Sabotage
Individual Report

Henry Huang

83420661

Team 11

Contents

Individual Contributions

General Hardware Interfacing

I worked with Moiz to get a baseline hardware configuration working. We spent significant time experimenting with the GPS module and scrapped the Bluetooth module, trying different methods of interfacing the De1-SoC with them. After I discovered and tested the Nios II's hardware abstraction layer, I considered this task complete.

WiFi Module

I worked on the ESP8266 WiFi module of Sabotage on my own. This module serves two functions.

1. Take data gathered by the De1-SoC and its other connected hardware modules, parse it from serial input, format it as a JSON string, and send it to our backend server via an HTTP request.
2. Act as a WiFi access point, which allows our mobile app to ping the module based on its MAC address, read the signal strength, and thus determine proximity.

To meet these objectives, I developed conventions to communicate between the De1-SoC and the ESP8266. I also defined the guidelines to communicate with our remote server.

Backend Server and Database

I worked on the backend server and database of our project on my own. Because our project is supposed to be scalable for an arbitrary number of De1-SoCs, each associated with any number of phones running our mobile app, we needed a way to communicate and coordinate all systems and data. Hence, I implemented a server capable of processing and relaying all necessary data between De1-SoC game sessions and user phones. To make it scalable for an arbitrary number of devices, I interface the server with a database for long-term, persistent data storage.

Maintaining data consistency across an arbitrary number of clients proved challenging. We would decide on a structure for our data to be stored and retrieved but often found that implementing additional features required changing the fundamental structure of the data, which made rewriting code a common occurrence. For example, sabotage actions on the server were implemented later in the development cycle, and I found that the JSON data structures were

designed such that the requirements for sabotage action could not be fulfilled in a cleanly. Consequently, I restructured and rewrote large sections of the backend to make the addition possible. Despite my attempts to prepare, adding new functionalities to the server proved difficult.

Detailed Design

The figures in this section are detailed visual aids for understanding each component in isolation. For a complete overview of where these components fit in the overall system, refer to Figure A in the Shared Appendix.

WiFi Module

Since the ESP8266 MCU is supported by Arduino, I implemented all functionality for this component within the Arduino framework. Along with the standard Arduino library, I used the ESP8266WiFi, ESP8266HTTPClient, and WiFiClient libraries in my source code.

Function 1: Data Transmission

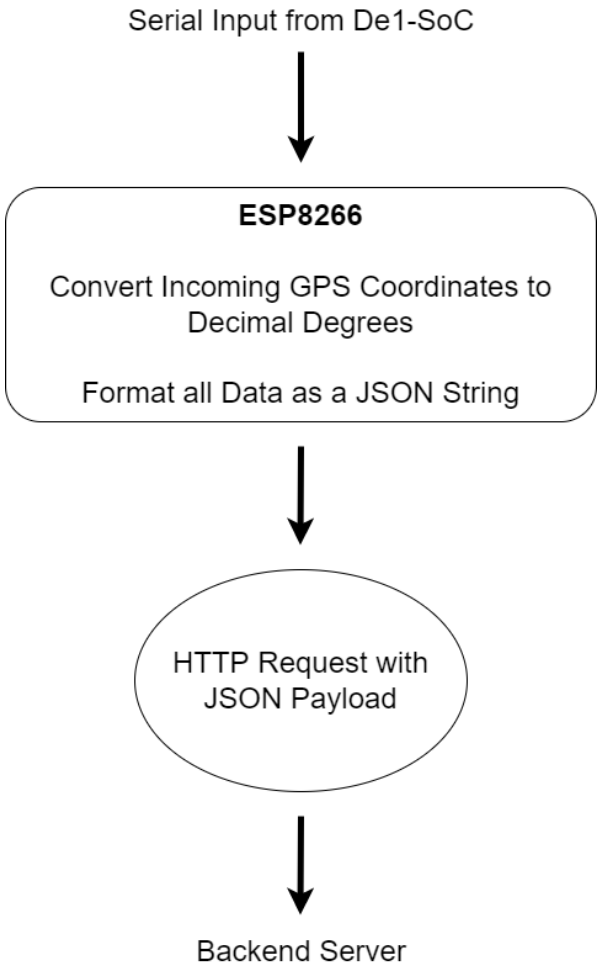

Figure 1: Function 1 Flow Diagram

First, it receives data from the De1-SoC via the RX pin on the ESP8266. We configured the baud rate to be 9600, but it can be modified to work with other baud rates like 115200 if needed. I defined the following convention for the data to be received over serial:

$<Session ID>, <Latitude>, <N/S>, <Longitude>, <E/W>, <NumPlayers>, <Win>*

1. The '$' character denotes the start of our data chunk, ',' is the delimiter between fields, '*' marks the end of our data chunk, and everything else is a field.
2. Session ID is a unique integer string identifier for the connected De1-SoC.
3. The <N/S> and <E/W> are single-character fields used to convert our latitude and longitude inputs to the Decimal Degrees format. The final latitude and longitude we send off to the server depend on whether we receive (N)orth or (S)outh, and (E)ast or (W)est.

4. NumPlayers indicate the number of players active around the De1-SoC.
5. Win is a four-digit integer. This is the user ID that a player enters on the touch screen connected to the De1-SoC when they find it.

The NumPlayers field is not used. The original intent was to track the number of players in a session from the De1-SoC, but later on, I found that it made more sense to track the number of players based on the mobile app. See Function 2: WiFi Access Point for why. The NumPlayers field was left in as a contingency for if we needed to use it again, and because its overhead is marginal.

After the data is received and processed, it is formatted into a JSON string. Finally, I use the ESP8266HTTPClient library to send a POST request to our backend server with the JSON formatted data as its payload.

This function is easy to extend as each field is a string that is processed in the same way. Hence, adding new ones only requires that they be defined in the JSON template and handled in the serial reader.

Function 2: WiFi Access Point

By configuring the ESP8266 as a WiFi access point, the mobile app can determine a player's proximity to the De1-SoC. This was simple to set up on the ESP8266. I primarily needed to extract the MAC address while the Arduino libraries handle the rest. Most of the work for this datapath is done from the mobile app.

The ease at which we can extract the MAC address from the ESP8266 made the need for the NumPlayers field detailed in Function 1: Data Transmission, unnecessary. Originally, we tracked this metric based on the number of player phones detected from the De1-SoC. This would have required filtering for an arbitrary number of MAC addresses, needing the MAC address of each player's phone to be relayed to the De1-SoC. This is significantly more complex than our current solution, where each phone scans for the MAC address of a single De1-SoC, and the number of players is tracked on the server. For more detail on how this is implemented, see the POST /app Implementation section under the Backend Server and Database Detailed Design section.

Backend Server and Database

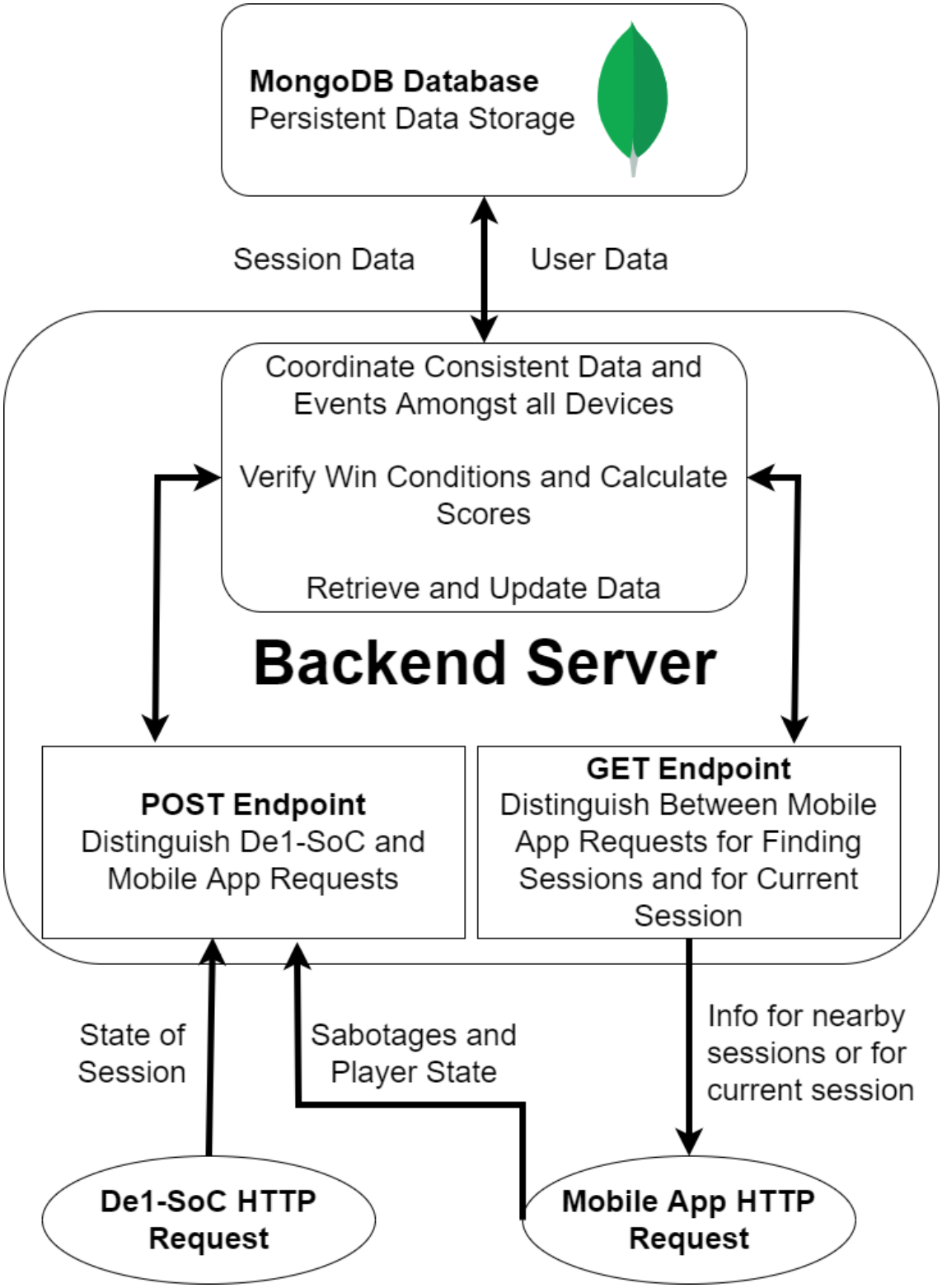

Figure 2: Backend Flow Diagram

Architecture

The two main libraries I used to implement this component were HTTPServer and Pymongo. HTTPServer is a standard Python library while Pymongo is not. Pymongo is the Python interface for MongoDB, a NoSQL database service that I used to manage all persistent data for the Sabotage games. The only other non-standard Python library I used was Requests, which was only used for testing the server. I hosted the server locally by using an application called ngrok to make my localhost accessible to the public internet. Hosting in this manner sped up development and testing as I did not have to configure the server for a specific service.

When an HTTP POST request is received, the server checks if it is routed to "/de1" for the De1-SoC or "/app" for the mobile app. When an HTTP GET request is received, it distinguishes between requests routed to "/sessions" for getting information on nearby sessions and "/lobby" if the request is for a player's current game session.

Implementation

This section details how data is processed on the server and is very specific to how the data structures are designed, referencing specific fields in each data structure. My Secondary Appendix displays the POST /de1, POST /app, GET /sessions, and GET /lobby request and response data structures I made in a standard JSON format. If the descriptions in this section get confusing, refer to my Secondary Appendix as a guide.

Except for GET /sessions, each request includes a "SessionID" field in the request parameters and will respond with error 400 if invalid. Similarly, aside from POST /de1, each request has a "UserID" field that also returns error 400 if invalid.

On receiving a POST /de1 request, it takes the coordinates of the De1-SoC from the payload and checks if it changed from the stored location in the database. If it changes and the data is non-zero, then its coordinates are updated in the database and a "Moved" flag is set to True in the database. This "Moved" flag serves to alert an administrator in case of theft or disruption of the De1-SoC, requiring manual oversight to resolve. If the coordinates changed but the result is (0, 0), this means the GPS module was unable to ascertain its location. In this case, I set it so the last known location of the De1-SoC on the database is used. This allows Sabotage game sessions to be easily moved indoors if needed. If the payload contains the "Win" field, then the server verifies if the user ID contained in this field is valid. If the user ID is not registered or the

database's record of the user ID's current game session does not match the "SessionID" field, then the HTTP response returns successfully with a warning, doing nothing to the state of the server and database. But if the user is verified to be allowed to win, then their score is calculated based on how long it took to find the De1-SoC since their game began in addition to how many other players they faced in the session. This score is placed on the session-specific leaderboard with the winner's user ID. The winner is also rewarded with sabotage tokens to be used against other players in later games. Lastly, a win cooldown is set specific to the winner and the session they won, preventing them from re-entering their win on the De1-SoC same De1-SoC for a time.

On receiving a POST /app request, using the "UserID" field in the payload, the requesting player's current session is retrieved from the database. If the "SessionID" field of the payload differs from the current session registered in the database, the user is removed from the old session and is placed into the new one specified in the payload. However, to prevent players from starting games from an unreasonable distance away, error 400 is returned if this new session is not near the user. Once the session is updated, the current time is set for the user in the database, which serves as the start time for score calculations. This route also processes sabotage attempts. The server checks if the user has available tokens to perform a sabotage action, deducting a token if so and doing nothing if not. Then, the server sets this sabotage as active in the session specified by the payload. The sabotage is active for a set duration and is asynchronously removed when finished via a timeout. Lastly, since this request pertains to when a user joins or leaves a game session, session-specific user metrics are tracked based on these requests. For example, the number of players in a session changes based on when users send POST /app requests that join or leave sessions.

When a user starts the app, the first action they take is to assign themselves a user ID and to find nearby game sessions. These are handled on the server via the GET /sessions request. If the mobile app sends a GET /sessions request with a user ID of 0, then the server interprets that to mean it is a new user. Hence, it generates a randomly selected and unique user ID for that user.

The purpose of the GET /lobby request is for the mobile app to poll the current state of the player's game session. Using the "SessionID" parameter of the request, the server retrieves sabotages active in the session. Additionally, it tells the mobile app when the player wins the game. Lastly, it retrieves the leaderboard for the session.

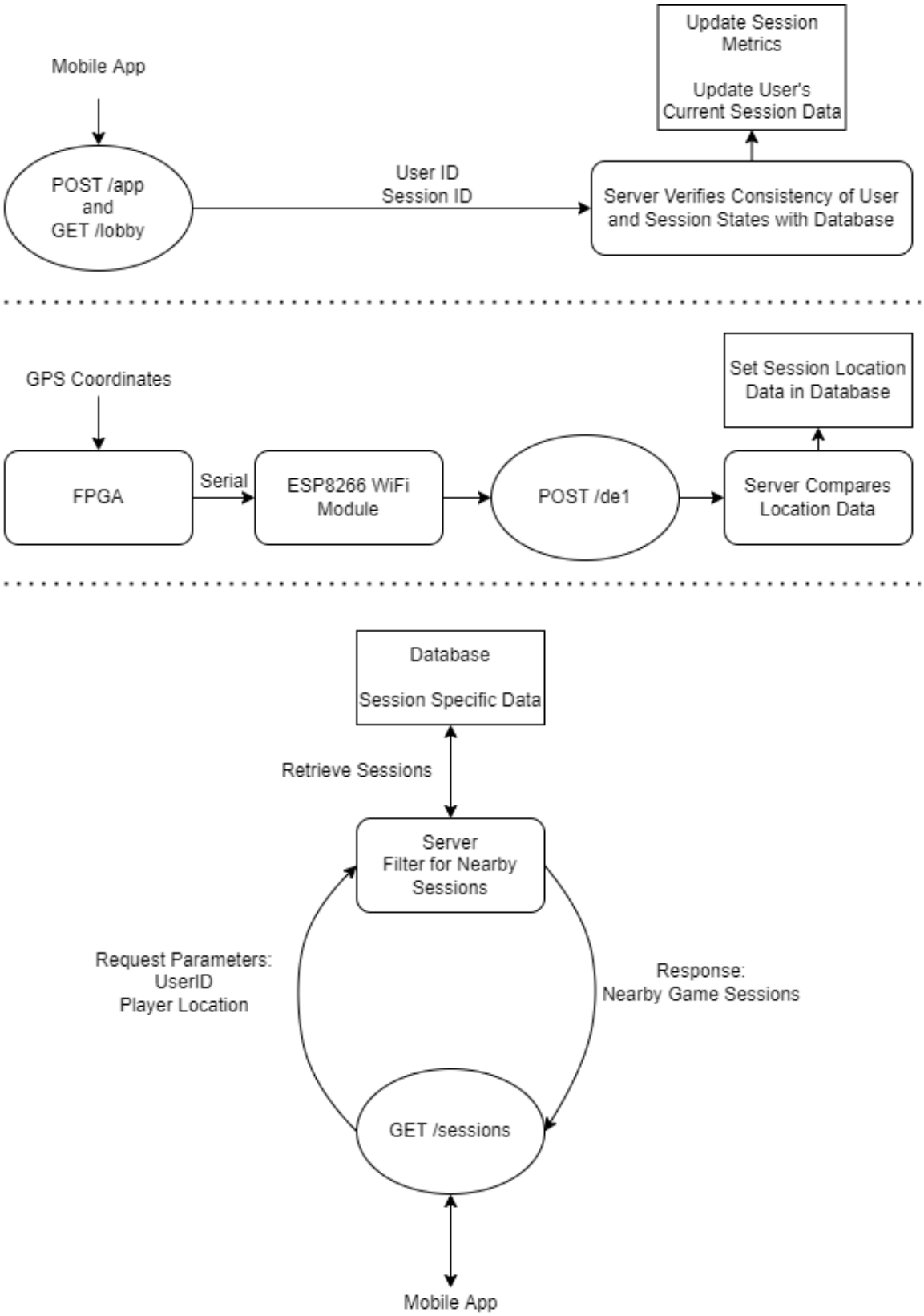

Figure 3: Get /sessions Datapath Flow Diagram

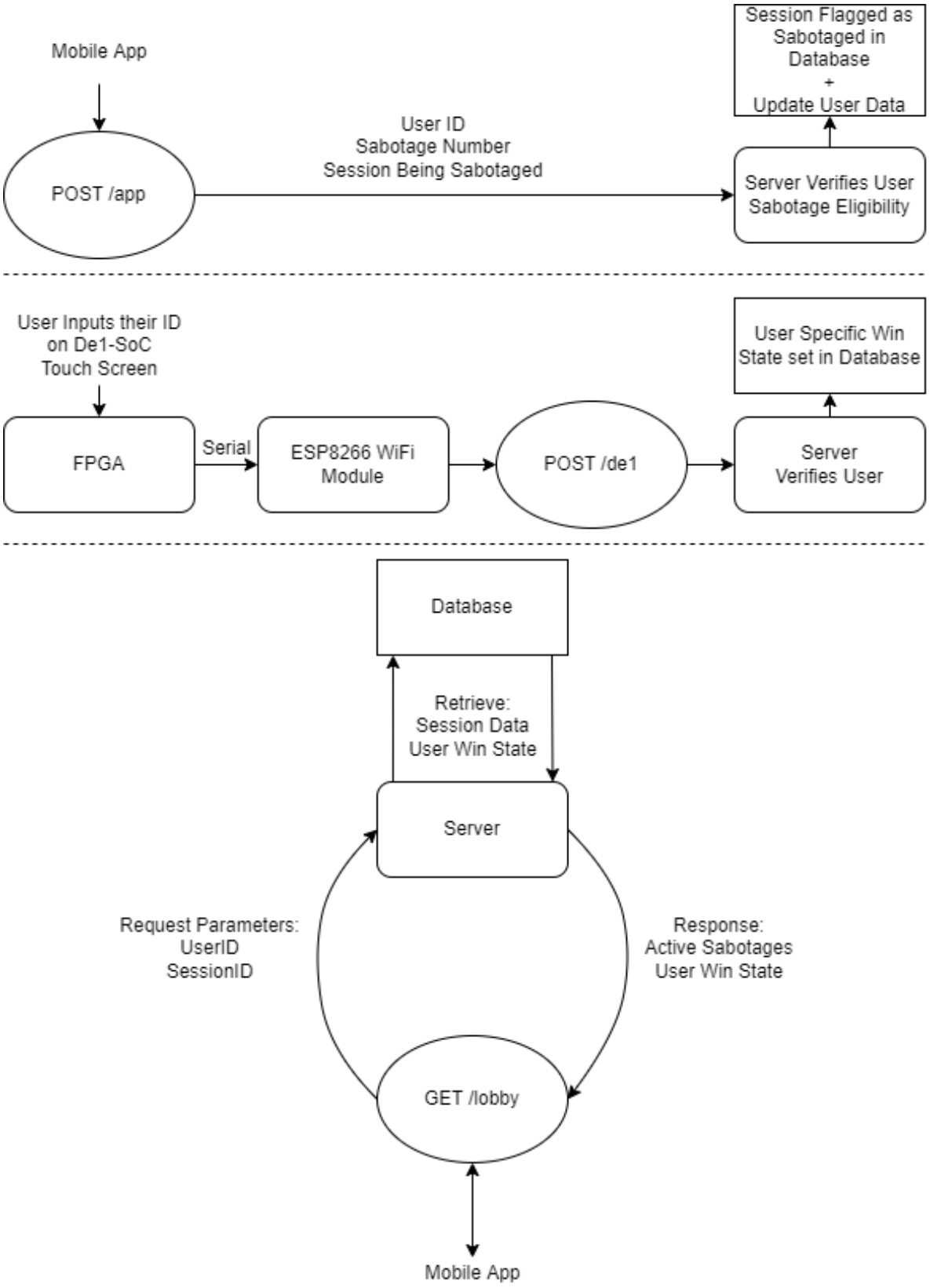

Figure 4: GET /lobby Datapath Flow Diagram

Extensibility

From the backend perspective, different sabotage actions are simply integers. The specific effect of the sabotages are implemented in the mobile app. Consequently, adding new sabotage action variants requires no additional work for the backend.

Although adding new features to the backend can be difficult, I developed a testing framework in Testing/Server/ to facilitate additions. Clearing and filling test database clusters with data only needs a few lines in InitDB.py and Test_Data.py to be changed. Simply add new data to Test_Data.py and set it in InitDB.py. Now whenever the Test_Client.py is run, the new data is ready to be used. Additionally, Test_Client.py tests all original functionalities, making it easier to catch new bugs for older features.

Since the backend is implemented without a specific hosting service, it does not use any platform-dependent code and can therefore be adapted for any hosting service. Alternatively, anybody could host it locally for their enjoyment or testing.

From the beginning, we planned to gather metrics on how each session is performing. We believe that informing the players of details such as how many players are in each session allows for more strategic gameplay, as it can greatly impact scores. Adding additional metrics to the backend is simple. Because MongoDB is a NoSQL database, new metrics can be dynamically added to each session independently. Different sessions can also have different metrics fields as the backend code always retrieves every metric for a session when the mobile app requests them. Hence, setting metrics requires little extra overhead while retrieving them requires no additional work.

Challenges

When starting the project, I anticipated that the hardware needed a lot of initial work to get started. Hence, I began the project focused primarily on getting a general framework for the hardware working. This meant my attention was divided between hardware and the server. Initially, we did not plan on using the Nios II at all and wanted to connect all our hardware to the Arm HPS instead. I spent a significant amount of time learning how to connect peripherals from the FPGA to the HPS and was successful. However, this proved to be a lot more work than simply using the Nios II to control our peripherals, so we scrapped the Linux component with

one exception. Our Linux image already had a VGA driver implemented, so I assumed it would be easier to use it than implement our own. I designed a protocol to coordinate between the touch screen controlled by the Nios II and the VGA display controlled by the HPS, but when I tried to test it, I found that programming the Nios II broke the Linux installation on the De1-SoC. After spending more time attempting to make both processors work together, I eventually gave up and had Moiz implement the VGA controller on the Nios II instead. Because my attention was divided between the backend and the hardware, scrapping all the work I did for the HPS meant a lot of time that could have been used for the backend was wasted. However, as the hardware was completed within a day of me completing the backend, my initial prediction that the hardware needed a lot more time to get started proved to be correct. Overall, seeing as I achieved all goals set out for the backend, I believe that I overcame the challenge of working on two completely different components and was able to recover from the time lost to scrapping a lot of work.

Team Effectiveness

Overall, I believe my team was able to work well together. There was very little friction between us throughout the development of Sabotage. Whenever we had conflicting ideas for Sabotage, we always came to a peaceful resolution. Unfortunately, we faced communication and organizational problems. We initially defined procedures for Github branches, issues, pull requests, and Kanban boards. However, after setting the guidelines, it quickly fell apart. Aside from branching conventions, I was the only one following the other procedures we set. I was also the only one updating READMEs as I worked. Consequently, tracking other members' progress was more tedious than it needed to be, especially when commits were often not pushed to Github. I also found that despite writing extensive documentation, my teammates did not always read it. There were instances when my teammates would inquire about one of my design decisions, where I responded by deferring to my documentation. Consequently, the effort I put into rigid documentation was not fully capitalized on. Preparing for integration was also challenging as reading source code that is always subject to change cannot replace concretely defined specifications.

Other Comments

Integration

To ensure my tasks would integrate well with my team, I consistently wrote concrete specifications and READMes for all my code, updating them whenever I made changes. By defining concrete specifications, I could abstract away most of the confusing details of my code for my teammates. Hence, we never needed to drastically restructure our code for tasks to fit together.

Ensuring Success

Early in the project, the team had a lot of ambitious ideas for Sabotage. Although a lot of good suggestions came up, I knew the time constraints we faced made most of them unfeasible. Hence, whenever a new idea was suggested and my team got invested, I always intervened and argued to limit the scope. I was adamant that we need to focus on the core features of Sabotage before spending time on extra, less vital ones. Seeing as we finished very close to the deadline, I believe I made the right decision.

References/Acknowledgements

WiFi Module

I used the following resources to configure the ESP8266 WiFi module. These sources include setup instructions and example code.

General setup:

https://learn.adafruit.com/adafruit-huzzah-esp8266-breakout/using-arduino-ide

Sending HTTP Requests:

https://randomnerdtutorials.com/esp8266-nodemcu-http-get-post-arduino/#http-post

Setting up a WiFi Access Point:

https://learn.sparkfun.com/tutorials/esp8266-thing-hookup-guide/example-sketch-ap-web-server#:~:text=Not%20only%20can%20the%20ESP8266,pages%20to%20any%20connected%20client

Backend Server and Database

The server is based on the Python HTTPServer library. The documentation is found here:

https://docs.python.org/3/library/http.server.html

In addition to the documentation, I used UBC Sailbot's server code, which I contributed to, as an example and reference: https://github.com/UBCSailbot/network-table/blob/master/projects/land_satellite_listener/land_satellite_listener.py

I used MongoDB to manage my database. I interfaced with MongoDB using their Pymongo Python API:

https://www.mongodb.com/

https://pymongo.readthedocs.io/en/stable/

I used the ngrok application to host my server on my local machine:

https://ngrok.com/