Logo

Description automatically generated

Team 63 – Mobile Attendance Application

Brandon atwal | owais ashraf | anthony j chen | jacob c danel

# Summary of Hours Worked on Attendance App, per Team Member:

Brandon Atwal:

* About 50 hours related to working on code: Primarily Bluetooth and multithreading, but some UI too.
* About 25 hours in communication and meetings with the team, Prof. Cole, and lectures with Dr. Razo
* About 8 hours working on documentation alone (time spent working on documentation with group is counted in the meetings hours).

Owais Ashraf:

* About 25 hours spent on UI design (research, testing and implementation)
* about 10 hours in overall android studios testing and configurations
* About 10 hours in administrative meeting

Anthony J Chen:

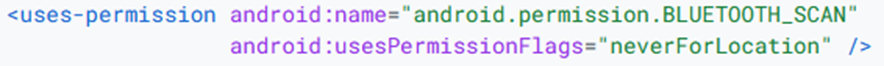
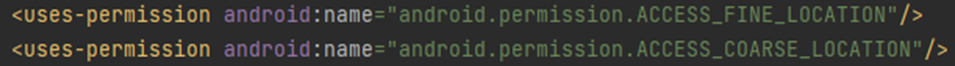
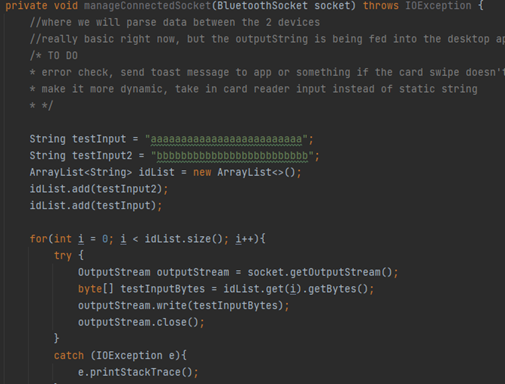
* About 50 hours related to working on the code: Primarily Multithreading and Bluetooth
* About 15 hours in communication consisting of meetings, lectures, and picking up an android device from Owais so I could properly debug.
* About 3 hours for documentation

Jacob C Danel:

* About 20-25 hours related to document creation/maintaining, setting up our teams’ channel for logging time, repository for code, etc.
* About 55-60 hours working on Bluetooth code.
* About 10 hours with meetings with team, Prof. Cole, lectures with Dr. Razo

## Application Development issues, Managerial Issues, How we overcame them:

Brandon Atwal:

* Application wasn’t working for some older versions of Android OS.
  + During one of the earlier versions of the mobile app, Jacob and I ran into an issue where the app was in a working state for his phone but not for mine. I suggested it likely had to do with the fact that my phone was running an older version of the Android OS than I was, he was on Android OS 10 while I was on Android OS 11. I believed this because the app’s target SDK was 33, and Jacob’s phone was running a much newer version of Android than my phone. I did more research on API versions and found that the Android Bluetooth documentation states that in Android 12 (API 31) the Bluetooth permission requirements changed. Since Jacob had set the Bluetooth permissions to work with his phone that was above API level 31, it worked properly on his but not on mine (which was running API level 29). Through further research I found that to resolve the issue we could simply change the permissions such that they work for both older and newer versions of Android. We removed the permissions Jacob had set up: And replaced it with the following permissions:This made it such that the app can access Bluetooth and scan for devices on Android devices of any API level.
* Intermittent IndexOutOfBounds error.
  + In an early version of the app, we had designed the interface such that the user first must press a button to search for devices, then is shown a list of Bluetooth devices that are detected, then the user taps on the device they want to connect to. We had the connection working and the list view working properly. However, on some occasions when pressing the discover button, an IndexOutOfBounds error would be thrown causing the app to fail.   
    I attempted to debug this issue. I decided to first attempt rubber duck debugging, as it is an effective way to find logical errors in code. I tried rubber duck debugging, but the code seemed as if it should not give any errors. I was unable to find the bug using rubber duck debugging, as the logic of the code was correct. I then attempted to use the debugger. I set breakpoints in the code and stepped through the code. I paid special attention to the contents of the two ArrayLists used in the code: deviceList and deviceListStrings. The deviceList was an ArrayList of type BluetoothDevice and deviceListStrings was an ArrayList of Strings corresponding to those Bluetooth devices. I eventually found that the issue was that when the user presses the search button (which starts/restarts bluetooth discovery), deviceList would be cleared but deviceListStrings would not. The two arrays needed to be cleared when the button is pressed a second time so that discovery restarts properly. Since deviceList was cleared but not deviceListStrings, deviceListStrings would sometimes have more elements than deviceList. When the user tapped on a UI list item (which populated based on the device strings), the app would try to connect to an item with an element ID that was out of bounds for the deviceList, resulting in it throwing an IndexOutOfBounds error. The fact that deviceList was cleared but deviceListStrings was not cleared was likely an oversight made by the person who initially wrote that section of code. By having both lists clear at the same time, I was able to resolve the bug. During a later meeting, Professor Cole clarified that he did not want the user to have to manually select a device to connect to. He wanted our mobile app to automatically connect to Attend without user input. Thus, deviceListStrings and the device selection UI had to be removed. However, this was a valuable experience for me in using the debugger which will help me in future projects.
* A bug in which the app could not send more than one ID to the desktop app.
  + Jacob had reworked our mobile app’s Bluetooth functionality such that it would automatically connect to the desktop app over Bluetooth and send an ID to it. However, he ran into an issue where he could not get it to send more than one ID to the desktop app. I downloaded his code and tested it. I saw that the mobile app would send the first input successfully, but the second input would not be sent. On first inspection, the code seemed as if it should work. However, at the time I did not have a strong understanding of how the Bluetooth sockets worked, so I decided to do more research. I spent a few hours researching sockets so that I understood what his code was actually doing, and eventually figured out what was causing the issue. Here is a snippet from Jacob’s code: In this code, he is attempting to send two test inputs over the Bluetooth socket sequentially. For every string, it gets the output stream from the Bluetooth socket, writes the string to the output stream as bytes, then closes the output stream. However, closing the output stream essentially terminates the connection: according to Java documentation, java.io.OutputStream.close() closes the stream and releases any system resources associated with it. A closed stream cannot perform output operations and cannot be reopened. Thus, the code sends the first ID, closes the stream, then attempts to send the next ID, but cannot since the stream is closed. The solution I found was to flush the stream instead of closing it. Flushing a stream forces any buffered output bytes to be written out. In other words, it would wait for all the data to be sent.   
    This caused the app to be able to properly send as many messages as desired to Attend via Bluetooth. However, later I would find out that the docs also says that java.io.OutputStream.flush() does nothing, so the entire statement can be removed. Removing the flush statement does not improve efficiency, but it does remove an extraneous line of code.

* Managerial issue: a group member missed two meetings in a row.
  + One of our group members, Anthony, missed two meetings in a row. He did not give any notice that he would miss the meetings. After the meetings, he would state that he overslept. We were concerned that this may happen again and did not want this to become a regular occurrence. Since I am the team leader, I lead the effort. I set up an unofficial call with Anthony through Discord. The other team members joined, and we spoke with Anthony. We realized there was a possibility Anthony may be having other issues that were causing him to miss meetings, either in his personal life or education. Thus, we approached the subject carefully, and when we asked him, he said that it was simply that his sleep schedule had deteriorated due to other schoolwork, which caused him to oversleep. We let him know he is a valuable member of the team and that missing more meetings could cause issues for the entire project. He promised not to miss any more meetings and has not missed or been late to any meetings since.

* Retrieving bytes from the Attend desktop app over Bluetooth
  + We needed to make it so that the app can receive bytes from Attend after the mobile app sends \*ID\* to Attend. The bytes would be an array of student IDs. As Professor Cole stated, this would be acceptable from a privacy standpoint since they are not associated with any names or other personal information. We already had the sending messages to Attend functionality finished, so all we needed to do was make our app also listen to Bluetooth for bytes sent from Attend. Anthony and I worked on it together using in-person pair programming. We realized the best time to get the list of IDs would be as soon as the app establishes Bluetooth connection with Attend. We modified the logic of our thread that handles establishing Bluetooth connection with Attend, called ConnectThread. We made it so that once the socket is connected, the app sends \*ID\* to Attend. We then had it wait for 1.5 seconds (Thread.sleep) to ensure the desktop app had time to send a response. We then access the socket’s InputStream and read bytes off there until there are none left. At the time, Attend did not send an asterisk to denote the end of output. Thus, there was no way to know when Attend was finished sending the IDs. Thus, using a long 1.5 second wait was a crude, temporary way to ensure Attend had sent all the bytes. Once Prof. Cole modified Attend to send an asterisk to denote the end of the output, I removed the wait time. With the wait removed, I had it simply read indefinitely until an asterisk was received. After ConnectThread finishes getting all bytes from the Input Stream, we converted the bytes into an ArrayList of strings. We used a synchronized list so that the strings would also be accessible from other threads. This is because we knew that later we would need another thread to check if a swiped card’s data contains an ID from the list of ID strings. Anthony and I worked well together while pair programming. When one of us would get stuck, the other would often fill the gap. This made our work far more efficient than if one of us had done it alone.

Owais Ashraf:

* Figuring out the second activity screen
  + Initially the design idea we had placed was to have an initial screen for the Bluetooth connect then have a secondary screen specifically for the desktop app itself, meaning the first screen would only show the Bluetooth connections in terms of what Bluetooth connections are available and what those device names were and then the second screen would show the actual implementation of the attendance app meaning the input box for the card swipe ID, as well as the final toast message that would appear in terms of the acceptance or failure of the card, swipe. The issue that arose was it was pretty difficult to figure out a way for the Bluetooth screen to automatically go forward and find the second activity screen as soon as a Bluetooth connection was established but eventually, we had decided to drop that idea of the design for the app and just make it one screen that would have the Bluetooth connection on the top and then after that it would have the attendance app information underneath
* Progress report issue
  + Since we had all our progress reports in different sheets on different files it was difficult for us to figure out a way of compiling them all into one I was put in charge of this task and accordingly, I put it all into one master file that would have all the progress reports together and then after that, I ran a formula for all of the files to add up into a final table summary in the beginning. Unfortunately, because of some formatting issues in one of the files, the final amount was coming up incorrect for half of the team hence after I looked through the files individually and figured out exactly which file that issue was occurring in I found the solution and was able to deal with that formatting issue. This specific issue was that one of these cells was being tripled meaning that the cell itself was showing the place of three cells just for one person. When in reality it should have just been oneself for that one person specifically. After correcting that format and running the formula again it was showing the correct amount for the final hours

Anthony J Chen:

* Working on Connect Thread with Brandon
  + Brandon has a writeup of this in his section
* Looking at the code without a device that could run it
  + Up until I started writing code, I didn’t have a device that could run the app as it was intended. This caused me some serious issues as the only thing I could offer the team at this point was my insight into how I thought the code should run, instead of the more valuable insight into how the code was running. Eventually, I was able to borrow an Android phone from Owais that allowed me to contribute more significantly to the team.
* Working to multithread ID submissions
  + The validation of card input to the list that was sent from Attend needed to be multithreaded so that the main UI thread wasn’t taken up by the validation code and could quickly accept new card inputs to not slow down the attendance process. However, I ran into several issues here including the fact that since I initially had a misunderstanding as to how Bluetooth Sockets worked, I was creating a new one each time a created a new Check Thread which caused the card inputs to not be sent over to Attend. After a large amount of digging, I finally realized this was the issue and ripped out the Socket creation code and fixed that particular issue.
* Adding auto-submit card swipe functionality
  + There was a minor issue that the field would not refocus after a card swipe to be ready for another card swipe. However, after a quick Google, that problem was sorted fairly quickly.
  + The main issue here was that I anticipated that the card reader would always read the card correctly, I did not expect the card reader to only read the whole card information sometimes. So, I had to implement an auto-submit function which submitted what card information we were given after one second of no change in the input field. There was a problem with this however, as the fix I had for the minor issue above was causing the afterTextChangedWatcher that I put on the input field to trigger, which then caused an infinite loop. So, because of this I had to implement a gate called refocusCalled that nulled the first call of afterTextChanged in the Watcher after the input field was refocused.
  + There was also a problem where the field would continually submit after a card swipe was submitted. This was fixed by resetting the start time to zero every time a card was submitted which was used as a flag to disable the timer between card swipes.

Jacob C Danel:

* Coming up with an initial plan for implementing Bluetooth.
  + Figuring out how to tackle setting up the Bluetooth functionality was fairly daunting at first. Most of my attempts at implementing some form of Bluetooth functionality, mainly video tutorials, were unsuccessful largely due to my unfamiliarity with android development. I was able to understand and implement the java aspects fairly easily but a lot of my issues early on stemmed from not understanding how to implement the UI alongside the java code. Particularly how the UI components interacted with one another and how to write the code for it in the java code, manifest, and other xml files. Once I better understood how those various files (.java, activity\_main.xml, the AndroidManifest, etc.) all interacted with one another it became somewhat easier. It was a lot of trial and error figuring out how to deal with the errors being thrown by the compiler because one part of an .xml file didn’t match properly with the java file, and so on.
* Figuring out how to scan for nearby devices.
  + The next issue that I ran into was figuring out how to properly retrieve and parse through the nearby, Bluetooth enabled, devices. Part of my issue with this early on was my Bluetooth dongle that I was using was actually defective. So, for a large chunk of time, I assumed it had to do with my code and kept making changes and trying to fix the problem there. I eventually figured out that it wasn’t (entirely) the code that was the issue when I tried connecting and pairing my phone to my computer through my phone’s native Bluetooth settings. Once I had realized this and got a working Bluetooth dongle, I reattempted the discovery for nearby devices and found that the broadcast receiver wasn’t properly populating my device list in the UI. What I ended up doing to solve this was setting the broadcast receiver up as a global variable which could then be used in other functions to better parse through the nearby devices and populate a device list that can then be used to connect to a chosen device.
* Implementing list of nearby devices and being able to connect to one by tapping on it.
  + Continuing on from my previous issue, once I had properly implemented the list, I had to set it up so that when one of the devices from the list was tapped on in the app it would connect to it using its MAC address. This was fairly easily to set up as all I needed to do was ensure that the chosen device contained the matching uuid (the uuid associated with the desktop app) and then connect to said device by using the BluetoothDevice class’s function ‘createRfcommSockettoServiceRecord’. The issue I had with this was the way I had initially coded the app to make the connection. It was somewhat unreliable with making the connection to the desktop attendance app so in early versions there was the need to frequently restart both apps to make the connection. This was later solved with a rework of how the connection was made to the app (via getting the device from a list of paired devices and checking each one until we found the device with the uuid for the desktop app).
* Figuring out why the application wasn’t working for some older versions of Android OS.
  + During one of the earlier versions of the mobile app, Brandon and I ran into an issue where the app was in a working state for my phone but not for his. He suggested it likely had to do with the fact that he was running an older version of the Android OS than I was, he was on Android OS 10 while I was on Android OS 11. The strange thing was that in the Android Manifest I knew that it had a parameter set to check for a minimum OS version (more specifically the SDK). That minimum was set to ensure the device running the app had an SDK of at minimum 24, so it should have worked for any devices running Android OS \* and up. However, Brandon eventually figured out it had to do with a permission parameter in the Android Manifest that was causing the issue. To allow for Bluetooth discovery I had put in the permission, Because I misunderstood the documentation for this, my understanding was that you needed to provide this specific parameter for the Bluetooth permissions so that it the app would never track an individual’s location (because that is unnecessary for this app and goes against Prof. Cole’s desire to avoid breaching privacy in any manner with the app). This was needed however to be able to scan for nearby devices, as we needed a starting point to begin the discovery process. So, it was changed to instead have the permissions below. The data for the location is never stored and/or sent anywhere so privacy concerns are a none issue.
* Figuring out how to connect to an unpaired device.
  + Figuring out how to implement this properly took a lot of reworking the permission checks in the java code. To save time writing this in every part of the java code that had to do with Bluetooth I wrote a function that did the permission check to avoid just copy and pasting the same code over and over and bloating up the program. Once the permission checks were properly set up all that was required for the app to prompt the user to pair the devices was to check if the selected device’s bond state was true. If it was then the program could proceed with making the connection. Otherwise, the app would prompt the user to pair the devices through their device’s on-screen prompts and then if said user confirmed and bonded the devices it would continue with the connection. If the user declined, then the app would simply close as there was nothing else for it to do pass that point.
* Reworking Bluetooth functionality to automatically connect to desired device.
  + After my team had one of our meetings with Prof. Cole, he told us that the connection needed to be made automatically on app start up and that the current iteration of the mobile app didn’t meet that requirement. As previously mentioned, it would make the connection to the desired app via a list of nearby devices with Bluetooth functionality and after the user had tapped the desired device from that list. So, the first idea on how to change this was to parse through the device list that the broadcast receiver generates and check each one until I found the device that had the desired uuid. The issue here was with how I set up my broadcast receiver. I was unable to determine when the list was actually populated. Rather, assuming I properly understand how the broadcast receiver works, it wouldn’t populate the list as quickly as I thought it did so I couldn’t have the initial device connection code in the OnCreate function like I previously had. I think that had only worked before because I also had an onClickListener set up in the OnCreate function that would allow the app to return to that function once one of the device’s was tapped on from the list while the app was running. I was unable to find what sort of On”something” function would work to achieve the same effect. What I ended up doing was to parse through a list of already paired devices and make the connection from the paired device that had the desired uuid. I figured it was safe to assume that the two devices would already be paired prior to running the mobile app. This was the only solution I could come up with to achieve the functionality that Prof. Cole desired.