Table of Contents

Objective

The goal of this project is to implement a fully functional real time air traffic monitoring and control (ATC) system using QNX. This includes brainstorming, designing, working in a team, testing, analyzing and implementing a real time system to achieve the objective of this project.

Introduction

A great example of a real time system is an Air Traffic Monitoring and Control system (ATC). The ATC project involves implementation and testing of a real time air traffic control system which is being executed in QNX real time OS using C++. Moreover, the goal of the project is to maintain air traffic flow, by separating the aircraft from each other, so that it can prevent any sort of clash or collision.

The simplified ATC is maintained by having a computer system which takes input from the radar and Operator console. The radar's job is to get Flight ID, flight level of the aircraft, aircraft's speed and aircraft position and send it to the computer system. The operator's work is to request an aircraft to change its speed, altitude and position. Then, the computer system checks for aircraft collision, and displays the IDs and position of aircraft on the console. Next, the computer system transmits the controller command to the communication system which is used to send a command to the aircraft.

Next, the requirements for this project is to implement the ATC in C++ using the QNX development environment. Initially, the project was planned to be done by using memory sharing, however message passing was applied to pass information to all the classes. Message passing is done by communicating and synchronizing messages without even sharing the same address, hence, communication is done with the help of the network which is already present on the virtual machine.

Additionally, threads were implemented by creating a thread using pThreads. pThreads are the API thread implementation for C++. PThreads allow tasks to run in parallel, concurrently and control multiple workflows by spawning new concurrent processes.

Analysis

This project has many requirements all of which must be fulfilled to achieve a working simulation of a simple air traffic monitoring and control (ATC) system. Firstly, an ATC is used to control the airspace at any time to allow for aircrafts to move within a designated area in the airspace. There are two main goals of the ATC. The first being that it must maintain order in the air traffic flow. It must ensure that aircraft are not colliding or on a collision course, therefore it must ensure that the movements made are safe. The second is that the ATC needs to prevent collisions from happening within the airspace by diverting and reorienting aircrafts. The version that is to be implemented is a simplified version of the ATC which is a much more complex system.

The airspace that is considered is that of a 3D rectangle with dimensions 100000 ft by 100000 ft by 25000 ft situated at 15000 ft above sea level. The ATC needs to manage the aircraft within this designated area. It must do so with certain requirements. Firstly, the current position of each aircraft must be shown every five seconds for the display, then constraint violations must be checked at a certain interval. The velocity must be able to be changed if a collision is oncoming. Next, an alarm must be emitted if there is a safety violation that is found or if one will happen within three minutes. There must also be a history file made that needs to be updated every 30 seconds that contains the history of the airspace and finally, every request made should be stored in a log file.

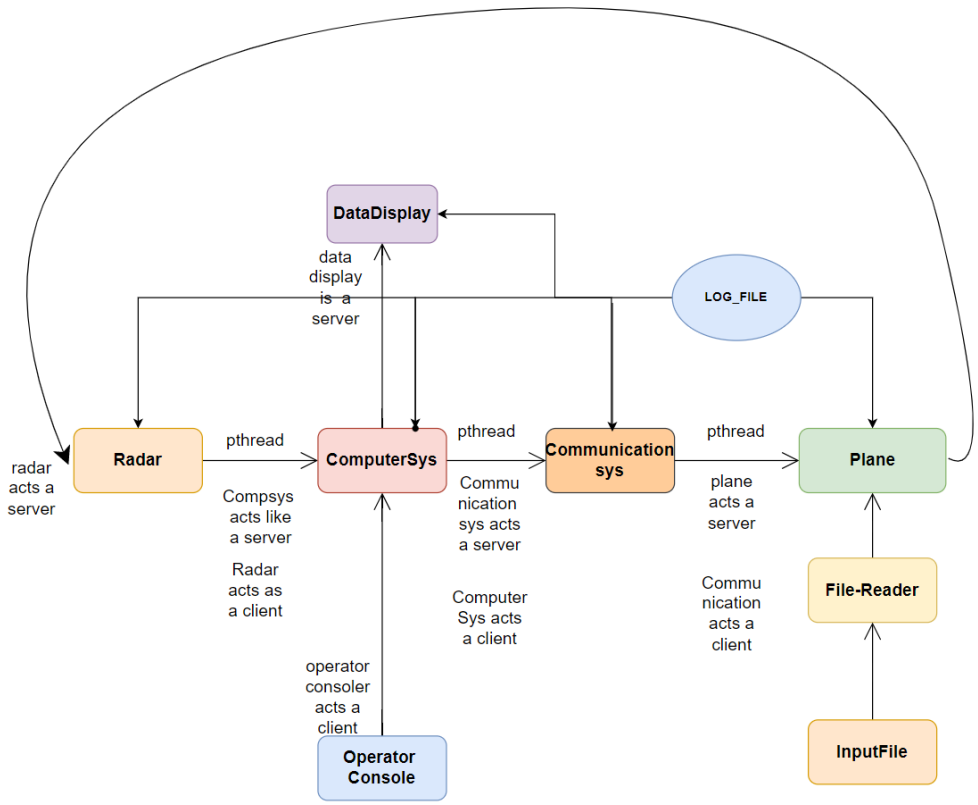
To do this, five subsystems are necessary. The radar, the computer system, the communication system, the operator console, and the data display. There are two radars necessary for the correct functionality of the ATC. The primary surveillance radar (PSR) and the secondary surveillance radar (SSR). The PSR, as its name suggests, verifies the aircrafts that are in the airspace. The SSR on the other hand, must emit a signal that will allow for the following information to be obtained by other subsystems: the flight ID, the aircraft flight level, the aircraft speed, and the aircraft position. The computer system is the brains of the entire ATC. It is required for this sub-system to emit an alert if there is a safety violation and it is required to send the ID of the aircraft who’s information needs to be displayed. The operator console is the sub system that takes the commands from the user and sends signals for the commands to be applied or information to be requested. The communication system is the subsystem that is required to send messages (or to communicate)

with the aircrafts.

On the technical side of the requirements, an input file containing aircraft information is used to insert aircrafts into the airspace. It contains information about position, speed, time, and ID. Each aircraft is required to be a thread. The system should handle periodic tasks and sporadic tasks should be dealt with using the periodic polling method. The system is required to be tested under different loads with multiple test cases to ensure proper functionality.

Design

The following design was used to code for the Air Traffic Control System. We considered each class to be a server and client, which will be used to receive and send messages by using message passing with the help of thread. This shows how we planned to solve the problem.



We planned our ATC components around the diagram provided in the project description. The radar would act as a server, since it was receiving information from the planes. The radar would then pass the information to the computer system, another server. The computer system communicates with many of the components. It passes data to the data display, operator console and communication system. The operator console passes operator inputs back to the computer system as well. The communication system receives the operator inputs from the computer system and passes them to the respective planes. The file reader component generates the input file, then parses the information and passes it to the planes to construct them. Lastly, the plane, communication system, computer system and radar passes information to the log component for logging errors and any useful information in a log file.

Implementation

There are several classes created for this project: 320_mnst, ATC, Computer_system, Comsys, data_display, fileReader, log, opConsole, plane, Radar and Timer. The following methods and classes were used in order to achieve the goal of the project:

1) ***320_final.cpp*** is the main class and its purpose is to call the ATC start function.

```
18 int main() {
19     // Call ATC start function to start ATC operation
20     ATC atc;
21     atc.startATC();
22
23
24     return 0;
25 }
26
```

Figure 1: Screenshot of 320_final.cpp

2) **ATC**: ATC function is used to start the entire system. It constructs all the components needed to run the ATC. It also calls the generate planes function from the fileReader class.
3) **Computer System**: The Computer System is responsible for logging the positions of the aircrafts periodically, checking for any airspace violations between the aircraft and forwarding messages between all the components of the ATC. The Computer System begins by creating a thread and calling the **compSys_start** function. The start function then calls the **positionLog** function. This function is responsible for logging the position of the aircrafts. The function uses the PSR first to find the number of planes in the airspace. It then uses the SSR, with the information from the PSR to find the positions, velocity and ID's of all the planes in the airspace. It stores this information in a vector pair, with the first element containing the Plane ID as an integer and the second containing a **PlaneInfo** data type, which holds two 3 dimensional vectors.

```
void ComputerSystem::positionLog(data_display display, PSR psr, SSR ssr, vector<Plane> planes) {
    // Temp radar function name
    cout << "Entering position log" << endl;
    vector<int> airspace;
    airspace = psr.findAllPlanesInAirspace(planes);

    vector<pair<int, PlaneInfo>> planePositions = ssr.interrogatePlanesInAirspace(airspace, 1);
```

Figure 2: Beginning of Position Log Function

One vector holds the position in x, y and z and the other holds the velocity. This vector pair gets forwarded to the data display using message passing. Furthermore, the function also logs the positions and velocities of the planes to a file every 30 seconds. First a timer is declared with a period of 30 seconds, then the data is stored into an output file using *ofstream*. Once written the timer waits for the next period to repeat the cycle.

checkAll is the next function in the Computer System and iterates through a nested for loop limited by the number of planes in the airspace. Inside the for loop, the **checkViolation** function is called for two planes at a time. The **checkViolation** function takes two planes and a lookaheadValue, which can be adjusted by the controller. The lookahead value determines how far the algorithm will look into the future and check for violations. The algorithm begins by multiplying the current velocity of the planes by the lookahead value and adds it to the current positions, to get the future positions of those planes.

```
int zLimit = 1000;
int xyLimit = 3000;
// Create timer for checking constraint violations at current_time+n seconds, n is input by the operator.
Timer checkTimer(lookaheadValue, 0, 0, 0);
while(true){
    Vec3 p1CurrentPos = p1.getCurrentPosition();
    Vec3 p2CurrentPos = p2.getCurrentPosition();
    Vec3 p1CurrentVel = p1.getCurrentVelocity();
    Vec3 p2CurrentVel = p2.getCurrentVelocity();

    Vec3 p1Projection = (p1CurrentPos.sum(p1CurrentVel.scalarMult(lookaheadValue)));
    Vec3 p2Projection = (p2CurrentPos.sum(p2CurrentVel.scalarMult(lookaheadValue)));
```

Figure 3: Beginning of checkViolation function

It then takes the projected values and adds and subtracts the violation limits to/from them. This creates a 3-D rectangular space around the projected position of each aircraft that no other aircraft can occupy. An if statement then compares the maximum and minimum

values of the 3-D rectangular spaces of each plane.

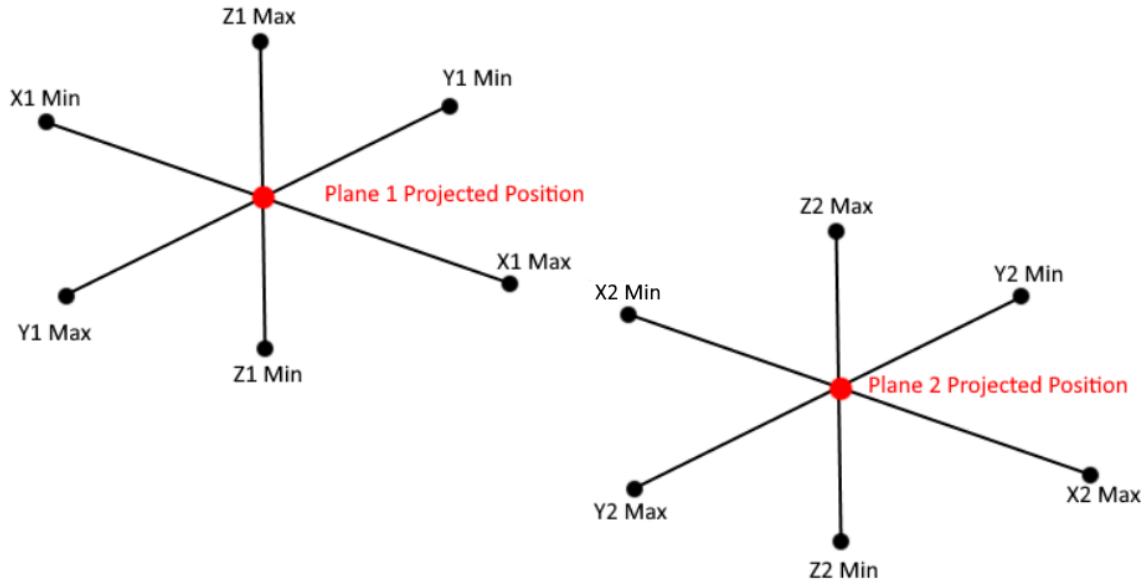


Figure 4: Visual representation of 3-D space for violation checks

If a maximum value from any dimension is greater than the minimum value of the other plane for the same dimension, then the system signals a violation. First the operator is notified, then a message is sent to the operator console containing one of the plane's information where the operator will then adjust the plane's velocity to avoid the violation.

```
// Take the projected position of the plane and add/sub the limits to get a 3D space around the plane that no other plane can occupy
int x1Min = p1Projection.x - xyLimit;
int x1Max = p1Projection.x + xyLimit;
int y1Min = p1Projection.y - xyLimit;
int y1Max = p1Projection.y + xyLimit;
int z1Min = p1Projection.z - zLimit;
int z1Max = p1Projection.z + zLimit;

int x2Min = p2Projection.x - xyLimit;
int x2Max = p2Projection.x + xyLimit;
int y2Min = p2Projection.y - xyLimit;
int y2Max = p2Projection.y + xyLimit;
int z2Min = p2Projection.z - zLimit;
int z2Max = p2Projection.z + zLimit;

// Check if any values overlap, signaling a violation
if ((x1Max >= x2Min && x2Max >= x1Min) && (y1Max >= y2Min && y2Max >= y1Min) && (z1Max >= z2Min && z2Max >= z1Min)){
```

Figure 5: Calculations for maximum and minimum values in checkViolation

Lastly, the remaining functions in the Computer System are used for communication amongst the other components of the ATC. One function, **waitForOpCon** uses message passing to receive messages for the Operator Console for velocity change requests. It then calls the **sendToComm** function, which forwards the velocity change to the communication system.

4) **Comsys:** This class is used for communication systems. This class consists of pthreads for message passing because it is a server. This class also contains 3 methods. The first method **getChid** is a simple getter for the communication channel ID. The second method is **waitForCommunication** which is used to wait for any speed changes from the operator. It first establishes a channel with the operator console and uses an indefinite while loop to wait for any changes. Once a message is received, the third method, sendToPlane function is called. This method takes the speed change from the previous method, and passes it to the plane by establishing a connection between it and the plane in question.
5) **Data display:** This class is used to display all the plane's data in the terminal. Data display contains three main methods. The first is the getChid, which is a simple getter for the display's channel ID. The second is receiveMsg, which uses an indefinite while loop to wait for any messages that the display will be sent from the computer system. When a message is received, the method will call the third method, **displayPlanes**. This method runs periodically, displaying the planes' positions every 5 seconds. It takes a vector pair containing the plane ID's in one vector, and the plane's coordinates and velocity in the other. The data is then displayed in the terminal, then waits for the next period to begin.

```
void data_display::displayPlanes(vector<pair<int, PlaneInfo>> received){

    Timer displayTimer = Timer(5, 0, 0 ,0);
    while (true){
    // Display the ID, coordinates, and velocity of the planes in the airspace
        for (int i = 0; i < sizeof(received); i++){
            int x1 = received[i].second.planePosition.x;
            int y1 = received[i].second.planePosition.y;
            int z1 = received[i].second.planePosition.z;

            int x2 = received[i].second.planeSpeed.x;
```

Figure 6: displayPlanes function

6) **File Reader**: This class is used to generate the input file which is **generatePlanes** and then reads the file which is **readFile** We created a random number generator for all the inputs such as the ID, arrival time, positions and velocity for x,y, and z to generate the planes. This was created inside a for loop which iterates based on the numbers of plane inputs to allow different levels of congestion.

```
int ID, arrivalTime, x, y, z, sx, sy, numOfPlanes;
int sz=0;
numOfPlanes = 20;
// Seed the random number generator
srand(time(NULL));
// Loop for randomly generating plane parameters. Change numOfPlanes to adjust congestion level
for (int i = 0; i<numOfPlanes; i++){
    ID = rand() % 100;
    arrivalTime = rand() % 3000 + 1;
    x = rand() % 100001;
    y = rand() % 100001;
    z = rand() % 25001;
    sx = rand() % 1001 - 500;
    sy = rand() % 1001 - 500;

    char buffer[8];
    sprintf(buffer, "%d,%d,%d,%d,%d,%d,%d,%d\n", arrivalTime, ID, x, y, z, sx, sy, sz);
```

Figure 7: Random plane generator in generatePlanes

The readFile opens the generated file and parses it for the initial parameters (**initialPlaneParameters**) which get stored in a variable **Plane(planeParams)**. This variable is then used to construct all the planes and stores them in a vector called **vecPlanes**.

7) **Log**: This class is used to log errors and any information that might be useful. It contains three functions, all of which are quite similar. The **LOG_ERROR** function takes a string

containing information about an error, and writes it to the log file on the virtual machine. The **LOG_INFO** and **LOG_INFO_WITH_VALUE** function do the same, but contain other useful information that is not associated with an error. The **LOG_INFO_WITH_VALUE** also takes an integer value so specific values can be written to the log file as well.

8) **Operator Console (opConsole)**: The Operator Console is responsible for allowing the operator to interact with the ATC. It contains three main functions, the first of which is **reqChangeAircraftInfo**. This function takes a Plane ID and receives an input from the operator for changing the associated plane's velocity. First, it asks the operator to enter the velocity in terms of X, Y and Z to avoid a collision. It then takes the input velocity and stores it in a 3-D vector called newVelocity. The **LOG_INFO_WITH_VALUE** function is called to log the velocity change, then the velocity change is forwarded to the computer system.

```
// Notify the operator that a collision will occur, and ask for an input for the velocity change in x, y and z
cout << "New velocity to avoid plane collision" << endl;
cout << "input new velocity in x: " << endl;
cin >> x;
cout << "input new velocity in y: " << endl;
cin >> y;
cout << "input new velocity in z: " << endl;
cin >> z;

// Assign the previously input values to the newVelocity variable
newVelocity.x = x;
newVelocity.y = y;
newVelocity.z = z;
```

Figure 8: Inputting the new velocity in reqChangeAircraftInfo

The next function is the **waitForMsg** function, which waits for a violation notification from the Computer System. It uses an indefinite while loop to constantly listen for any notifications, then calls the **reqChangeAircraftInfo** function.

Last is the **reqInfoRadar** function which takes an operator input to find the position and velocity of the requested plane. First the operator is asked to enter the ID of the plane they would like to get information from. This request is then logged using the **LOG_INFO_WITH_VALUE** function. The request is then sent to the radar, where it will return the requested information.

9) **Plane**: The Plane class contains four main functions. The first is **plane_start** which is

called when a plane thread is constructed. This function starts the plane's period tasks by calling the **updatePosition** function. This function runs periodically, every one second. It takes the plane's current velocity and multiplies it by the time period, in this case just one, and then adds it to the current position. This result is the updated position of the plane and is therefore assigned to the currentPositon variable. The function then checks if the plane has left the airspace, by checking the current position against the airspace constraints. If it has left, the left variable will be set to true, where the plane thread will terminate.

```
void Plane::updatePosition() {
    Timer planeTimer = Timer(1, 0, 0 ,0);
    int time_period = 1;
    while(true){
        currentPosition.x += currentVelocity.x * time_period;
        currentPosition.y += currentVelocity.y * time_period;
        currentPosition.z += currentVelocity.z * time_period;

        if (currentPosition.x < 0
            || currentPosition.x > 100000
            || currentPosition.y < 0
            || currentPosition.y > 100000
            || currentPosition.z < 0
            || currentPosition.z > 25000) {
            left = true;
        }
        planeTimer.wait_next_activation();
    }
    if (left == true){
        stop();
    }
```

Figure 9: updatePositon function

The next function is **waitForCmd**, which uses an indefinite while loop to wait for any messages sent by the communication system or radar. A case statement is used to differentiate the commands from one another. If the message command starts with 1, the radar is pinging the plane and it replies with its position and velocity. If the command starts with 2, the communication system is trying to send it a velocity adjustment where it will apply the new velocity and respond with an acknowledgement.

```
while (true) {
    // Wait indefinitely until message is sent to plane.
    rcvid = MsgReceive(chid, &cmd, sizeof(cmd), NULL);

    // If the message received is not a pulse
    if (rcvid != 0) {
        switch (cmd.command) {
        case 1: {
            // 1 signifies a ping from the radar, plane responds with position and velocity.
            currentPlanePosition curr{ currentPosition, currentVelocity };
            MsgReply(rcvid, EOK, &curr, sizeof(curr));
            break;
```

Figure 10: waitForCmd function

The last function is the **enterAirspace** function which creates a communication channel for the plane to allow for message passing.

10) **Radar**: This class contains both radars, the primary surveillance (PSR) and the secondary surveillance radars (SSR). A constructor and a destructor were created for both classes, the PSR and SSR. A method was created for the PSR **std::vector<int> findAllPlanesInAirspace(std::vector<Plane*>planes)** where it is used to find all the airplanes that are in the airspace. Inside the method, is a timer that checks every 1 sec to make sure if the plane has entered the airspace limits since the PSR function inspects all the planes and sees which are in the airspace. The SSR function has two methods, one that requests their info and one that sends the info to the computer system. The **interrogatePlanesInAirspace** checks for the plane's ID and requests the info from the plane and will send the message to the computer system. The **sendtoComputerSystem** takes a vector pair, one integer vector with the plane IDs and the other plane information which contains the aircraft's flight level, speed and position. It has a for loop where it will send the data to the computer system.

```
std::vector<std::pair<int, PlaneInfo>> SSR::interrogatePlanesInAirspace(std::vector<int>planes, int signal){

    // create a vector
    std::vector<std::pair<int, PlaneInfo>> planesInfo = {};

    //create a for loop in order that for each id, ask the plane for its info
    for (int j=0; j < planes.size(); j++){

            //create a channel
            int chid = ChannelCreate(planes[j]);

            //create a message

            planeCommand cmd = { NULL, 1, NULL};

            //send the message to the plane
            MsgSend(planes[j], &cmd, sizeof(cmd), NULL, 0);

            //receive the message from the plane
            currentPlanePosition crPP;
            MsgReceive(chid, &crPP, sizeof(cmd), NULL);

            // add the plane id and the response to the vector
            PlaneInfo planeInfo = {crPP.currentPosition, crPP.currentVelocity};
            planesInfo.push_back(std::make_pair(planes[j], planeInfo));
    }

    return planesInfo;
};

void SSR::sendToComputerSystem(std::vector<std::pair<int, PlaneInfo>> planes) {

    //create a for loop to send the info to the computer system
    for (const auto& data : planes)
    {
        MsgSend(data.first, &data, sizeof(data), NULL, 0);
    }
};
```

Figure 11: SSR function

11) **Timer**: The timer class contains all the necessary functions to construct timers for all of the periodic tasks in the ATC. The first function is the constructor, which takes four unsigned 32-bit integers as its input. The first two are the period in seconds and milliseconds respectively, while the latter two are the offset in seconds and milliseconds. The constructor then creates a SIGALRM signal and initializes the signal set. The signal event is then set and the timer is checked for successful creation. Lastly, the next function, **setTimer** is called. This function takes the same inputs as the constructor, except the millisecond inputs are in nanoseconds. This function assigns the period and the offset to the timer and starts it. Last, is the **wait_next_activation** function, which uses a dummy variable to wait for the next signal. When used in a process, the process will wait until the next signal using this function, allowing it to run periodically.

Overall, almost all of the Air Traffic Control system was implemented, but there were many difficulties seen along the way. First, the biggest issue we have in our program is threading. We struggled to get the threads to start properly and call the function from the **pthread_create** function. This prevents the airplanes starting their thread, not allowing them to call the **updatePostion** function and start flying. We also faced some difficulties in merging our classes together since some parts of the code were written long before other parts were. Many variable names and other aspects of the code had to be modified to remove errors and compile the code. Lastly, we faced some difficulties in getting our Log class working. At one point, there were too many log objects being called from separate classes with the same object name, which caused a compilation error. There were no indications of there being an error anywhere in the code, causing a long setback in our progress.

Contributions

Michael Dickson:

Programmed the Computer System, Planes, Timer, ATC and Log classes. Contributed code to the File Reader, Data Display, Operator Console and Communication System. Fixed many errors while trying to merge the code and spent many hours troubleshooting. Contributed to the implementation section of the report.

Theebika Thiyagaraja Iyer:

Programmed the Radar. Contributed code to the File Reader, Log file. Fixed errors when combining all the codes together. Spent many hours troubleshooting to see why the code isn't compiling and did a lot of testing. Organized the report and contributed to a big part of the report: objective, implementation, lessons learned and conclusion. Reviewed the code and removed/added comments.

Nabila Tabassum:

Programmed Communication System. Contributed threads in multiple classes. Helped in debugging errors while merging the codes. Contributed to the introduction, design and drawback, conclusion part of the report.

Simren Matharoo

Programmed operator console. Contributed code to multiple classes. Fixed errors and debugged code. Spent time troubleshooting. Contributed to the report including but not limited to the analysis section and the lessons learned section.

Lessons Learned

The project was more difficult than what we anticipated. We learned many valuable lessons throughout the course of the project. The project was programmed using the QNX development environment, meaning there was much to learn about QNX library functions such as **ChannelCreate,** which creates a communication channel for message passing, **ConnectAttach** which connects the client to the server's channel and **MsgReceive** (receiving message) for message passing. These were all important methods as they were the basis of inter-class communication in our project.

We learned about threads in operating systems and have been using them in most programming projects. In this project, we utilized pThreads for all our server classes. This allows the servers to run simultaneously on their respective threads and enables them to receive messages from clients. It allows for the implementation of a schedule to allow for the proper functioning of the ATC system.

Furthermore, while compiling the code, we learned that the order of the input header file was important. We had a lot of errors and by changing the order, the numbers of errors were reduced. This is something we did not know about and that it affected the compilation. Depending on how class objects were being called from other classes, the order that their respective header files were included in was very important. For example, if the Computer System was declared before the Plane class in any class, the code would not compile because the Computer System required specific data types, such as Vec3, from the Plane class to be declared first.

There were ways we could've changed how we worked on the project. For instance, we divided the tasks and each member took 1-2 classes. Then later on we combined everything together and then tried compiling the code. However, if we changed by having meetings from the beginning and we brainstormed the project together as a team, we could have had a different approach and could've worked efficiently. Also, if each member compiled had their code to make sure if it's compiling and then if all codes were compiled, it would have helped us and maybe our code would have compiled successfully.

Drawbacks:

1. One of the major drawbacks of designing the Air Traffic Control system was installing the QNX software on our laptops. Other than one, none of our teammates could install the Virtual machine on our laptop as we all were having issues with the installation.
2. Being unable to install the software delayed our project timeline, as most of us had to come to the lab to complete their portion of the project.
3. Since nobody had beforehand knowledge of using QNX, it created an obstruction for us to get started using QNX.
4. Time management was the biggest issue for our team. As most of our group-mates were doing their capstone and mini-capstone, we had a difficult time organizing our schedules.
5. Lastly, implementation of threads in C++ was something new for us. We spent hours just to learn how to implement threading in QNX by using C++.

Conclusion

To conclude, the goal of this project has not been fully met since the code compiled however, did not give any outputs. While building the code, it didn’t have any errors or warnings. Our assumption is that there might be errors with the threads and how we implemented them. All the requirements from the project description were implemented in the code. The project was complex and the team met numerous challenges throughout the project including breaking down the project, comprehending and identifying what each class contained, and how to tackle it. The most important issue that we dealt with was receiving a lot of errors and not knowing how to solve them, for example, when we had errors on the Makefile. Another important issue we faced was getting started with the QNX environment. This led to a lot of wasted effort that could've been utilized elsewhere. In the future, it would be advantageous to have a more thorough planning stage for the entire process.

The lessons learned from designing an Air Traffic Control system in QNX, will assist us in getting better life and job opportunities in the future as it has provided experience programming dependable, high security and safe real time systems. Using QNX in automation, and transportation is a winning opportunity for us to achieve our goals. Moreover, using QNX, will assist us to build secure connections which provide isolation and freedom from any sort of hindrance, thus, leading us to create more job opportunities for people of all fields.

References

1. Priya, Bhanu. “Differentiate between shared memory and message passing model in OS.” *Tutorialspoint*, 30 November 2021, https://www.tutorialspoint.com/differentiate-between-shared-memory-and-message-passing-model-in-os. Accessed 6 April 2023.
2. “ About This Guide.” *BlackBerry QNX*, 26 October 2022, http://www.qnx.com/developers/docs/7.0.0/index.html#com.qnx.doc.neutrino.lib_ref/topic/p/pthread_create.html. Accessed 10 April 2023.

Form of Originality