end myself.

Leaderboard

Sends a Lobby GET request to the server, parses the leaderboard data and displays the session leaderboard. Leaderboard parsing and display was created by me. There is also a possibility for extension by implementing a global leaderboard. This would only require adding the global leaderboard feature to the Sessions GET request and updating the server code.

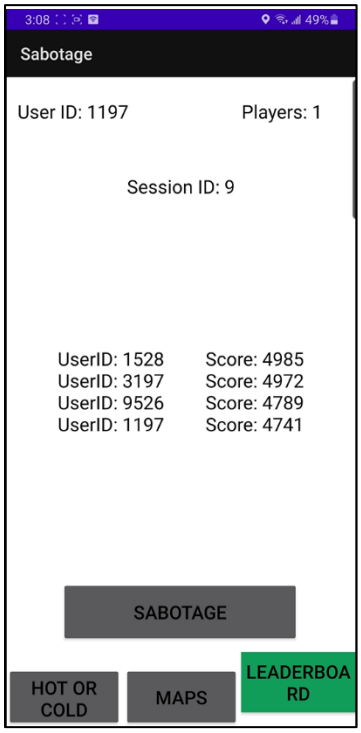

Figure 8.1: Actual Leaderboard

Challenges

1. Difficulty of Hot or Cold Screen (Figure 5.1 – 5.4)

Because the Hot or Cold Screen was the main feature of our app, it was the most important and took the most time. It started quite simple, but as the project went on, more and more features were added onto the Hot or Cold Screen, and some of them would interfere with other features. This included: GPS location tracking, Wi-Fi RSSI reading, winning button (Figure 6.1), making sure the player stays within the session play range, updating player count, handling sabotages. Furthermore, each feature had many "corner cases" that had to be handled, such as: no GPS signal, cannot find Wi-Fi RSSI, and possible POST/GET errors. At some point, I had to restart from scratch, and systematically build each feature with extensive error checking and test it with test cases, before being able to combine them.

2. Consecutive POST/GET requests

As stated in the POST/GET section above, since the requests are enqueued and run on separate threads in the background, there were possibilities of race conditions when two requests were

made back to back, which could be a problem if one request was dependent on data from another request. For example, new user logins use a GET request to retrieve a new user ID, and then posts that new user ID to let the server know they're logged in. If the POST request does not receive the user ID before posting, it will cause a failure. I could call both on the main thread, so they would be sequential, but that made the app feel slow and unresponsive. To fix this, I just let the POST be called on the successful response of the GET request, which means that the POST cannot go through until the GET requests data is confirmed.

3. Randomness in map circles

My first attempt, to create randomness for the circle, I would create a random object each time the map was opened. That means the randomness was different each time the map was opened. This caused two problems.

1) A player can constantly exit to maps and try to triangulate the location of the De1-Soc.

2) If a player exited the play radius and got kicked back into the maps screen, it may display that they are still within the circle, since a new circle was created, which could cause confusion for a player.

This problem actually had a simple fix. I just seeded the randomness based on the session ID. A value I know will remain constant throughout the game. Thus, each circle was random, but consistent among all users, all the time.

4. Testing network features

Near the second half of this project, I began working on implementing networking into the app. However, I realized it was extremely difficult to test as both the app code and the server code had to be on the same page to even test certain features. For example, if one made an update to the GET request format, then the other had to modify their code.

Team Effectiveness

I think our team worked very well together. Everyone did what was asked of them, and we were all aware of each other's progress at all time.

One improvement I can think of is, we should have done better prioritizing, and started integration testing a little earlier. We focused a lot on hardware at the beginning of the project, and we ignored the server, which was fine at the beginning. However, once we started reaching the end of the project we realized that the server had a lot of catch up to do, and since everything is connected to the server, we ended up losing a lot of testing time as the server played catch up.

Other Comments

-How did you ensure that your tasks would integrate with your team?

I was able to test the Wi-Fi RSSI reading early on by downloading a test version of the app (that required no networking) to test this feature.

Once I knew that worked, I mainly focused on working with the server. I would constantly keep an eye on the server code in Github, and update accordingly, then test on my computer with an emulator to make sure that my code was up to date.

- What hurdles did you have to overcome?

The beginning was the most difficult for me, since I had almost completely forgotten how to code in Java, and I had never done app development before. So, I would mess around with Android Studios in my free time, looking at example code, and eventually I started picking it up myself.

- What did you do to ensure success, or at least improve the likelihood of success?

Every time an update was made to the server, or I added a new feature to the app, I would go through my list of working features, to make sure that they were all still working as intended. This helped me catch a lot of bugs the moment they happened instead of figuring it out late in the project and panicking. I also had plenty of error checking code, especially for my GET and POST requests and that came in very handy during integration testing.

- What did you learn?

In terms of technical skills, I learned a lot about app development on Android Studios. But, more importantly, I learned a lot about working in a team and integration. It was new to me to have such disconnected tasks from one another. Because my app was only connected to the server,

sometimes I would have long periods where I just worked on my own, especially in the beginning when we did not start integration. I realized later on how important it is to keep up to date with everyone, and always have a plan to start integration, even if not everyone is ready yet.

- Were there any team dynamic issues?

The only issue was the fact that some of our tasks were very disconnected. So, sometimes we might end up all working separately and forget to communicate, but we would always meet up in labs and discuss our progress in the end.

References

1) https://stackoverflow.com/questions/43039149/how-to-ask-gps-permission-and-after-finish-show-map (Permissions Code)
2) https://developer.android.com/reference/android/location/LocationListener (LocationListener Docs)
3) https://developer.android.com/reference/android/location/LocationManager (LocationManager Docs)
4) https://www.youtube.com/watch?v=4JGvDUlfk7Y&list=PLrnPJCHvNZuCbuD3xpfKzQWOj3AXybSaM&ab_channel=CodinginFlow (Retrofit tutorial)
5) https://stackoverflow.com/questions/639695/how-to-convert-latitude-or-longitude-to-meters (meter offset to latitude longitude code)
6) https://developers.google.com/maps (Google Maps Developer page)
7) https://developer.android.com/guide/topics/connectivity/wifi-scan (Wi-Fi Scan Docs)
8) https://stackoverflow.com/questions/13950338/how-to-make-an-android-device-vibrate-with-different-frequency (Vibrating Code)
9) https://stackoverflow.com/questions/13398104/android-how-to-put-a-delay-after-a-button-is-pushed (Adapted Button Cooldown Code)
10) https://developers.google.com/maps/documentation/android-sdk/reference/com/google/android/libraries/maps/model/Circle (Circle Class Doc)