Ryerson University

Department of Electrical, Computer, & Biomedical Engineering
Faculty of Engineering & Architectural Science

Course Title:	Distributed Cloud Computing
Course Number:	COE892
Semester/Year (e.g.F2016)	W2024

Instructor:	Dr. Muhammad Jaseemuddin

Assignment/Lab Number:	Project Final Report
Assignment/Lab Title:	Enhancing Air Traffic Control Systems

Submission Date:	April 5th, 2024
Due Date:	April 5th, 2024

Student LAST Name	Student FIRST Name	Student Number	Section	Signature*
Muraleethasan	Saiharan	500970443	03	SM
Nagar	Ashreet	500969636	03	AN
Thanaseelan	Atheesh	500962758	01	AT

Introduction

This progress report provides an overview of the development journey of our air traffic control system prototype. Our team has been diligently working on designing and implementing a functional model that adeptly manages communication between the control center and simulated aircraft clients. The primary objective of this endeavor is to showcase the seamless coordination of simulated air traffic, focusing on the efficient execution of commands between the control center and aircraft clients. In this report, we will discuss the task breakdowns, milestones achieved, challenges encountered, and the timeline for upcoming tasks, as we strive towards creating a robust and effective air traffic management solution.

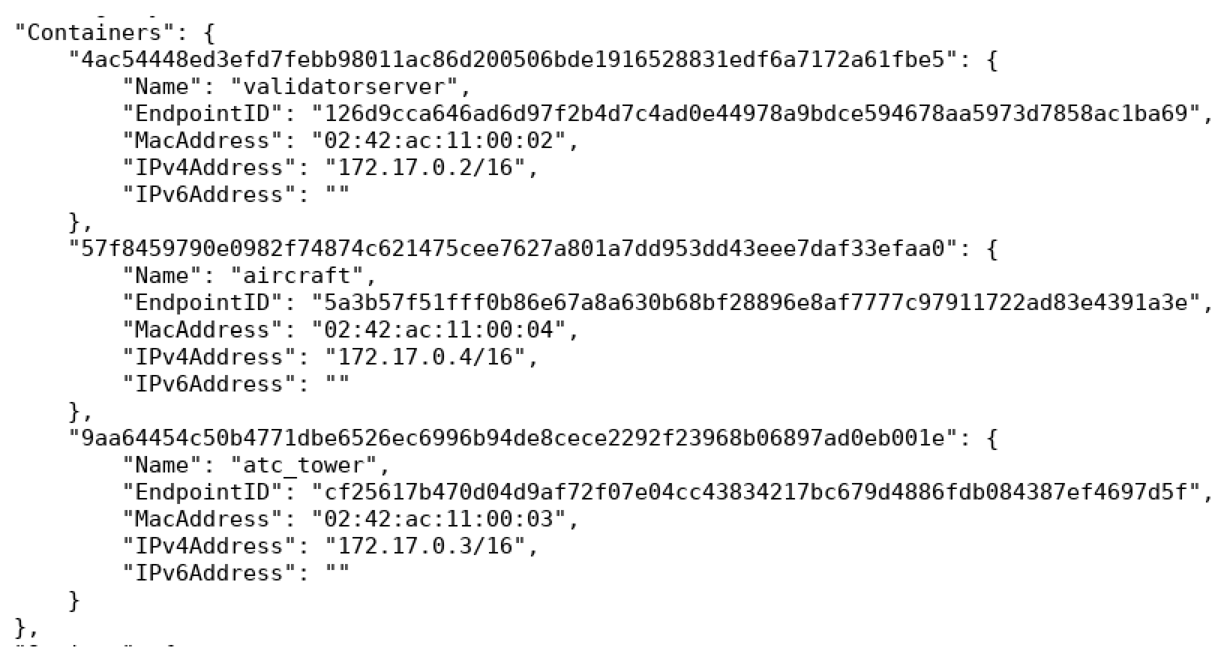
How to Run

This project comes with Dockerfiles, which simplify the deployment process of this project. First, each image must be built by using the docker build command (docker build -t *image_name* .) in the directories /validator, /atc, and /aircraft. Then, the images can be run using the following commands:

> docker run --rm -d -p 50051:50051 --name *validator_imagename validator_containername*
>
> docker run --rm -d -p 8000:8000 --name *atc_imagename atc_containername*
>
> docker run --rm -d -p 8001:*AC_PORT* --name *aircraft_imagename aircraft_containername*

Each aircraft will need a separate port number, which should be filled in on "*AC_PORT*". The port number can be any number, as long as it is not 50051, 8000, privileged (1-1023), or in use by another application.

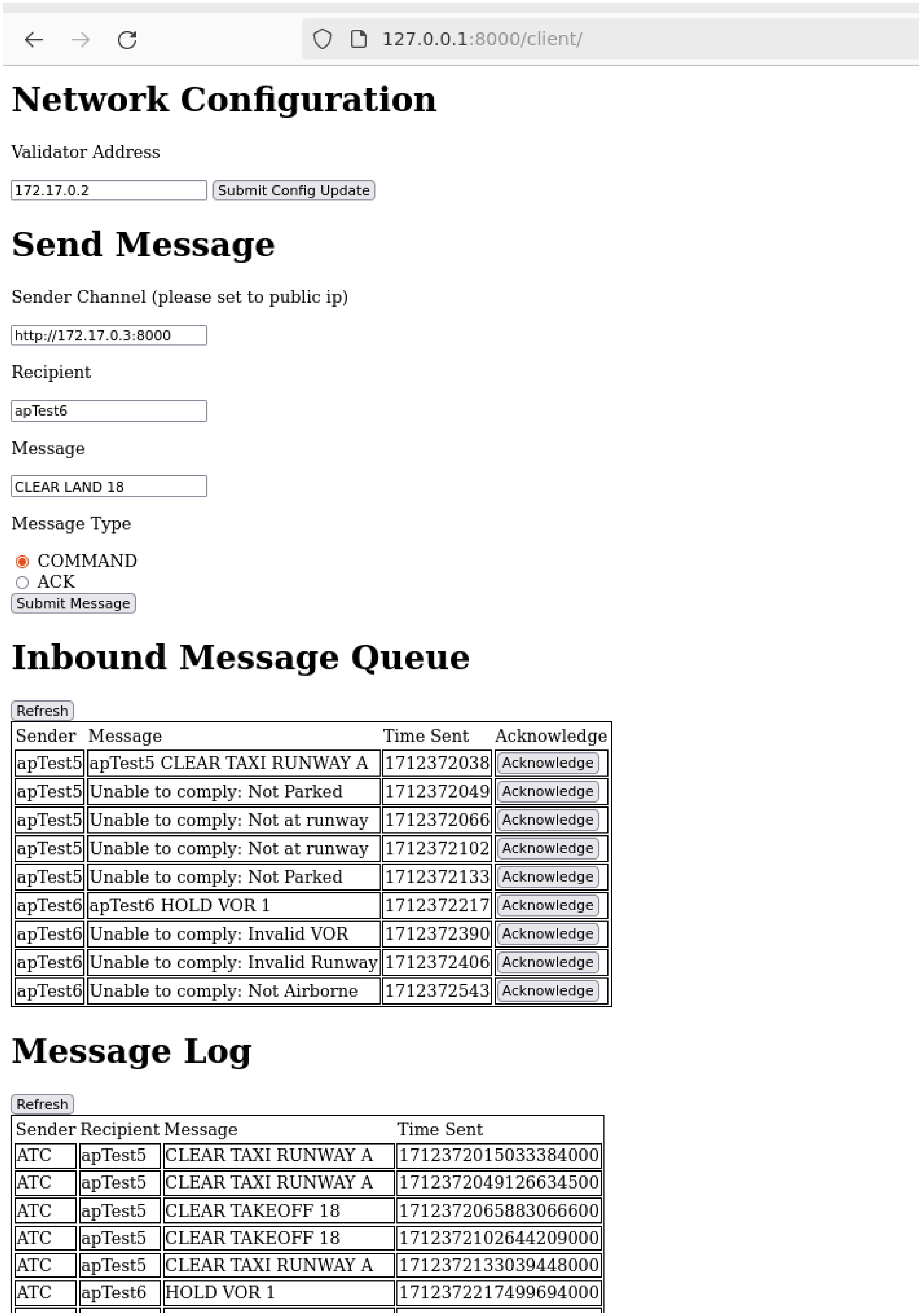
When running the containers on a local machine, the container IPs should also be found using "docker network inspect bridge".



```
"Containers": {
    "4ac54448ed3efd7febb98011ac86d200506bde1916528831edf6a7172a61fbe5": {
        "Name": "validatorserver",
        "EndpointID": "126d9cca646ad6d97f2b4d7c4ad0e44978a9bdce594678aa5973d7858ac1ba69",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
    },
    "57f8459790e0982f74874c621475cee7627a801a7dd953dd43eee7daf33efaa0": {
        "Name": "aircraft",
        "EndpointID": "5a3b57f51fff0b86e67a8a630b68bf28896e8af7777c97911722ad83e4391a3e",
        "MacAddress": "02:42:ac:11:00:04",
        "IPv4Address": "172.17.0.4/16",
        "IPv6Address": ""
    },
    "9aa64454c50b4771dbe6526ec6996b94de8cece2292f23968b06897ad0eb001e": {
        "Name": "atc_tower",
        "EndpointID": "cf25617b470d04d9af72f07e04cc43834217bc679d4886fdb084387ef4697d5f",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
    }
},
```

Example Container IPs

The ATC webpage should now be accessible from “http://127.0.0.1:8000/client/”. First, the validator address must be set to match the one found in the “docker network inspect bridge” command. Also, whenever a message is sent to an aircraft, the “Sender Channel” must be changed to the ATC address found in the “docker network inspect bridge” command, as well as including the port number :8000, as shown below. This is a shortcoming of running the system in a local docker configuration, as by default, the docker networking bridge separates the address used for container to container communication from the address used for client communication. A bridge can be set up so that the webpage url matches the container communication url, allowing the sender channel to be correctly autofilled.



Network Configuration

Validator Address

172.17.0.2

Send Message

Sender Channel (please set to public ip)

http://172.17.0.3:8000

Recipient

apTest6

Message

CLEAR LAND 18

Message Type

[✓ COMMAND]
ACK

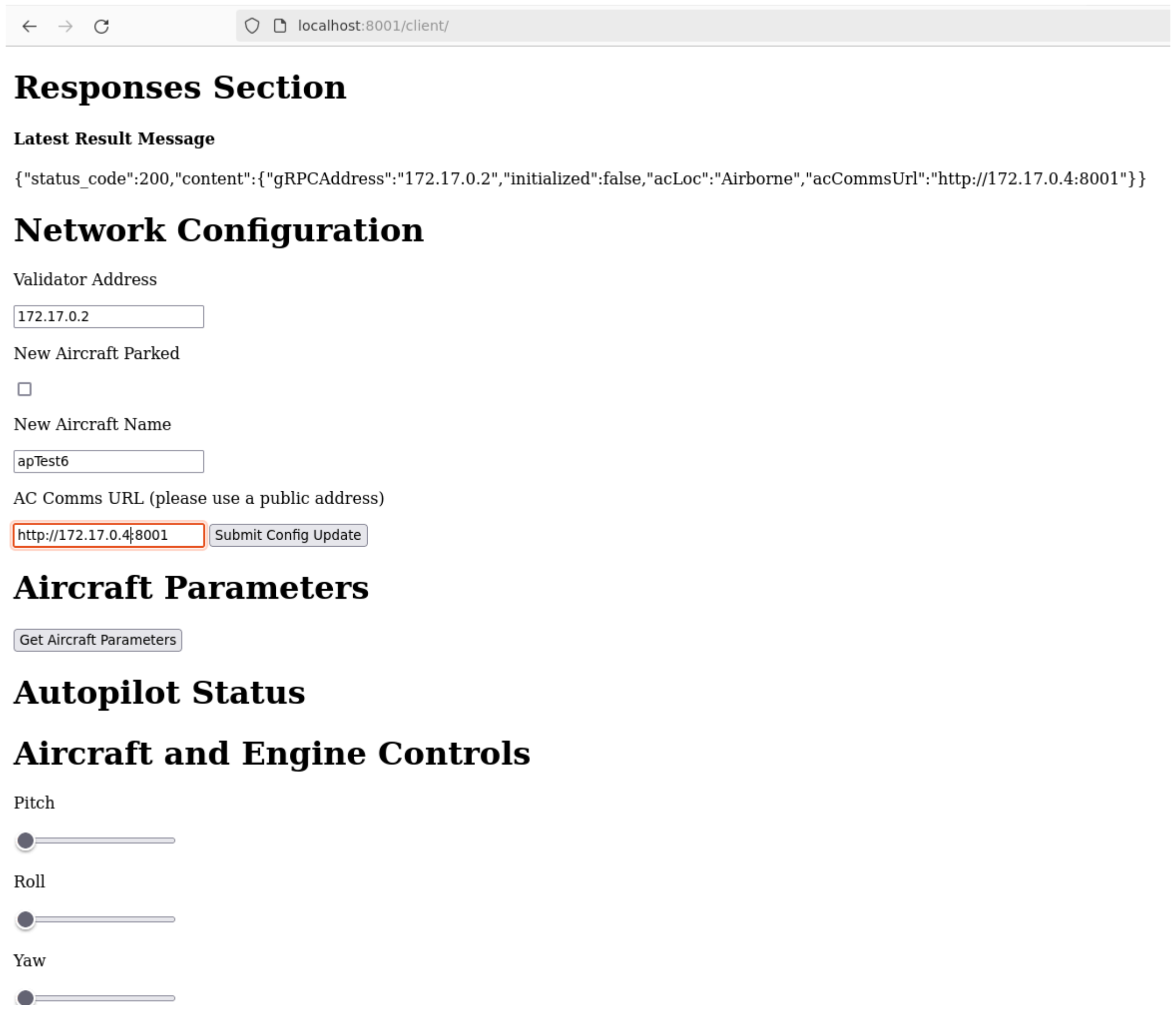
Inbound Message Queue

Sender	Message	Time Sent	Acknowledge
apTest5	apTest5 CLEAR TAXI RUNWAY A	1712372038	Acknowledge
apTest5	Unable to comply: Not Parked	1712372049	Acknowledge
apTest5	Unable to comply: Not at runway	1712372066	Acknowledge
apTest5	Unable to comply: Not at runway	1712372102	Acknowledge
apTest5	Unable to comply: Not Parked	1712372133	Acknowledge
apTest6	apTest6 HOLD VOR 1	1712372217	Acknowledge
apTest6	Unable to comply: Invalid VOR	1712372390	Acknowledge
apTest6	Unable to comply: Invalid Runway	1712372406	Acknowledge
apTest6	Unable to comply: Not Airborne	1712372543	Acknowledge

Message Log

Sender	Recipient	Message	Time Sent
ATC	apTest5	CLEAR TAXI RUNWAY A	1712372015033384000
ATC	apTest5	CLEAR TAXI RUNWAY A	1712372049126634500
ATC	apTest5	CLEAR TAKEOFF 18	1712372065883066600
ATC	apTest5	CLEAR TAKEOFF 18	1712372102644209000
ATC	apTest5	CLEAR TAXI RUNWAY A	1712372133039448000
ATC	apTest6	HOLD VOR 1	1712372217499694000

The aircraft's webpage can be accessed from "http://localhost:*AC_PORT*/client/", where "*AC_PORT*" must be replaced by the port number selected prior. First, the validator address must be set to match the one found in the "docker network inspect bridge" command. Also, the "AC Comms Url" must be changed to the aircraft's address found in the "docker network inspect bridge" command, as well as including the port number selected for the aircraft, as shown in the example below. As explained prior, this is a shortcoming of running the system in a local docker configuration.


localhost:8001/client/

Responses Section

Latest Result Message

{"status_code":200,"content":{"gRPCAddress":"172.17.0.2","initialized":false,"acLoc":"Airborne","acCommsUrl":"http://172.17.0.4:8001"}}

Network Configuration

Validator Address

172.17.0.2

New Aircraft Parked

☐

New Aircraft Name

apTest6

AC Comms URL (please use a public address)

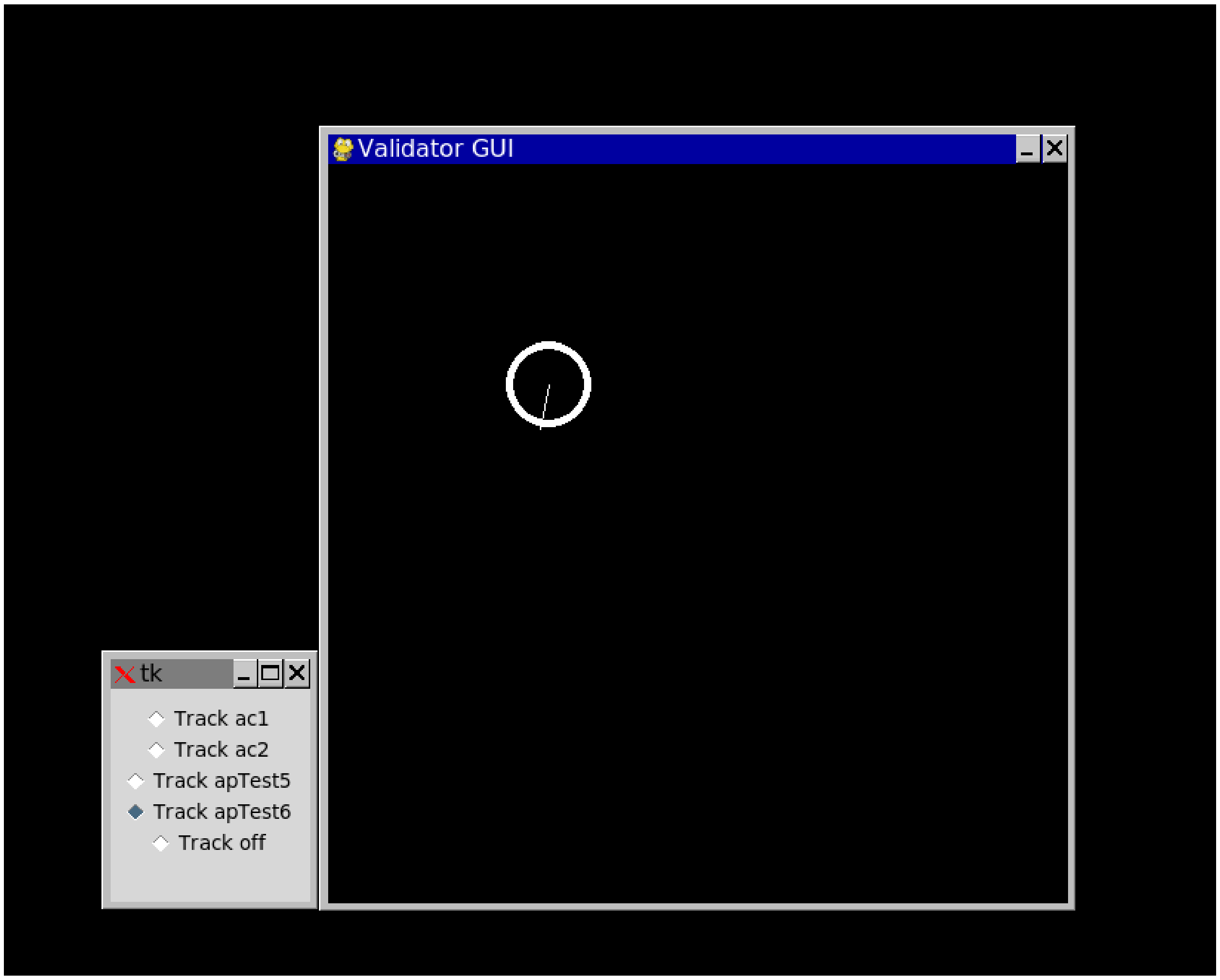
http://172.17.0.4:8001

With the validator, ATC, and all aircraft setup and running, you can now try issuing commands from the ATC to the various aircraft, observing the responses that show up in the inbound message queue.

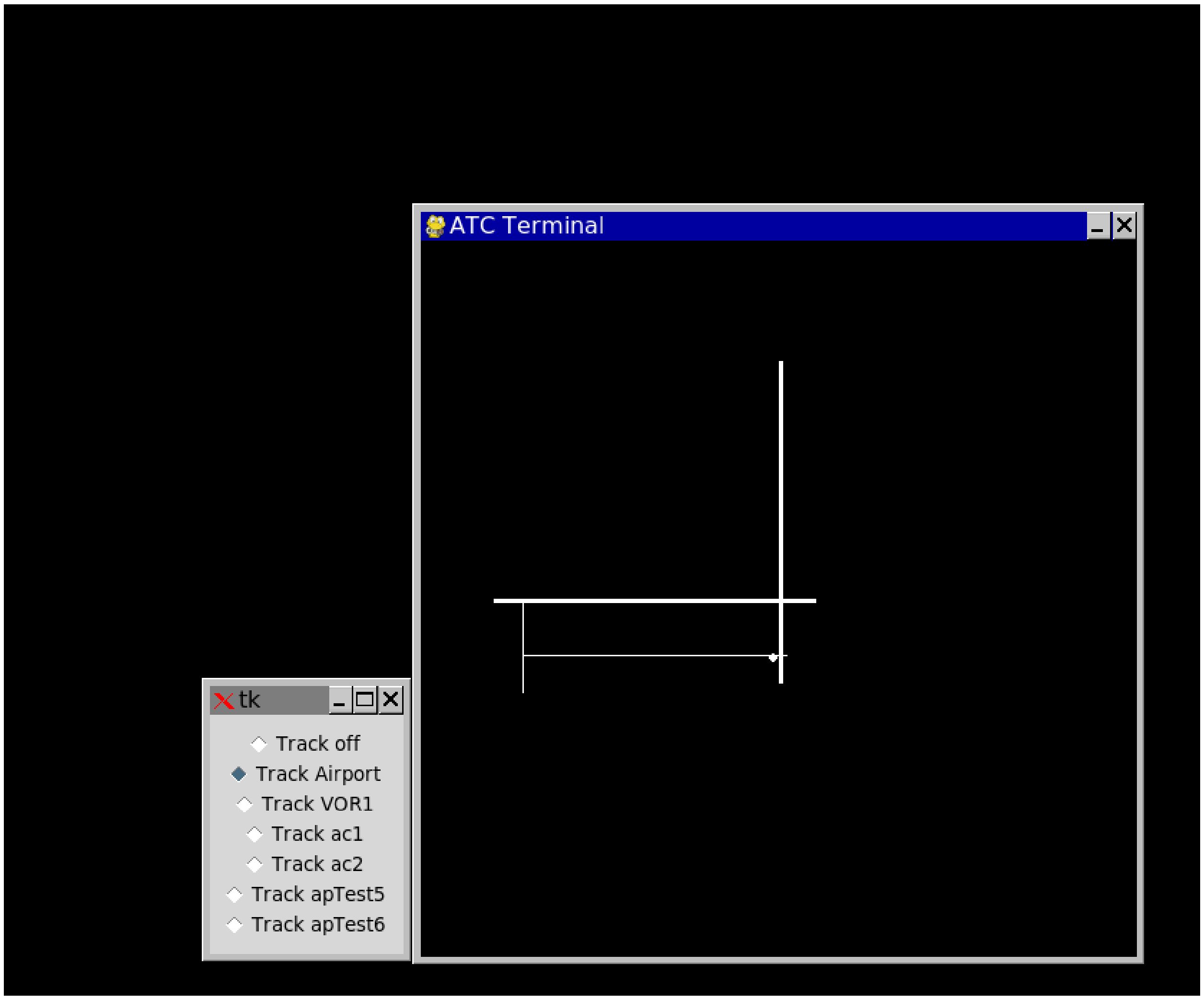
Optional GUI Setup:
The system also comes with two GUI applications to view aircraft states, found in ./validator/validatorGui.py and ./atc/atcTerminal.py. Both GUI applications require the "*pygame*" library

to be installed via pip, and also require that the version of Python is built with "*tKinter*" support. Python with "tKinter" support can be either installed from the Python website for Windows, and MacOS, from Homebrew on MacOS, and from your distribution's package manager on Linux.

Both programs open two windows, one being the main screen with objects drawn on it, and another being a smaller screen with radio buttons to select what to track. The view can be manipulated, by using the up and down arrows to zoom in and out, and *when Track is off*, clicking the main screen and dragging allows the view to be moved around. Also, while hovering over an aircraft, a tooltip will pop up displaying its velocity and position.



Shown below is the ATC Terminal, which provides a few more features. It also draws the airport and VOR, which can be tracked using the corresponding track options.



Tasks

The tasks were broken down into 3 categories: Air Traffic Control, Validator, and Aircraft system.

The Air Traffic Control system is tasked with receiving requests from numerous aircraft and determining optimal instructions to minimize delays and avert potential safety hazards.

- Research and define the key components and requirements of an air traffic control system.
- Design a simplified version of the air traffic control system architecture.
- Implement algorithms for vectoring in aircraft from their current position to the airfield, giving consideration to the fact that the aircraft heading and airfield heading must match

The validator examines the present positions of aircraft to ascertain whether any collisions have taken place. If such an event occurs, it promptly notifies both the Air Traffic Control system and the aircraft involved. Additionally, the validator serves the purpose of furnishing aircraft with visibility-related information, which is contingent upon factors such as speed and size.

- Study collision detection algorithms.
- Implement a simplified physics engine for the simulated environment.
- Implement collision detection
- Develop algorithms to calculate visibility-related information for aircraft based on speed and size.
- Improve the visualization tool to visualize height, aircraft names, and heading

The Aircraft system functions as a client of both the Air Traffic Controller and Validator. It initiates requests to the Air Traffic Controller and executes commands received from it. Additionally, leveraging spatial data provided by the Validator, the aircraft may opt to refuse commands it deems hazardous.

- Define the communication protocols between aircraft and the air traffic control system. (done)
- Develop a basic simulated aircraft model capable of sending and receiving messages. (done)
- Develop algorithms for aircraft to use the vectoring information from air traffic control to adjust the velocity components

Milestones Achieved

Of the tasks mentioned in the Tasks section of this report, the following tasks were completed:

Ashreet has completed base implementation of the air traffic controller. It makes requests to the validator to see the position of aircraft in range, and send commands to them based on their position relative to the airport. The air traffic controller considers the vector representing the airplane’s current heading and the vector representing the airplane’s position relative to the airport, and calculates the new heading. Having a base interface for the other teammates to work around means that future work can be done more independently, as the other teammates can refer to the already defined interfaces. Ashreet further advanced the air traffic control system by implementing algorithms for managing multiple aircraft and preventing collisions. These algorithms were designed to analyze the positions and trajectories of multiple aircraft simultaneously, prioritizing safety and efficiency. Collision avoidance measures were integrated to detect and mitigate potential conflicts, ensuring safe separation between aircraft. Additionally, the team expanded the system's capabilities to consider multiple airfields and overlapping airspaces. This involved modifying algorithms to accommodate diverse operational scenarios and enhancing coordination between air traffic control sectors. Through iterative development and rigorous testing, the team successfully adjusted the algorithms to handle complex airspace configurations while maintaining optimal aircraft routing and safety standards. This collaborative effort enabled the system to evolve into a robust and scalable solution for managing air traffic efficiently across various operational environments.

Atheesh has completed the base implementation of the aircraft. It takes requests that give it a new heading to steer towards, while also updating the validator of its current position and velocity. Since the validator calculates the aircraft’s heading using the X, Y and Z components of its velocity, trigonometry was used to convert the heading into the correct X, Y and Z components to send to the validator. The fact that the speed must gradually change was also considered. As mentioned previously, a defined interface allows the other group members to work on their parts without slowing down to discuss the characteristics of the interface. Atheesh progressed by studying collision detection algorithms, enabling the system to detect and mitigate potential conflicts in the simulated environment. Leveraging this knowledge, a simplified physics engine was implemented to model the dynamics of aircraft movement, considering factors such as velocity, acceleration, and collision responses. This engine facilitated the implementation of collision detection mechanisms, which analyze the spatial relationships between aircraft and their surroundings to ensure safe separation. Furthermore, algorithms were developed to calculate visibility-related information for aircraft based on their speed and size, enhancing situational awareness for air traffic controllers. Additionally, the visualization tool was enhanced to display aircraft height, names, and headings, providing comprehensive situational awareness for operators. By integrating these advancements into the air traffic control system, the team achieved a more comprehensive and efficient solution for managing air traffic, improving safety and operational effectiveness in simulated environments.

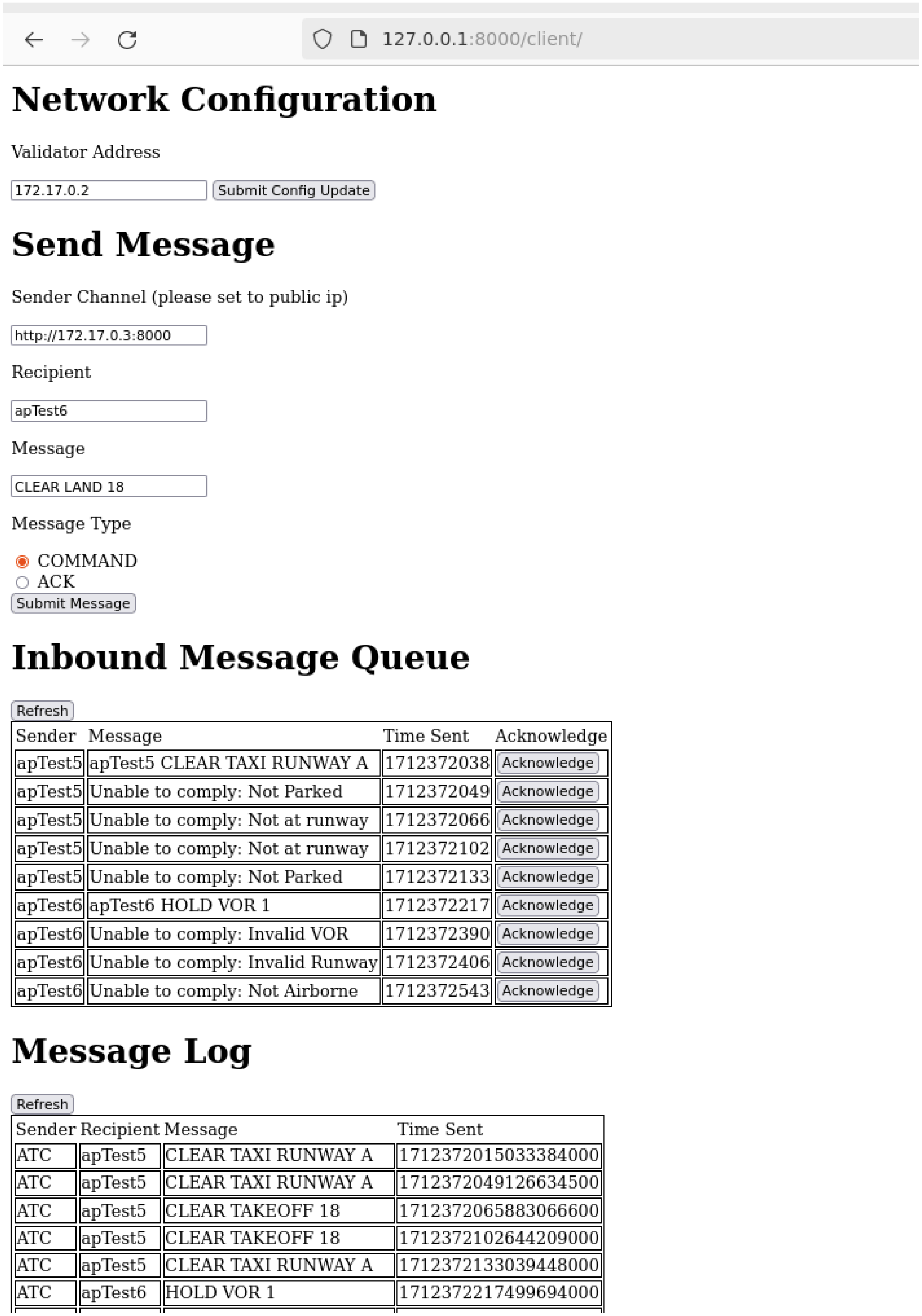
Saiharan has completed the structure for the validator, as well as an initial set of procedures and data structures the validator uses. The validator works by storing the position and velocity of each aircraft, and updating it in a set update period. Meanwhile, a gRPC server listens for calls either reading or updating the aircrafts' information. Also, a small separate program was made that reads the aircraft information, and displays it on a grid, to ensure correct operation. As mentioned previously, having a base interface for the other teammates to work around means that future work can be done more independently, as the other teammates can refer to the created. Saiharan progressed by defining communication protocols between aircraft and the air traffic control system, ensuring seamless exchange of information. These protocols were implemented within the simulated aircraft model developed by Saiharan, allowing aircraft to send and receive messages effectively. Moreover, algorithms were developed to enable aircraft to utilize vectoring information provided by air traffic control to adjust their velocity components, optimizing routing and ensuring adherence to air traffic instructions. Additionally, the system was enhanced to facilitate aircraft making requests to air traffic control, enabling dynamic interaction between aircraft and controllers. Furthermore, robust error handling mechanisms were implemented to address scenarios where an aircraft is deemed crashed by the validator, ensuring appropriate actions are taken to maintain system integrity and safety. Through collaborative effort and iterative development, the team successfully integrated these functionalities into the air traffic control system, further advancing its capabilities and effectiveness in managing simulated air traffic scenarios.

Output with Screenshots

Once the setup steps are followed, the ATC webpage is accessible. Its main function is to send commands to aircraft. Valid commands include

- CLEAR LAND 18
- CLEAR LAND 27
- CLEAR TAXI RUNWAY A
- CLEAR TAXI RUNWAY B
- CLEAR TAXI PARKING A
- CLEAR TAXI PARKING B

Whether or not the commands are followed depends on the aircraft's state. For example, a parked aircraft will not comply with a CLEAR LAND command.



127.0.0.1:8000/client/

Network Configuration

Validator Address

172.17.0.2 Submit Config Update

Send Message

Sender Channel (please set to public ip)

http://172.17.0.3:8000

Recipient

apTest6

Message

CLEAR LAND 18

Message Type

[✓ COMMAND] ACK

Submit Message

Inbound Message Queue

Refresh

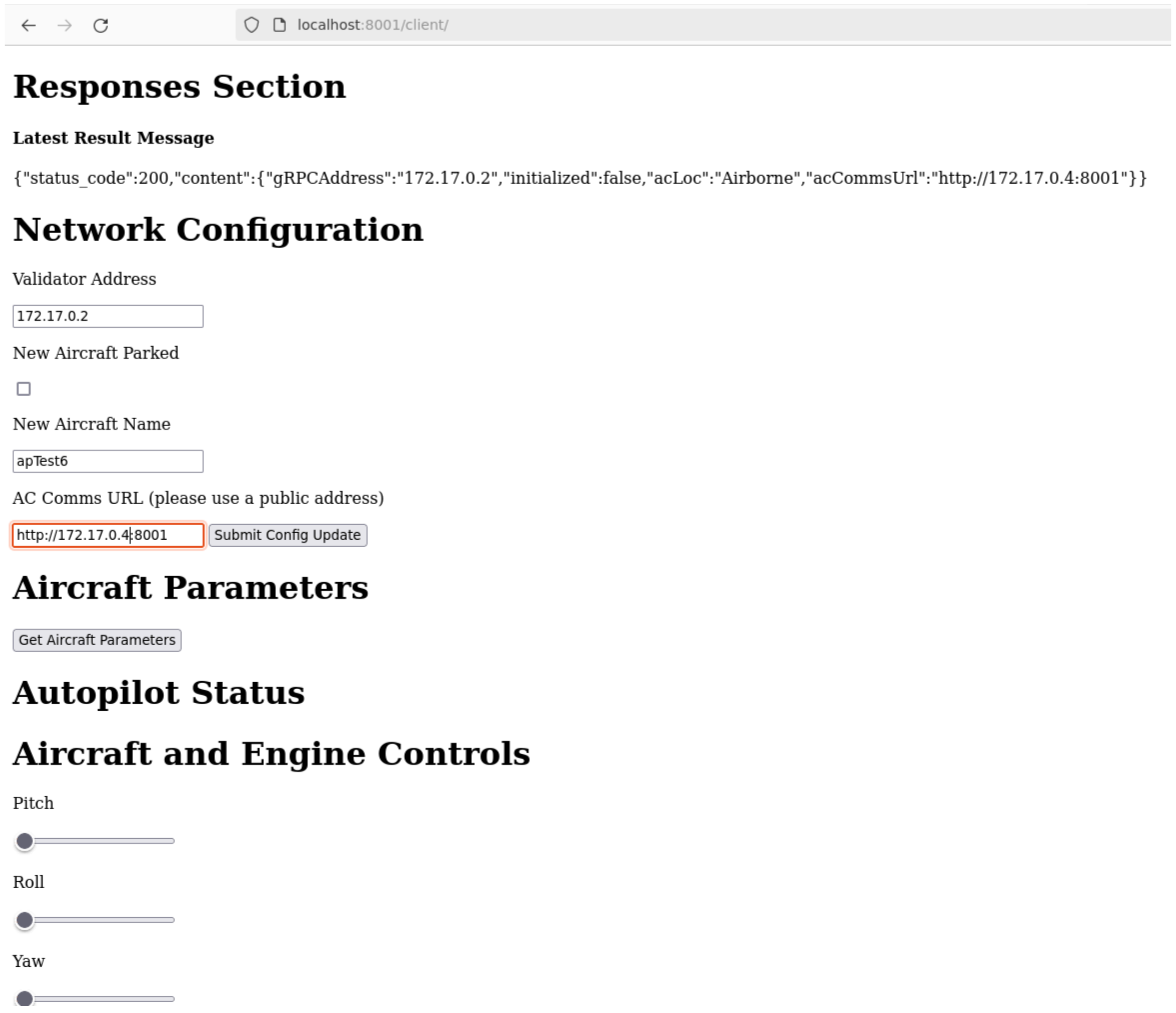
Sender	Message	Time Sent	Acknowledge
apTest5	apTest5 CLEAR TAXI RUNWAY A	1712372038	Acknowledge
apTest5	Unable to comply: Not Parked	1712372049	Acknowledge
apTest5	Unable to comply: Not at runway	1712372066	Acknowledge
apTest5	Unable to comply: Not at runway	1712372102	Acknowledge
apTest5	Unable to comply: Not Parked	1712372133	Acknowledge
apTest6	apTest6 HOLD VOR 1	1712372217	Acknowledge
apTest6	Unable to comply: Invalid VOR	1712372390	Acknowledge
apTest6	Unable to comply: Invalid Runway	1712372406	Acknowledge
apTest6	Unable to comply: Not Airborne	1712372543	Acknowledge

Message Log

Refresh

Sender	Recipient	Message	Time Sent
ATC	apTest5	CLEAR TAXI RUNWAY A	1712372015033384000
ATC	apTest5	CLEAR TAXI RUNWAY A	1712372049126634500
ATC	apTest5	CLEAR TAKEOFF 18	1712372065883066600
ATC	apTest5	CLEAR TAKEOFF 18	1712372102644209000
ATC	apTest5	CLEAR TAXI RUNWAY A	1712372133039448000
ATC	apTest6	HOLD VOR 1	1712372217499694000

As described in the setup page, the aircraft's webpage will be accessible from a browser. Both its message tables can be updated using the corresponding update button, and the aircraft parameters can be read using the Get Aircraft Parameters button. Adjusting the Aircraft and Engine controls will update the control state internally, as will be shown by the responses section.



localhost:8001/client/

Responses Section

Latest Result Message

```
{"status_code":200,"content":{"gRPCAddress":"172.17.0.2","initialized":false,"acLoc":"Airborne","acCommsUrl":"http://172.17.0.4:8001"}}
```

Network Configuration

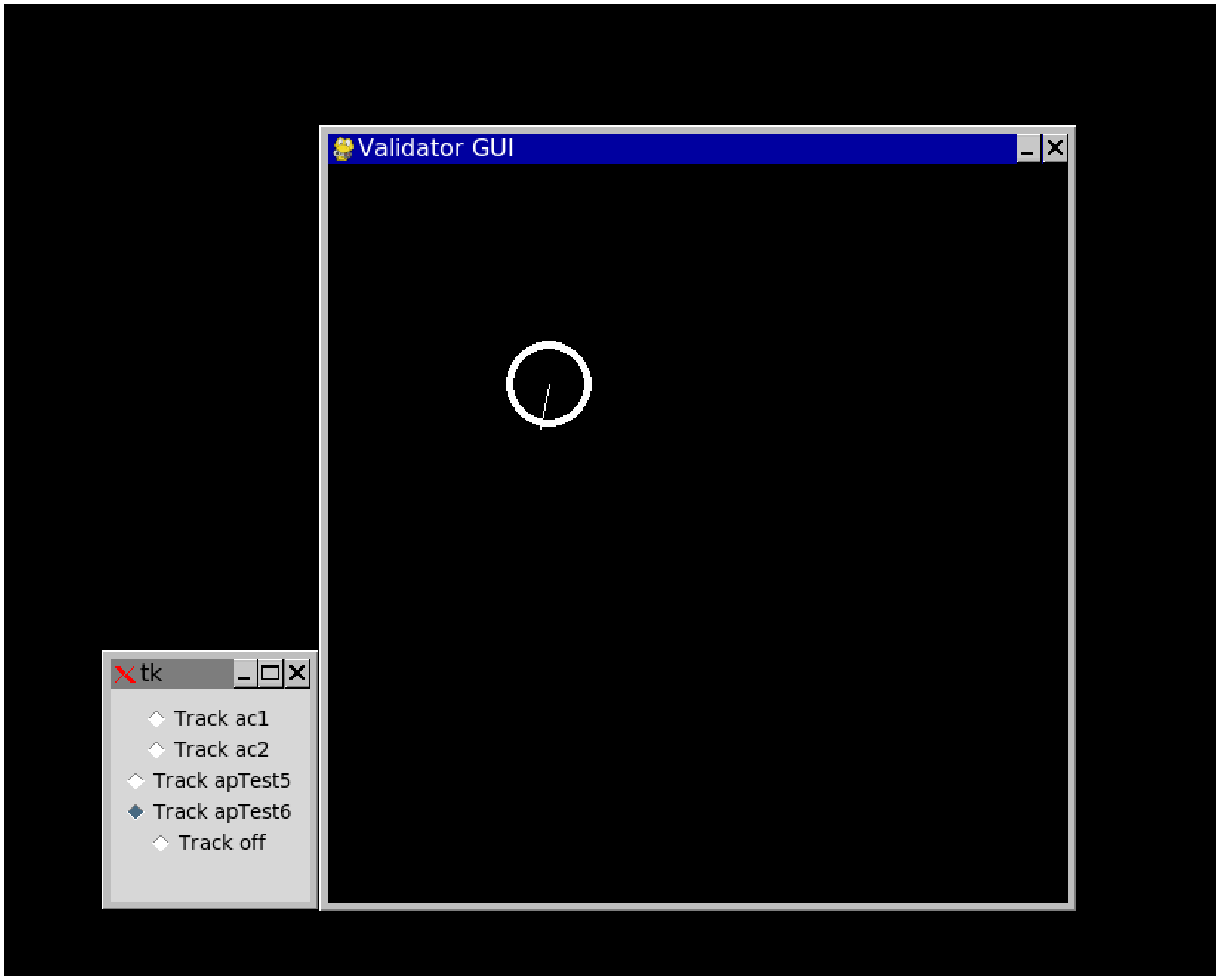
Validator Address: 172.17.0.2

New Aircraft Parked: ☐

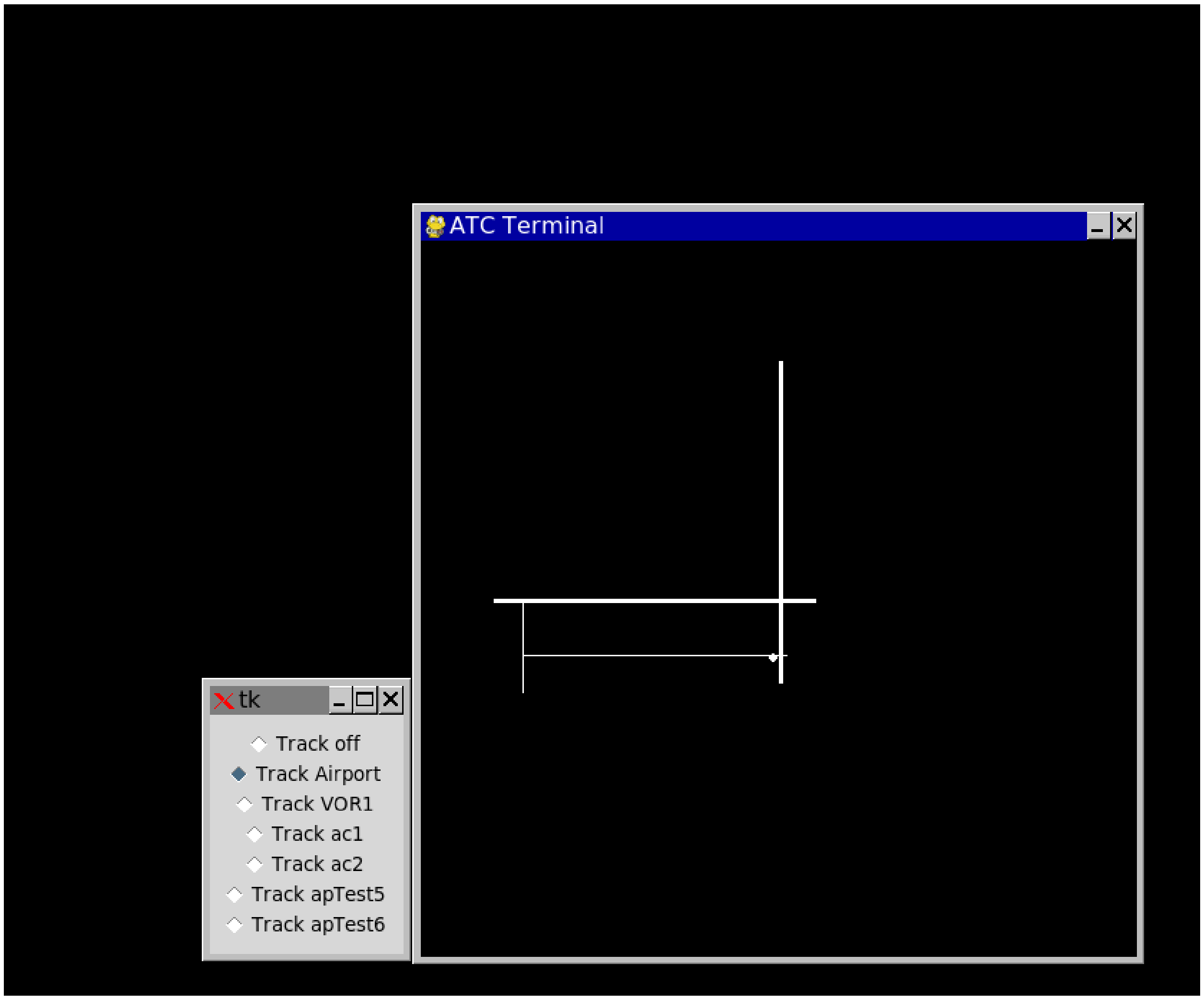
New Aircraft Name: apTest6

AC Comms URL (please use a public address): http://172.17.0.4:8001

Both GUI programs open two windows, one being the main screen with objects drawn on it, and another being a smaller screen with radio buttons to select what to track. The view can be manipulated, by using the up and down arrows to zoom in and out, and *when Track is off*, clicking the main screen and dragging allows the view to be moved around. Also, while hovering over an aircraft, a tooltip will pop up displaying its velocity and position.



Shown below is the ATC Terminal, which provides a few more features. It also draws the airport and VOR, which can be tracked using the corresponding track options.



All Technologies Explained

In terms of communications technologies used, the system takes advantage of gRPC, and FastAPI to allow for communication between each component.

Communication to the validator was implemented in gRPC, which was selected for performance. To reiterate, the validator was responsible for being the physical environment in which aircraft and ATC would interact with each other, handling physical simulation, and visibility conditions. Therefore, any latency would not be tolerable, as the environment provided by the validator should be as close to reality as possible. For example, the environment would update every 0.33 milliseconds, during which the ATC and all aircraft instances would make calls to update their situational awareness. A high latency communication platform would mean that instances would have delayed responses to incidents such as collisions, which is not a factor in a real deployment.

Communication between ATC and aircraft instances was done using a REST API developed using FastAPI. A REST API was chosen for this type of communication for numerous reasons. Firstly, for this application, gRPC's performance did not provide any benefits, as the rate of communication would be bottlenecked by the ability of operators to perform the required actions and respond, as opposed to validator communication which was handled by software making calls in short intervals. Furthermore, gRPC's rigid data structures are incompatible with the type of communication occurring between ATC and aircraft, which is more flexible as it is based on natural human languages. Finally, the REST API allows for more flexibility on the software used in clients. Consuming a REST API is doable using standard libraries in many languages, and at most a wrapper library could be used to simplify the calls. On the other hand, trying to consume from a gRPC service in JavaScript was found to be extremely complicated, requiring the simple HTML+JavaScript frontend to be migrated to Node.JS. FastAPI was chosen as the library used to implement the REST server due to the various capabilities it has that improved the functionality of the system. First was its ability to be quickly deployed with Uvicorn, which allowed for a seamless transition between development and deployment, unlike libraries such as Django and Flask which require more effort during deployment. Another was its ability to host HTML files, which was useful for giving the aircraft and ATC instances an interface for users to understand and change the state of the instances.

In the validator, aircraft, and ATC applications, synchronization methods needed to be applied, as the application would be serving and updating its state over the network, as well as locally, either by a separate thread responsible for updating the state or by a user. While the selected libraries, FastAPI and gRPC, provided an environment for network clients to be handled asynchronously, the local threads still needed to be managed. First, an Event object, from the Python threading library, was used to notify the threads of events, such as if the main thread received a signal to shut down, it would use an event to notify a thread. Next, Lock objects were used to manage access between resources. For example, if an aircraft object did a remote procedure call, modifying its velocity, the validator would lock the aircraft object, which would prevent the update thread from modifying it and potentially leading to an erroneous state.

For deployment, the technology chosen was containerization, provided by Docker. Containerization was chosen because it greatly simplified the deployment of the application. Once images were created, the containers could be quickly deployed locally with a command, or on a public cloud using container hosts. Considering that the aircraft component could have several deployments, this was worth the tradeoff of taking the time to create the images. Furthermore, creating the images ensured that the project that was submitted was host-agnostic, preventing difficulties in trying to deploy the project in the future. Containerization also provided no disadvantages, since unlike a virtual machine, the performance penalty would be minimal, and since the project did not use any privileged resources, a bare metal deployment provided no advantages other than insignificantly lower overhead.

Challenges

One of the challenges that were encountered was trying to manage the variables that would be used in the main loops of the program, but would also be updated by incoming calls. For example, the aircraft would be managing its velocity as it turns to the heading requested by the air traffic controller. However, if the air traffic controller requested another adjustment, or the validator updated the aircraft's state, the aircraft program would need to handle this. To solve this, locks had to be used to ensure that the state was not erroneously changed simultaneously. Another challenge was that since the three program parts were being worked on separately, disagreements would arise on how the data would be passed between each program. In the end, it was decided that the initial data structures would be agreed on first, then the implementation could proceed.

Conclusion

In conclusion, our progress report highlights the significant advancements made in developing our air traffic control system prototype. Through the diligent efforts of our team members, we have successfully completed tasks across the Air Traffic Control, Validator, and Aircraft System domains, laying a solid foundation for efficient communication and coordination between the control center and simulated aircraft clients. Despite encountering challenges, our proactive approach to problem-solving has ensured steady progress. Moving forward, we remain committed to refining our prototype, confident in our ability to deliver a robust and effective air traffic management solution.