COEN 6341 PROJECT

Real-time system for air traffic monitoring and control

Name: Taneer Hossain Sanjana
Student ID: 40197789
Email: sanjanataneer@gmail.com

Name: Hongyi Chen
Student ID: 40189018
Email: C_ONGYI@live.concordia.ca

Table of Contents

1. Objective:

The safe and efficient management of aircraft movements within airspace is of paramount importance, with Air Traffic Control (ATC) serving as a critical component in the aviation industry. In view of this, the main objective of this project is to design, implement, and test a simplified yet functional real-time ATC system. The system will employ primary and secondary surveillance radars, an operator console for sending commands, a communication system for transmitting commands, and a computer system for computations and alerts. The implementation will utilize periodic tasks, with each aircraft represented as a thread or process, and leverage the QNX RTOS. Through this project, we aim to provide insights into the practical application of real-time systems and the associated design and implementation challenges.

2. Introduction:

2.1 An overview of Air Traffic Control (ATC):

ATC (Air Traffic Control) is a vital system that manages the movement of aircraft in the skies. The air traffic control (ATC) system is responsible for ensuring the safe and efficient flow of air traffic across the world's airspace. At its core, ATC is a complex network of systems, procedures, and personnel that work together to keep air traffic moving safely and smoothly. When you board a plane and take off into the sky, you can thank the air traffic control system for keeping you safe. Air traffic control (ATC) is a critical element of modern aviation, using sophisticated technology and expert personnel to manage the movement of aircraft in the skies.

ATC system involves a network of ground-based controllers who monitor and guide the aircraft from departure to arrival. The ATC system is responsible for maintaining separation between aircraft, ensuring that they don't collide with each other or with obstacles on the ground.

When an aircraft makes its way through the skies, air traffic controllers in the respective area and division monitors its flight. At the moment an aircraft enters a different zone, the traffic controller officer passes this information off to the new division to take control. The air traffic control in the United States is run by the FAA. It has five divisions for its air traffic control system, namely “Air Traffic Control System Command Center”, “Air Route Traffic Control Centers”, “Air Traffic Control Tower”, “Terminal Radar Approach Control (TRACON)” and “Flight Service Station”. [1]

Figure 1 Air Traffic Control Facilities [1]

The ATC system is composed of three main entities: the tower control, approach control or terminal radar control (TRACON), and en-route control. The tower control manages aircraft movement within the airport area and an airspace volume with a radius of 2 to 30 miles and a height of 1,500 to 2,000 feet centered on the airport. The TRACON is responsible for aircraft within a radius of 30-50 nautical miles from the airport and up to 10,000 feet in altitude. Finally, the en-route control handles aircraft movement at higher altitudes and longer distances, from the moment they are "handed-off" to the en-route ATC to the moment they leave the en-route control airspace.

The primary goals of an ATC system are to ensure safe movement and navigation of aircraft in the airspace, maintain reasonable and orderly air traffic flow, and prevent collision between aircraft and obstacles on the operating ground.

For this project, the goal is to implement and analyze a simplified version of an Air Traffic Control (ATC) system for en-route control of aircraft flows in a 3D rectangular airspace that is 15,000ft above sea level. The system should display a plan view of the airspace every 5 seconds, check all aircraft in the airspace for separation constraint violations at a configurable time interval, emit an alarm if a safety violation is found or will happen within 3 minutes, store the airspace in a history file every 30 seconds, and store operator requests and commands in a log file. The subsystems of the ATC system are primary surveillance and secondary surveillance radars, a computer system responsible for computations, an operator console enabling the controller to send commands to the aircraft, and a communication system responsible for the transmission of controller commands to the specified aircraft. The system should also be able to handle

inputs of aircraft details from an input file and produce output files containing information on the history of the airspace, operator requests and commands, and alarms.

2.2 Requirements of the project:

Below are the requirements of the project:

1. A file needs to be created that contains details of each aircraft, such as their ID, location, and speed. This file will be used to simulate the entry of aircraft into the monitored airspace and to show the position of each aircraft every five seconds.
2. Each aircraft will be implemented as a periodic task that updates its location every second and responds to radar requests with its ID, speed, and position. The system will consist of a set of periodic tasks that share a single processor and handle sporadic events using periodic polling.
3. The system will simulate radar operations by communicating directly with each aircraft thread/process to determine their location in the monitored space.
4. The system will consist of a set of periodic tasks that share a single processor and handle sporadic events using periodic polling.
5. The system will check all aircraft in the airspace for separation constraint violations at a specific time in the future (determined by a variable parameter that can be changed at runtime). This will help ensure the safety of the airspace.
6. The system will emit an alarm if a safety violation is found or if a safety violation will happen within 3 minutes. This will alert the operator to take appropriate action in case of a safety issue.
7. The system will store the airspace in a history file every 30 seconds to generate an approximation of the airspace history over time. This will help to analyze the history of the airspace and identify patterns of aircraft movement.
8. The system will store the operator requests and commands in a log file. This will help to keep track of the actions taken by the operator and ensure proper communication between the operator and the system.
9. The system will be tested under various operating conditions to determine system load and measure the execution times of each process. The scheduling feasibility will be tested using rate monotonic fixed-priority assignment. This will help ensure that the system is functioning optimally under different operating conditions.

10. The system must be able to run on the QNX Software Development Platform 7.0 or later version for x86 targets. This is a specific software and hardware requirement that must be met for the system to function properly.

2.3 Assumptions:

There are some potential assumptions that was made during the implementation phase of the project.

One assumption that could be made is that the system is intended for a small or mid-sized airport, where the number of aircraft to be monitored is not too large. This is because the system is expected to store the airspace history every thirty seconds and check for separation constraint violations at a fixed time interval, which can be computationally intensive if there are too many aircraft in the airspace.

It is also said in the specification that, an aircraft enters the considered area flying in a horizontal plane at a constant velocity. So, it can be assumed that for simplicity the X axis in the velocity would be updated.

Another possible assumption in the ATC project is that the scheduling algorithm, such as rate monotonic fixed-priority assignment, will provide sufficient processing resources to meet the system's real-time requirements. Therefore, it is crucial to verify that the scheduling algorithm can offer the necessary computational resources to ensure the system's responsiveness and performance.

Additionally, it could be assumed that the system will be developed and tested on a specific hardware platform or operating system, such as QNX Software Development Platform 7.0 or later version for x86 targets, as mentioned in the requirements. This could limit the system's portability to other hardware or operating systems.

Finally, it could be assumed that the system will have access to sufficient resources, such as processing power and memory, to handle the different components' computational requirements effectively. This could be important in ensuring that the system can perform the required tasks in real-time without significant delays or errors.

3. Analysis:

Based on the given specification and requirements, it is clear that a software-based system is required to monitor and manage the airspace for en-route flights. The system is expected to operate in a real-time environment, which means that it must meet strict timing requirements to ensure that critical tasks are completed within their deadlines. To achieve this, it is recommended to use a Real-Time Operating System (RTOS) such as QNX RTOS, which is specifically designed for real-time systems.

The system will be implemented using periodic tasks, which means that each aircraft will be implemented as a thread that could update its location every 5 second based on its speed and previous location. The system will have a user interface that will allow the air traffic controller to monitor the airspace, and it will display a plan view of the airspace every five seconds, showing the current position of each aircraft.

The timing requirements for the ATC system, such as updating aircraft locations and storing airspace history, may be open to interpretation. However, it is important to define and test these requirements to ensure that the system meets its real-time constraints and operates effectively. Additionally, while the specification mentions a three-minute time window to execute a safety alarm for a safety violation, this could result in multiple violations occurring. To address this, we assume that the system will check for safety violations simultaneously across all aircraft.

The system must be implemented using periodic tasks, which means that each task will execute at a fixed interval. The RTOS will be responsible for managing these tasks and ensuring that they meet their deadlines. QNX provides a wide range of real-time services, including task management, inter-process communication, and memory management, which will be essential for implementing the system.

To manage the airspace, the system will receive an input file containing information about the aircraft entering the en-route monitored area. Each entry in the input file will contain the aircraft's ID, coordinates, speed, and the time it should appear within the boundaries of the area. The system will then simulate the radar operation by communicating directly with the thread/process of each aircraft to learn the presence and location of each aircraft in the monitored space.

The radar operation can be simulated using QNX's message-passing mechanism, which allows threads to communicate with each other efficiently. Each aircraft thread can send its ID, speed, and position to the radar thread, which can then use this information to learn the presence and location of each aircraft in the monitored space.

Furthermore, QNX RTOS provides a file system that can be used to store the airspace history file and the operator requests and commands log file. The file system in QNX RTOS is designed to be reliable and efficient, ensuring that data is stored and retrieved correctly and quickly.

In summary, the system will use a real-time operating system (RTOS) QNX to ensure that critical tasks are executed within strict time constraints. The implementation of the simplified en-route ATC system may be related to real-time operating system theories such as scheduling, synchronization, and communication. These theories are essential in ensuring that the system meets its real-time requirements, such as ensuring

that critical tasks are executed within strict time constraints and ensuring that communication between tasks is efficient and reliable.

4. DESIGN:

Initially, an Object-Oriented Programming (OOP) approach was considered for the implementation of the radar system. OOP seemed like a natural fit for some aspects of the system, for example modeling radar signals as objects with properties and methods. Additionally, different types of radar systems could be easily implemented as subclasses of a base radar class.

However, upon further consideration, it became apparent that an OOP approach may not be the best fit for all aspects of the radar system. In particular, the real-time signal processing involved in an ATC system can be quite computationally intensive and require low-level optimizations. These requirements may not be well-suited to the encapsulation and abstraction provided by OOP.

Furthermore, the system requires multiple processes that need to communicate with each other simultaneously. This can introduce additional complexities when implementing an OOP design. Given these considerations, it was ultimately decided to pursue a different approach for the radar system implementation.

Our proposed system design can be compared with the idea of the GPS time pulse, which acts as the main synchronization source for the system. It ensures that all threads and processes are aware of the current time and can act accordingly. This component is crucial in ensuring that the system operates in real-time and can respond promptly to any changes in the airspace.

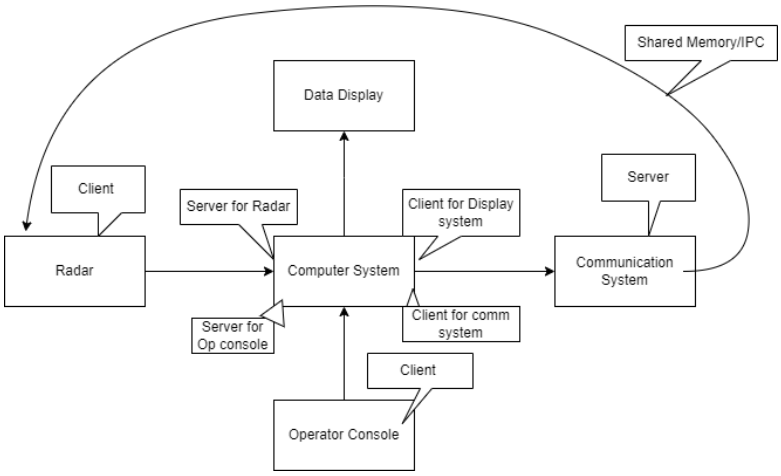

Figure 2 Components of the simplified ATC to be implemented.

In figure 2, the architecture of the ATC system is illustrated, with each component functioning as a client or server-based thread, facilitating interprocess communication between them through message channels and message queues. The use of message queues will allow for efficient and synchronized communication between components, with messages copied directly from one thread's address space to another. The system will utilize multi-threaded architecture, enabling concurrent execution and responsiveness, while client-server-based communication ensures seamless and uninterrupted communication between components.

Below is a brief overview of how each component is designed to be implemented.

Radar Console: The Radar Console is responsible for simulating the movement of aircraft and checking if they are within range of the ATC system. To accomplish this, a thread will be created for each aircraft, which will update its position every time a signal is received. A message channel will be used for sending and receiving messages, while a mutex and condition variable will be used to synchronize threads.

Computer System: The Computer System is responsible for communicating with various components of an aircraft system. To achieve this, it will be a multi-threaded application system. The server will receive input from the Radar Console, Operator Console, and other subsystems. It will process the data and send appropriate display messages to the Cockpit Display Subsystem.

Operator Console: The Operator Console will send commands such as "print aircraft information" or "change velocity," which will be assigned to different types. The message will be sent to the Computer System, which will push the message to the Display System or Communication System, depending on the message type.

Communication system: The Communication System is responsible for sending controller commands to specific aircraft in the Radar Console when the Output Console requests them. The command will be sent through interprocess communication as well.

Display system: The Display System will have different types of enumerators and will act accordingly. For example, if the Operator Console requests specific aircraft information, the Computer System will send a command to the Display System, which will then display the requested information. Additionally, the Display System will show an emergency alert when two aircraft are at a safety violation.

In conclusion, the system architecture includes a radar system that simulates the movement of aircraft and checks for their proximity to the ATC system. The computer system acts as a central hub for the various subsystems of the ATC system, processing data from the radar system, operator console, and other subsystems, and sending appropriate display messages to the display subsystem. The display system will

then display the requested information, including any safety emergency messages, in a clear and concise manner.

5. Implementation:

A brief overview of the project is given below:

Aircraft Information check:

As mentioned earlier in our assumption, that the system is intended for a small or mid-sized airport, where the number of aircraft to be monitored is not too large. So instead of each entry in the input file to be an aircraft, we are testing the aircraft by randomly generating the values in the Radar system, the number of aircrafts can be changed, and the multiple threads will be created accordingly.

As we also assumed earlier that this simplified model tends to reduce the complexity, so we calculated accordingly to randomly generate the aircraft positions, according to the airspace tracking range. But it's also implemented to check if it's under the tracking range, which is assumed to be 100000 ft x 100000 ft in the horizontal plane and 25000 ft in height.

In the **init_Aircrafts()** function, the code initializes the aircraft positions and velocities randomly. Here's a more detailed explanation of how the random values is generated for each aircraft:

The **position** of the aircraft is represented by a three-element array that stores the X, Y, and Z coordinates of the aircraft in feet. The **rand()** function is used to generate a random number between 0 and 99,999 (which is the size of the airspace in the horizontal plane) for the X and Y coordinates. The **rand()** function is also used to generate a random number between 15,000 and 39,999 (which is the range of altitude for the airspace) for the Z coordinate. The resulting values are stored in the **position** array.

The **velocity** of the aircraft is represented by a three-element array that stores the velocity in the X, Y, and Z directions in feet per second. The **rand()** function is used to generate a random number between 0 and 99 for the X velocity, and between 0 and 1 for the Y and Z velocities. The resulting values are stored in the **velocity** array.

For updating the new aircraft information, it takes a pointer to an Aircraft_t structure as an argument and updates the position of the aircraft by adding its current velocity multiplied by a time interval of 5 seconds. It then checks whether the aircraft is within the range of the ATC (Air Traffic Control) tracking system by checking its position in 3D space. If the aircraft is out of range, the in_ATC_tracking_range flag is set to 0,

indicating that it is no longer being tracked by the ATC and it is removed from the simulation. It will show a message too in that case.

Note that these random values may not be realistic for an aircraft's position and velocity, and the initialization logic depends on the requirements of the specific application.

Radar system:

In a Radar system, aircraft information and positions are constantly updated, and this information needs to be sent to the computer system through a message channel. To handle this, threads are created for each aircraft, with **'NUM_THREADS'** representing the number of aircrafts. This can be modified based on the system's load.

To ensure that all threads are synchronized, a timer is set up to periodically update the state of the aircrafts. When a signal is received, the **signal_handler()** function is called, which updates the aircraft state and broadcasts a signal to wake up all waiting threads. The **event_flag** variable is used to notify the threads, and it is protected by a mutex variable to prevent race conditions.

Once the aircraft's state is updated, the thread sends the current state of the aircraft to a message queue. The main thread listens for signals, and when an event flag is set, all threads send their aircraft information to the message queue. The **thread_radar_callback_func** is responsible for receiving these messages, printing the received aircraft information, and then sending it to the computer system.

The **signal_handler()** function increments a counter variable every time it is called, which is printed to indicate that an event has occurred. Finally, the **event_flag** variable is decremented, and the mutex lock is released.

In summary, the program uses signals, threads, and message queues to update and send aircraft information to the computer system. The program ensures synchronization among the threads, and mutex variables are used to protect critical sections. The **thread_radar_callback_func** function processes received messages and sends the information to the computer system.

Operator console:

In this process, the main function was "message passing" for communication between the client and the server. The client (operator console) sends messages to the server (computer system) using the **MsgSend** function, while the server receives messages using the **MsgReceive** function.

The program defines a **RequestType_t** enum that represents the different types of requests that the server can receive from the client. These include requests to display aircraft data, requests to update aircraft data, and requests to exit the program.

The **opMsg_t** struct is used to encapsulate the different types of messages that can be sent between the client and server. This struct contains a header that identifies the type and subtype of the message, as well as a payload that contains the actual request data.

The **consoleRequest_t** struct is used to represent the payload of the **opMsg_t** struct when a request is made to display aircraft data. This struct contains the **RequestType_t** enum value, as well as an array of parameters that define the specific aircraft data to display.

The program also includes an **_print_help()** function that displays usage information to the operator, and an **_parse_operator_input()** function that parses the operator's input and returns a **RequestType_t** value and an array of parameters that describe the requested operation.

Finally, the **main()** function creates a client process that interacts with the operator and sends messages to the server process. When the client process receives an "exit" command from the operator, it closes the connection to the server and exits.

Display system:

This process manages the display of information related to aircrafts. It includes functions for printing aircraft information and route alarms, which will be running concurrently in different threads. To ensure that the alarm message vector is reset periodically and threads waiting on the condition variable are notified of the change, a global signal is used. This is particularly useful in situations where multiple threads are processing data from a shared vector and need to prevent race conditions or data inconsistencies.

To accomplish this, a timer is created for the alarm display to be reset every 10 seconds. The display system receives messages from the computer system via the "display_msg_channel" attach point, which is a command from the operator console. The requested aircraft information will be printed accordingly and stored in a log file named "opreq_logfile_fd".

Additionally, the display system receives messages from the "display_alarm_msg_channel" attach point, which triggers the printing of safety violation alerts between two aircrafts. This ensures that any potential threats to aircraft safety are immediately communicated to relevant personnel.

Computer System:

This is the heart of the whole system. The computer system receives aircraft positions and velocities, commands from various clients and logs them in a file. It also performs a validation check to see if any aircraft violate a specific distance constraint and sends an alarm message to the appropriate client if it does.

Run_Constraint_Validation that takes in some input parameters and returns a value of the enumerated type **ConstraintViolationType_t**. The purpose of this function is to check if any aircraft in the given array of **aircraft_log** violates the distance constraint defined by the constant **DISTANCE_CONSTRAINT_FEET**.

If the present distance between any two aircraft is less than **DISTANCE_CONSTRAINT_FEET**, the function sets the **relative_distance_feet** variable to the distance between the aircraft and populates the **idsFailedConstraints** array with the IDs of the two violating aircraft. Finally, the function returns **CONSTRAINT_VOLTATION_PRESENT_e** indicating that there is a constraint violation.

The **_update_most_recent_info** function updates the information of the most recent position and velocity of an aircraft. It checks if the aircraft is already being tracked by searching through a list of tracked aircraft IDs and updates the information if the aircraft is found. If the aircraft is not being tracked and there is space to track it, it is added to the list. If the aircraft is not found and there is no space to track more aircraft, a warning message is printed.

```
static void _update_most_recent_info(const Aircraft_t aircraftInfo)
{
    int isAircraftFound = 0;

    for(int i = 0; i < MAX_AIRCRAFT_TO_TRACK; i++)
    {
        if(aircraftInfo.id == most_recent_aircraft_log[i].id)
        {
            isAircraftFound = 1;
            memcpy(&most_recent_aircraft_log[i], &aircraftInfo, sizeof(Aircraft_t));
            break;
        }
    } // for(int i = 0; i < MAX_AIRCRAFT_TO_TRACK; i++)

    if(isAircraftFound == 0) // means we need to add it to the list
    {
        if(nAircraftsBeingTracked < MAX_AIRCRAFT_TO_TRACK)
        {
            memcpy(&most_recent_aircraft_log[nAircraftsBeingTracked], &aircraftInfo, sizeof(Aircraft_t));
            nAircraftsBeingTracked++;
        }
        else
        {
            printf("[WARNING]: Maximum number of aircraft being tracked has been reached! aircraftInfo.id = %u is not \n");
        }
```

Figure 3 _update_ most_recent_info function

The **_get_most_recent_aircraft_info** function retrieves the most recent information about an aircraft with a given ID. It searches through the list of tracked aircraft IDs to find the aircraft with the given ID and returns its information if it is found. If the aircraft is not found, a warning message is printed, and the function returns an empty aircraft information structure.

```
static void _get_most_recent_aircraft_info(const uint32_t aircraftIdToSearch, Aircraft_t* aircraftInfo)
{
    int isAircraftFound = 0;

    for(int i = 0; i < MAX_AIRCRAFT_TO_TRACK; i++)
    {
        if(aircraftIdToSearch == most_recent_aircraft_log[i].id)
        {
            isAircraftFound = 1;
            memcpy(aircraftInfo, &most_recent_aircraft_log[i], sizeof(Aircraft_t));
            break;
        }
    } // for(int i = 0; i < MAX_AIRCRAFT_TO_TRACK; i++)

    if(isAircraftFound == 0) // means we need to add it to the list
    {
        printf("[WARNING]: The request aircraft id %d is not found in the tracking log \n");
        aircraftInfo->id = 0;
        fprintf(logfile_fd, "[WARNING]: The requested aircraft ID %d is not found in the tracking log\n", aircraftIdToSearch);

    }
}
```

Figure 4 _get_most_recent_aircraft_info function

At the end, this process creates a thread that listens for messages from the radar system. Whenever a new message is received, the program updates the most recent information about the corresponding aircraft and prints out the updated information. The program also initializes message queues for communication and creates several channels to communicate with other components of the system, including an operator message channel, communication subsystems and a display message channel.

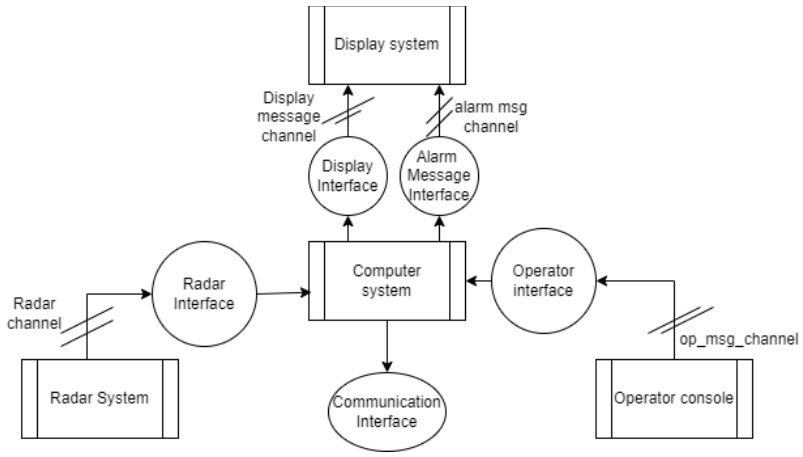

Figure 5 Computer System

Figure 5 represents a brief overview of computer system. The circle represents threads and message channels are connected accordingly, where message queues are sent per request.

The requirements implemented, partly implemented, and not implemented:

Our aircraft tracking system was designed with a set of clear requirements, and we are pleased to report that almost all of them were successfully implemented. Each aircraft was implemented as a periodic task and updated its location every second based on its speed and previous location. Furthermore, we developed a function to handle operator requests for information about a particular aircraft's ID, speed, and position. Additionally, we thoroughly tested our system under various operating conditions, including low, medium, high, and overload scenarios.

While we were unable to fully implement the requirement to inject aircraft information from an input file to the radar system, we simulated the aircraft information directly in the radar system. We believe that this approach still provides an accurate representation of how aircrafts are typically tracked in real-world scenarios, as aircraft positions are often simulated randomly. Also at present, the code only checks for present constraint violations. However, the code contains a **TODO** comment, indicating that future violations will be checked for in the future.

The requirements specified the use of rate monotonic fixed-priority assignment One common algorithm used in QNX is the sporadic (SCHED_SPORADIC) scheduling policy, which limits the execution time of a thread within a given period of time. This is particularly useful in systems when Rate Monotonic Analysis (RMA) is being performed for both periodic and aperiodic events, as it allows a thread to service aperiodic events without compromising the hard deadlines of other threads or processes. [3] While our team did not have extensive knowledge of this algorithm, we opted to use round-robin scheduling instead. In this algorithm, a task runs until it consumes a specified amount of time (a time slice) or gets blocked, after which it goes to the tail of the READY queue. This simplified version of a kernel data structure consists of a queue with one entry per priority, and each entry consists of another queue of the threads that are READY at the priority. [4]

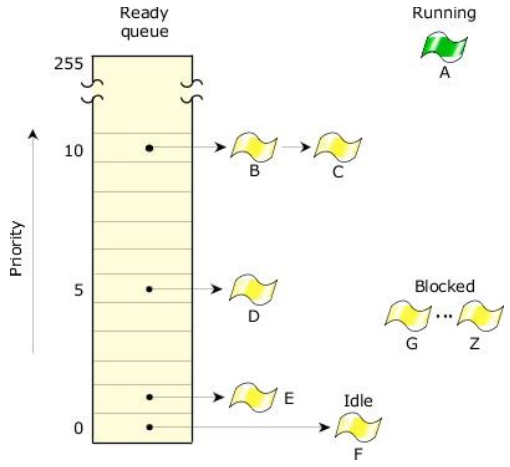

Figure 6 The ready queue for five threads.[2]

Although we used round-robin scheduling in our ATC system. Round-robin scheduling allows a task to run for a specified amount of time or until it becomes blocked, and then it goes to the end of the ready queue. In our system, this allowed for concurrent updates whenever an event occurred. While sporadic scheduling may be more appropriate for systems with both periodic and aperiodic events, round-robin scheduling was sufficient for our ATC system.

Unfortunately, we were unable to implement the communication process as specified in our requirements. Our system did include a thread in the computer system that would have facilitated communication through a message channel. However, we were not able to complete the implementation of a request system that would allow operators to change the velocity or position of aircrafts, which would have been sent to the communication system accordingly. Despite this, we remain confident in the functionality of our system and believe that it meets the needs of its intended users.

5.1. Problems encountered and difficulties faced:

- One of the first concern of the project was to monitor the input and output simultaneously.

  From our POD session, it was suggested to use WinSCP. WinSCP is a powerful and efficient file transfer tool that enables users to monitor the input and output of multiple processes running on different operating systems. In the case of connecting to a QNX RTOS through a virtual machine, WinSCP allows users to transfer files between the QNX RTOS and the Windows host system.

  The ability to monitor simultaneous input and output is particularly useful for testing and debugging concurrent processes running on the QNX RTOS. With WinSCP, users can easily transfer files between the QNX system and the Windows host, allowing them to quickly make changes and test the effects on the QNX processes.

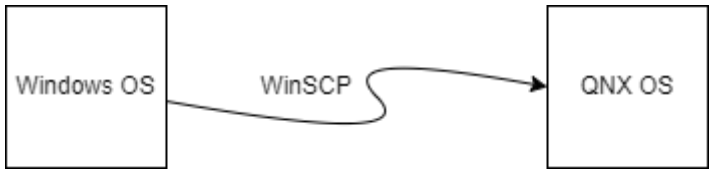

Figure 7 Using WinSCP application.

- The implementation of an Air Traffic Control (ATC) system requires a reliable and efficient Real-Time Operating System (RTOS). However, as with any complex system, issues may arise during the development and implementation phases. Our team encountered a particular difficulty with the QNX RTOS system, which pertained to how it packed data structures.

Specifically, the problematic data structure was the struct _pulse, which consisted of 16 bytes of data. When we attempted to embed this structure inside another data structure with a size of only 4 bytes, it failed to pack correctly. As a result, the system encountered a runtime error.

```
struct _pulse {
    _Uint16t                    type;
    _Uint16t                    subtype;
    _Int8t                      code;
    _Uint8t                     zero[3];
    union sigval                value;
    _Int32t                     scoid;
};
```

Figure 8 struct _pulse in QNX System

Data structure packing is an essential process that optimizes the use of memory by arranging data in a way that minimizes its memory footprint. The compiler performs this task by arranging the data in memory in an efficient manner. However, in the case of the struct _pulse, the compiler failed to pack it correctly into a 4-byte data structure, leading to the runtime error.

After investigating the issue, we discovered that the problem was with the way the QNX RTOS system packs data structures. We decided to solve the problem by creating our own data structure that would work properly with the RTOS system. By doing this, we were able to avoid the runtime errors and continue with our ATC project without any further issues.

In the original code, there was a check for the type and subtype of the received message using the fields **type** and **subtype** of the **struct _pulse** in the commented out lines. [fig 9]

```
//if (msg.hdr.type == 0x00)
{
   //if (msg.hdr.subtype == 0x01)
   {
      /* A message (presumable ours) received, handle */
      //printf("Server receive %d \n", msg.data);
      //printf("\nDISPLAYSYS [ALARM]: Received display message type: %d\n", (int)msg.displayInfo.displayType);
      //
      // TO DO: Implement how to print for different display types
      //
      switch(displayInfo.displayType)
      {

      case DISPLAY_TYPE_ROUTE_ALARM_PRESENT:
          _print_route_alarm_present(displayInfo.aircraftIdsForAlarm[0], displayInfo.aircraftIdsForAlarm[1], displayInfo.rel_d:
          break;

      case DISPLAY_TYPE_ROUTE_ALARM_FUTURE:
          _print_route_alarm_for_future_violation(displayInfo.aircraftIdsForAlarm[0], displayInfo.aircraftIdsForAlarm[1], displ
          break;

      default:
          break;
      } // End of switch(msg.displayInfo.displayType)

   }
}

MsgReply(rcvid, EOK, 0, 0);
```

Figure 9 Problem with RTOS Packing

This check was done to ensure that the received message was of the expected type and subtype before handling it. However, in the updated code, the check was removed, and instead, the switch-case statement handles messages based on their **displayType** field.

- During the implementation of our Air Traffic Control (ATC) system, we encountered another significant issue with the display alarm mechanism. The display alarm was designed to notify the operator of any safety violations or threats posed by aircraft movements. However, the alarm would repeatedly trigger each time new aircraft information was updated in the system, leading to the display of the same warning messages repeatedly.

  To address this issue, we devised a solution in the form of a timer. This timer was programmed to clear the alarm vector at regular intervals of 10 seconds, thus preventing the repeated display of redundant warning messages. With this mechanism in place, the operator was able to focus on new and important safety threats that required immediate attention, without being unnecessarily distracted by repetitive warnings. The implementation of this timer not only enhanced the effectiveness of the ATC system but also contributed to the overall safety of the airspace.

5.2. Result:

Below is the screenshot of the implemented consoles:

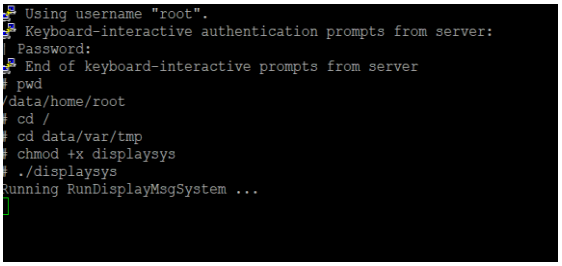

Figure 10 Display system console

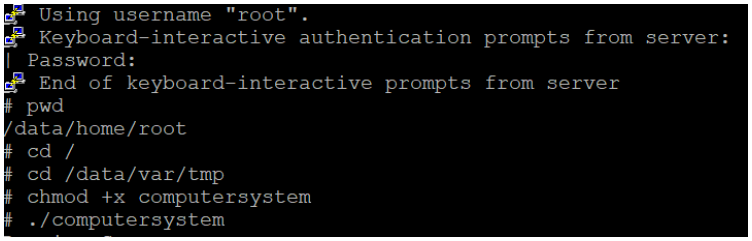

Figure 11 Computer System console

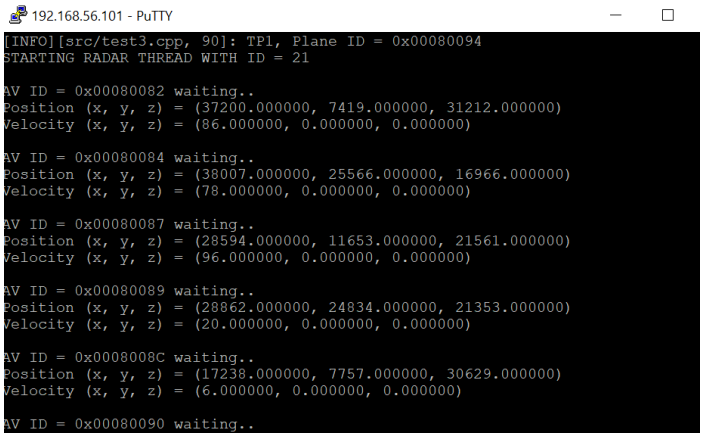

Figure 12 Radar system console

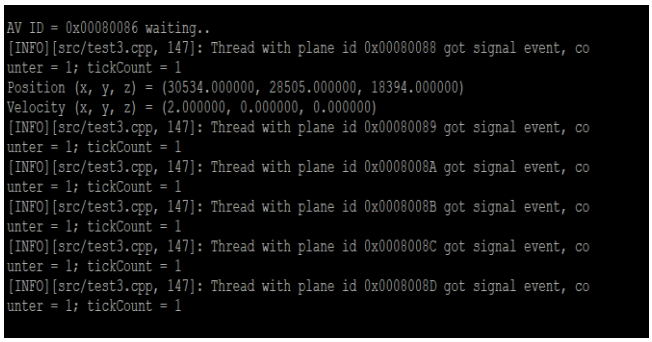

Figure 13 Radar system console

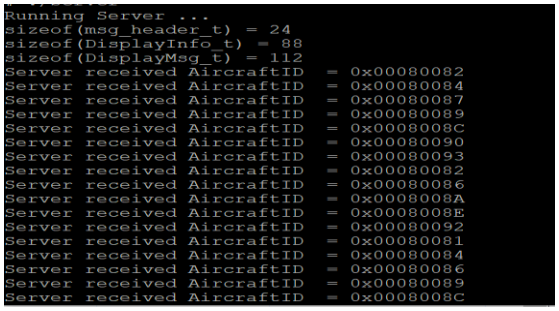

Figure 14 Computer system console

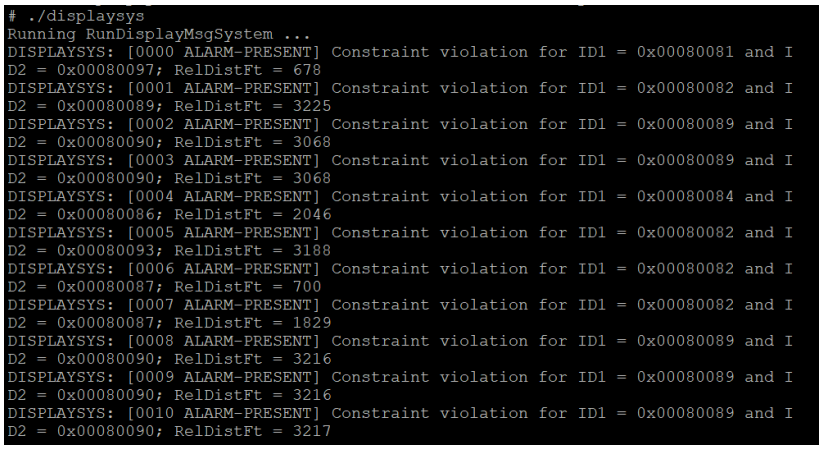

Figure 15 display system console

Figure 16 Operator input console

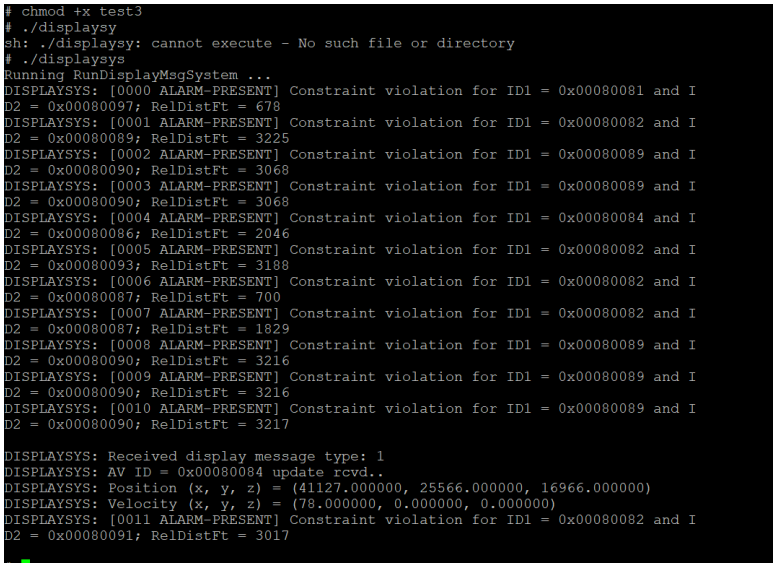

Figure 17 Display system console

The implementation involves the activation of both the display system and the computer system to ensure that they are prepared to receive aircraft information, as depicted in Figures 10 and 11. The radar console continuously monitors the airspace, awaiting updates every 5 seconds, as shown in Figures 12 and 13. When an aircraft comes within tracking range, the console sends the information to the computer system, which acknowledges receipt as shown in Figure 14. Meanwhile, the display system detects any constraint violations between two aircraft systems, displaying their IDs and relative distance, with updates occurring every 10 seconds [figure 15]. The operator console, as depicted in Figure 16, allows specific aircraft ID information to be requested, which is then displayed on the console of the display system, as shown in Figure 17. All these processes will be run concurrently.

As mentioned earlier, the round-robin scheduling algorithm was used to facilitate concurrent execution of processes in our ATC system due to its efficient execution time. The figures below illustrate the execution time for each process in both normal load conditions, which uses the profile timing tool of the QNX system.

Execution Time ×

displaysys #3 - Threads Tree

Name	Deep Time		Shallow Time		Count	Location
[89.74%] Thread 1	471.064 s	89.74%	471.064 s		1	
[<0.01%] _init_array	2.014 us				1	_initfini.c:32
[89.74%] _start	471.064 s				1	crt1.S:28
[<0.01%] Thread 2	0.124 ms	<0.01%	0.124 ms		1	
[10.26%] Thread 3	53.861 s	10.26%	53.861 s		1	

Figure 18 Execution time of display system

Execution Time ×

computersystem #4 - Threads Tree

Name	Deep Time		Shallow Time		Count	Location
[<0.01%] Thread 1	2.201 ms	<0.01%	2.201 ms		1	
[29.06%] Thread 2	76.645 s	29.06%	76.645 s		1	
[<0.01%] Thread 3	60.326 us	<0.01%	60.326 us		1	
[<0.01%] Thread 4	0.603 ms	<0.01%	0.603 ms		1	
[<0.01%] Thread 5	0.219 ms	<0.01%	0.219 ms		1	
[70.94%] Thread 6	187.065 s	70.94%	187.065 s		1	

Figure 19 Execution time of computer system

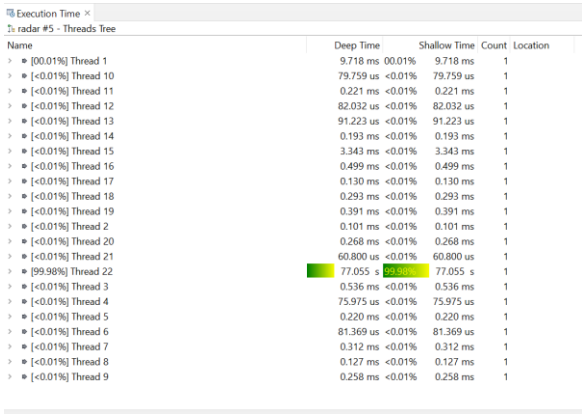

Figure 20 Execution time of Radar system

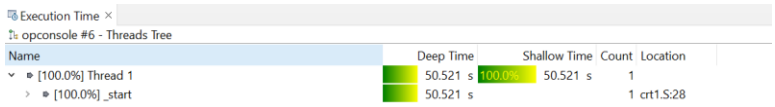

Figure 21 Execution time of Operator console

5.3. System under various operating conditions:

In the field of air traffic control, it is crucial to monitor the degree of congestion in the airspace and the associated IO traffic. As the airspace physics is calculated in a simple way, the number of aircraft being tracked serves as a key indicator for assessing the level of congestion in the airspace. To this end, we have conducted experiments to test our system's performance under different loads. Specifically, we have simulated underload, medium load, high load, and overload scenarios, with 2, 50, 200, and 1000 aircraft being tracked, respectively. Our system is designed to track a maximum of 500 aircraft, as indicated by the global variable MAX_AIRCRAFT_TO_TRACK.

It is important to evaluate how the system resources are utilized under different loads. Under low load conditions, the system is less resource intensive as fewer aircraft need to be tracked, and there is a lower likelihood of safety violations. In contrast, high load scenarios demand greater utilization of system resources, as there is an increased probability of constraint violations. This results in busy processes that are occupied for extended periods. In the case of overload, access to information becomes challenging. Our system will display a warning message in such scenarios, informing the user that the maximum number of aircraft that can be tracked has been reached.

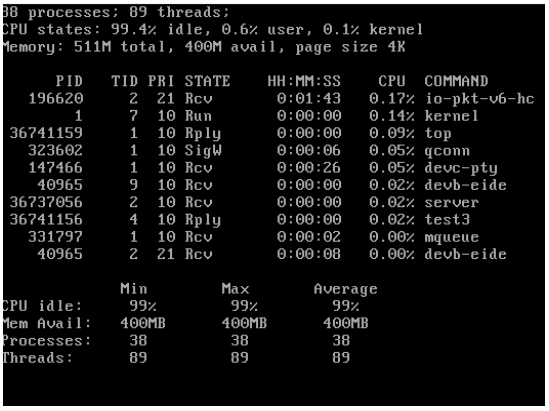

Figure 22 System under low load

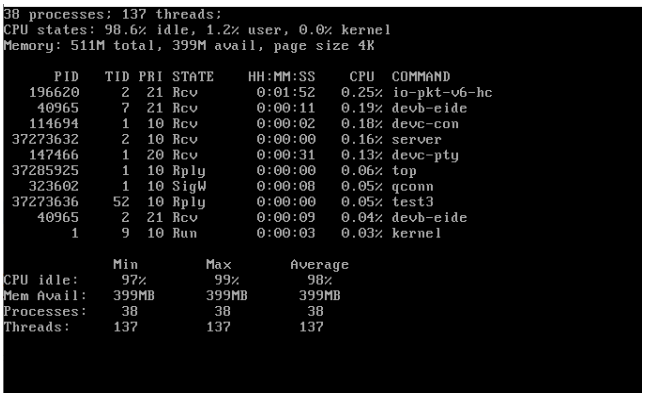

Figure 23 System under medium load

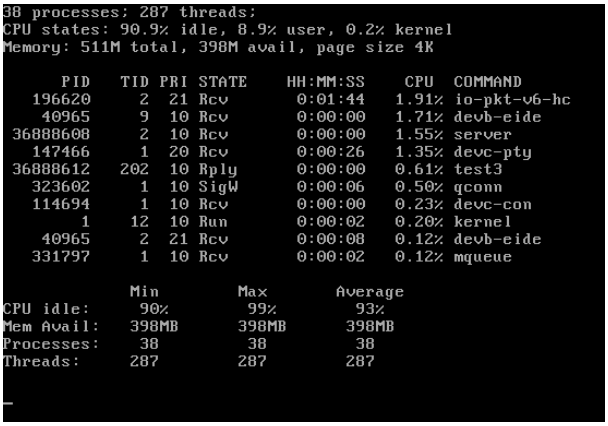

Figure 24 System under high load

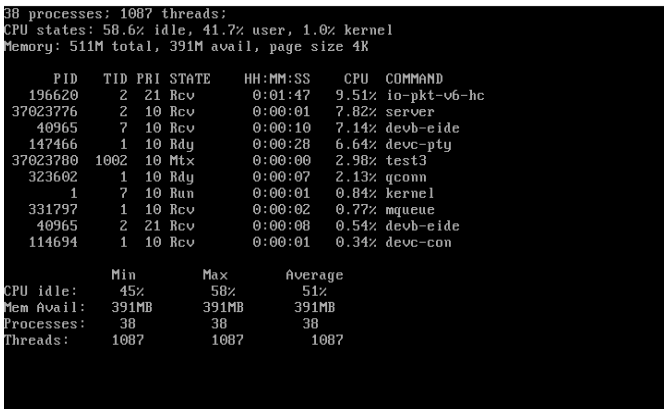

Figure 25 System under overload

In the above figures, the test results are summarized in terms of CPU states, which show the percentage of time and other resources the CPU spends in different modes of operation. The idle state indicates that the CPU is not performing any tasks, while the user and kernel states refer to the execution of user-level and kernel-level code, respectively.

The tests were carried out for underload, medium load, high load, and overload scenarios. The results of the tests revealed that the CPU utilization varied depending on the load, with idle-min ranging from 99% for underload to 45% for overload. Additionally, the average CPU utilization ranged from 99% for underload to 51% for overload. The tests also indicated that the user-level CPU utilization increased as the load increased, with values ranging from 0.6% for underload to 41.7% for overload. Moreover, the tests revealed that the system memory available decreased from 400 MB for underload to 391 MB for overload.

These results suggest that as the load on the system increases, the CPU and memory utilization also increase, which could affect the system's performance and efficiency.

6. Lesson learned:

At the starting of the project, our team had beginner knowledge at Real Time Operating System (RTOS). As a beginner in RTOS, learning about all these complex concepts was overwhelming. But during the course of this project, we gained a deeper understanding of several important concepts in real-time operating systems.

1. ***Real-time operating systems:*** We have gained a better understanding of the capabilities and limitations of real-time operating systems, including how they differ from general-purpose operating systems and how they support tasks scheduling and interprocess communication.

2. ***Interprocess communication (IPC):*** This was used extensively in the project to facilitate communication between the different processes running on the system. IPC mechanisms such as message queues and shared memory were used to exchange data between the different components of the system. Through this project, we learned about the different types of IPC mechanisms, their advantages and disadvantages, and how to implement them.

3. ***Shared memory:*** This was used to share data between different processes in the system. In the project, shared memory was used to store aircraft data, which was then read by the radar component. Through this project, we learned how to create and manage shared memory segments, how to read and write to shared memory, and how to synchronize access to shared memory.

4. ***Thread and thread management:*** Thread and thread management were critical to the success of this project. Thus, threads were used extensively in the project to implement different components of the system. For example, each aircraft was implemented as a thread that periodically updated its location. The operator console also used threads to manage the user interface. Through this project, we learned about thread creation, synchronization, and management, including how to create, start, stop, and synchronize threads. We specifically learned how to use POSIX threads to implement this mechanism.

5. ***Timers and periodic tasks:*** We have gained experience with using timers and periodic tasks to schedule and execute recurring operations, such as checking for airspace violations in the air traffic control system.

6. ***Resource sharing:*** Resource sharing was also an essential concept that we learned about in this project. As multiple threads accessed shared resources such as shared memory and message queues, it was essential to use synchronization mechanisms such as mutexes and semaphores to ensure that only one thread accessed the shared resource at a time. We learned how to use POSIX synchronization mechanisms to implement this mechanism.

7. ***Message passing:*** This was used to send and receive messages between processes in the system. For example, the communication system used message queues to send commands to the aircraft. Through this project, we learned about message passing and how to implement message queues in a real-time system.

8. ***Performance optimization:*** Real-time systems require high levels of performance and efficiency to ensure that they respond quickly and predictably to system events. We have learned about techniques for optimizing the performance of software, including profiling, benchmarking, and optimization strategies.
9. ***Using QNX IDE:*** QNX Software Development Platform 7.0 was used to develop the system. Through this project, we learned about the QNX IDE and its features, including how to create projects, build and debug applications, and use different tools and APIs provided by the IDE.

As we reflect on the ATC project, we recognize the need to explore the QNX system more extensively to enhance our understanding and troubleshoot any potential issues that may arise. Specifically, we encountered a problem with structure packing in the QNX RTOS, which required us to make modifications to our rightly written code. This issue shed light on the outdated and buggy nature of the system and motivates us to delve deeper into its inner workings.

Moreover, we aim to add a graphical system that displays the movement of planes on a projection screen. This feature would provide a more intuitive and user-friendly interface for the ATC system, enabling controllers to better track the movement of planes and respond to any safety concerns in a timely manner.

Overall, this project provided us with a valuable opportunity to gain hands-on experience with real-time operating systems and to deepen our understanding of the key concepts and techniques involved in developing complex, multi-component systems.

7. Conclusion:

The ATC project was implemented successfully with most of the requirements being met. The system was tested under various operating conditions, and it was observed that the system used more resources during high load and overload conditions, leading to a decrease in available memory. The most significant concepts addressed during the project include implementing aircraft as periodic tasks, updating their location, and handling operator requests, sharing a single processor among all processes or threads, and using a simulated radar system to track aircraft positions.

The primary issue that arose was the communication process, which was not fully implemented as per the specifications. Despite this, the project demonstrated the feasibility of implementing an ATC system capable of tracking multiple aircraft in real-time and efficiently handling operator requests.

In conclusion, the ATC project was a successful implementation of a crucial real-world system. While there were some limitations and areas for improvement, the system developed could be a valuable tool for managing air traffic in a real-world scenario.

7.1. Future work:

Future work could include further optimization of the system to improve resource usage during high load conditions, implementing the communication process as per specifications, and adding additional features such as collision avoidance and weather monitoring. The use of more advanced technology, such as machine learning, could also be explored to improve the accuracy and efficiency of the system.

Another potential improvement could be the addition of redundancy and fault-tolerant mechanisms to ensure the system's reliability in case of hardware or software failures. Additionally, exploring the integration of unmanned aerial vehicles (UAVs) into the airspace management system would be an interesting area of topic. This could involve developing algorithms for coordinating the flight paths of UAVs with those of manned aircraft, as well as designing new user interfaces for the air traffic controller to manage the additional complexity introduced by UAVs.

Furthermore, investigating the potential benefits of using advanced communication technologies, such as 5G or satellite-based systems, for improving the efficiency and safety of the airspace management system would also be a worthwhile area of future work.

8. Individual Contribution:

Taneer contributes major part of the reporting and programming of the radar, displaysys, opconsole, and computer system part of the code. Hongyi contributes helping Taneer in programming and making the report with the above part and applied communication system to the system, handling the operator’s commands, as well as adjusting the rest of the code.

9. References:

[1] J. Grabianowski, "How Air Traffic Control Works," HowStuffWorks, [Online]. Available: https://science.howstuffworks.com/transport/flight/modern/air-traffic-control.htm#:~:text=An%20airplane's%20transponder%20transmits%20flight,safe%20distances%20between%20ascending%20aircraft. [Accessed: Apr. 9, 2023].

[2] QNX Software Systems. "Ready Queue - QNX Neutrino RTOS Programmer's Guide". [Online]. Available:

http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Foverview_Ready_queue.html. [Accessed: April 8, 2023].

[3] QNX Software Systems. "Sporadic Scheduling - QNX Neutrino RTOS Programmer's Guide". [Online]. Available: http://www.qnx.com/developers/docs/qnxcar2/index.jsp?topic=%2Fcom.qnx.doc.neutrino.prog%2Ftopic%2Foverview_Sporadic_scheduling.html. [Accessed: Apr. 8, 2023].

[4] J. Schaffer and S. Reid, "The Joy of Scheduling," QNX Software Systems, Waterloo, ON, Canada, Tech. Rep. 2001. [Online]. Available: http://www.qnx.com/content/dam/qnx/whitepapers/2011/qnx_joy_of_scheduling.pdf. [Accessed: Apr. 8, 2023].