Concordia University

Report presented to

M. Rodolfo Coutinho

As a requirement for

Introduction to Real-Time Systems (COEN 320 – S)

By

Alexandre Vallières (40157223 – alexandre.vallieres@mail.concordia.ca)

Samson Kaller (40136815 – samson.kaller@gmail.com)

Adnan Saab (40075504 – adnan.9821@gmail.com)

Mohammed Al-Taie (40097284 – altaiem888@gmail.com)

Real-Time System for Air Traffic Monitoring and Control

Sunday December 4th, 2022

Objective

The project is aimed towards the design, implementation, testing and analysis of real-time systems by working with a simplified version of an air traffic monitoring and control (ATC) system. It aims to simulate the communication between all the components of a real-time system as well as their internal periodic, aperiodic and sporadic task management based on the given design specifications. The project makes use of the QNX real-time operating system (OS) to compile and runs a C++ code that uses threads to simulate every part of the simplified ATC system, including the airplanes.

Introduction

Theory

Air Traffic Monitoring and Control (ATC)

In real-life applications, ATC systems are divided into 3 volumes: tower control area, terminal radar control (TRACON) area and en-route area. An airplane starts at the tower control area when it is inside the airport's vicinity and enters the TRACON area after leaving it. After reaching a certain altitude and distance, it enters the en-route area, which a generally extremely large three-dimensional area high up in the sky where airplanes usually get to their cruising altitude and speed. This project mainly focuses on the latter area and must monitor the air traffic in order to prevent collisions.

As this is a simplified version of a real ATC system, it will monitor a volume of a certain size and at a certain high. It is separated in 5 different components, as shown in the following figure:

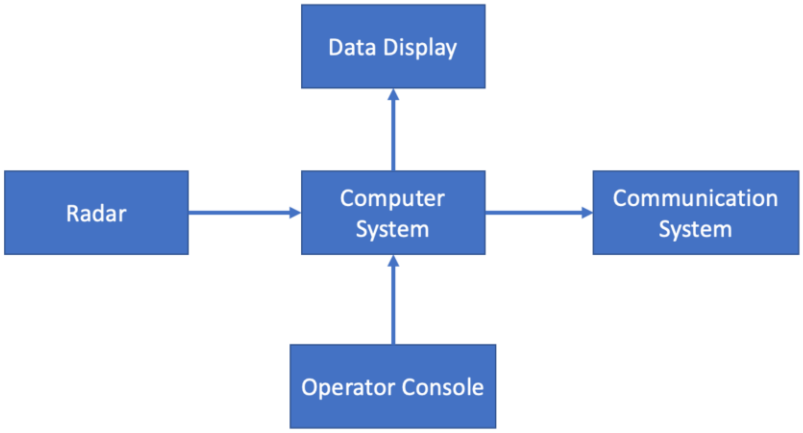

Figure 1: ATC system components

What is important to notice if that everything passes through the central computer system, and this is the component that will be responsible for every logical operation that happens on the system. As a very general explanation of the system, the radar system periodically scans the airspace in a rotary fashion. Its primary surveillance radar (PSR) emits ultrahigh-frequency radio waves that outputs the objects it comes in contact with on the screen as dots placed according to the distance between them and the center of

the radar. Its secondary surveillance radar (SSR) then emits an interrogation signal asking all the planes that its PSR came in contact with to respond with their flight ID, flight level, speed and position. Once it receives everything, it sends all the data to the computer system, which then passes it on to the data display component for the human operator to be able to read it. The computer system also analyses this data for incoming collisions within a certain interval and notifies the operator with an alert on the display system. The human operator has a console to input commands to send to individual airplanes, which also pass through the computer system before being sent to said airplanes through the communication system that receives the commands from the computer.

QNX Operating System

QNX Neutrino is a UNIX-based proprietary operating system made by Blackberry that can run on a virtual machine and is created via a virtual disk image. The QNX Momentics IDE uses it as its target onto which it compiles and runs C and C++ applications. The system itself is very minimalist as it only allows for console interactions and comes with a very small amount of base storage, which cannot be modified when created from the IDE itself. Its primary purpose is to design, run and simulate real-time system applications, which is exactly what this project involves. Although the overall QNX development tools leave very much to be desired, they are the closest one could get to the operating system of a real-life application and is thus used to simulate as accurately the conditions in which the designed application will run.

Requirements

The following subsections list all the design requirements based on the project outline:

Functional

1. The system must monitor a 3D en-route airspace in the shape of a rectangular prism that consists of 100,000 x 100,000 feet on the horizontal plane and 25,000 feet on vertical plane, which stands 15,000 feet above ground level.
2. Airplanes must have a minimum distance of 1,000 feet vertically and 3,000 feet horizontally between them, otherwise it is considered as a separation violation.
3. The computer system must periodically check all airplanes in the airspace for separation violations at current_time + n seconds, where n is an integer parameter that can be changed by the operator at runtime.
4. The computer system must emit a sonorous and/or visual alarm to the display system alerting the operator of any separation violations that will happen in the next 3 minutes.
5. The computer system must store the airspace in a history file every 30 seconds with enough information to generate an approximation of the history of the airspace over time.
6. The computer system must store all operator requests and commands in a log file.
7. The system must display a plane view of the airspace every 5 seconds showing the position of every aircraft.
8. The operator console must allow the operator to send commands to an aircraft to change its speed, altitude and/or position, or request for augmented information of that plane to be displayed on the screen.

Non-functional

9. The airplanes must enter the airspace in a horizontal plane at a constant velocity and altitude.
10. The airplanes must maintain their speed and altitude unless directed by the operator to do so.
11. Every operator command or request must first be passed from the console to the computer, and then to the communication system before being sent to the plane.
12. An input file must be used by the system as a list of airplanes entering the airspace, containing the time of entry, the flight ID, the initial X, Y and Z position, as well as the initial X, Y and Z speed of every aircraft.
13. Each aircraft must be implemented as a process or thread that updates its location every second while listening for radar requests and answering them.
14. The radar must learn about each airplane in the system by communicating directly with their respective thread or process.
15. The active part of the system must consist of a set of periodic tasks using periodic polling to handle specific sporadic events.
16. The system should be tested under low, medium, high and overload conditions determined by the number of airplanes in the space, the degree of congestion and the amount of IO traffic.

Assumptions

Based on the aforementioned requirements, there has been a few assumptions that needed to be made since some requirements are overlapping each other and some also have somewhat confusing wording.

The first assumption was made from requirement 7. Since a plane view display implies that the computer system must have access to the current airspace data and it is a periodic task that happens every 5 second, we assumed that this task was to be synced with the radar gathering data from the plane periodically. This means that the current flow of data every 5 seconds would start from the radar system gathering information about the airspace, that would then be passed along to the computer system that also passes it on to the data display system. Based on this, it would also make sense that the computer system takes this opportunity of fresh data to check for airspace violations.

However, the overall process of detection of violations becomes a little bit fuzzy when examining requirements 3 and 4. The thought process was long in order to analyze how to interpret them, but our team concluded that requirement 4 should be considered as a boundary stating that any violations that would happen later than in 3 minutes would not be considered. Since requirement 3 is very open to interpretation, we based ours on our existing assumptions of requirements 7 and 4 to conclude that every time the computer system checks for violations based on the data from the radar coming every 5 seconds, it should consider violations that will happen between now and in n seconds. This also means that the n parameter decided by the user should be blocked at 3 minutes for simplicity's sake and to limit the possible workload that a high n value paired with high airspace traffic would put on the computer system. We also assumed that there should be an arbitrary minimum interval to check for violations, since the operator should not be able to change it too low to still be able to react to it.

Requirement 8 was also subject to interpretation, as changing an airplane's position is not something that can be realistically done in an instant. Instead, we assumed that it meant to only change the plane's horizontal orientation by a certain number of degrees, since there is already a command to change the

vertical orientation. Finally, requirement 15 was interpreted as implementing every part of the system as processes or threads, since the actual description of requirement 13 also referred to periodic tasks as being either processes or threads.

Analysis

Based on our assumptions from the previous section, the following list shows the same requirements again for convenience, but with our own modified interpretation of those subject to assumptions:

Functional

1. The system must monitor a 3D en-route airspace in the shape of a rectangular prism that consists of 100,000 x 100,000 feet on the horizontal plane and 25,000 feet on vertical plane, which stands 15,000 feet above ground level.
2. Airplanes must have a minimum distance of 1,000 feet vertically and 3,000 feet horizontally between them, otherwise it is considered as a separation violation.
3. ~~The computer system must periodically check all airplanes in the airspace for separation violations at current_time + n seconds, where n is an integer parameter that can be changed by the operator at runtime.~~ → Every time the computer system receives data from the radar, it must check all airplanes in the airspace for future separation violations that will happen in the next n seconds, where n is an integer parameter that can be changed by the operator at runtime to a maximum of 180 seconds and a minimum of 5.
4. ~~The computer system must emit a sonorous and/or visual alarm to the display system alerting the operator of any separation violations that will happen in the next 3 minutes.~~ → The computer system must emit a sonorous and/or visual alarm to the display system alerting the operator of any separation violations that will happen in the next n seconds, where n is the parameter defined in the previous requirement.
5. The computer system must store the airspace in a history file every 30 seconds with enough information to generate an approximation of the history of the airspace over time.
6. The computer system must store all operator requests and commands in a log file.
7. ~~The system must display a plane view of the airspace every 5 seconds showing the position of every aircraft.~~ → The display system must show a plane view of the airspace with the position of every aircraft based on data gathered by the radar every 5 seconds that was passed through the computer system.
8. ~~The operator console must allow the operator to send commands to an aircraft to change its speed, altitude and/or position, or request for augmented information of that plane to be displayed on the screen.~~ → The operator console must allow the operator to send commands to an aircraft to change its horizontal speed, altitude or horizontal position, or request for augmented information of that plane to be displayed on the screen.

Non-functional

9. The airplanes must enter the airspace in a horizontal plane at a constant velocity and altitude.
10. The airplanes must maintain their speed and altitude unless directed by the operator to do so.

11. Every operator command or request must first be passed from the console to the computer, and then to the communication system before being sent to the plane.
12. An input file must be used by the system as a list of airplanes entering the airspace, containing the time of entry, the flight ID, the initial X, Y and Z position, as well as the initial X, Y and Z speed of every aircraft.
13. Each aircraft must be implemented as a process or thread that updates its location every second while listening for radar requests and answering them.
14. The radar must learn about each airplane in the system by communicating directly with their respective thread or process.
15. ~~The active part of the system must consist of a set of periodic tasks using periodic polling to handle specific sporadic events.~~ → Every component of the active system must be implemented as a process or thread using periodic polling to handle sporadic events coming from other threads.
16. The system should be tested under low, medium, high and overload conditions determined by the number of airplanes in the space, the degree of congestion and the amount of IO traffic.

Design

Simultaneous output and input

The first problem that we realized would need addressing was how to handle displaying data periodically on the console while continuously waiting on operator input. The most obvious was to make the code open a second console, either on the IDE or in the QNX virtual machine. This would have allowed us to use this console for outputs and keep the IDE console for operator input. Another alternative was to use the ncurses library in order to control the QNX console output very precisely and enable separating the VM window into multiple quadrants for different information or functionalities. After looking into both, we decided to go with the latter. No matter what solution we would have gone for, we would have needed to protect the output stream since it could potentially be used by multiple processes at the same time and generate something undesirable.

Airspace load creation

One of the requirements of the project is to create plane threads over time from an input text file, and this leads to another requirement stating that we need to test the system under various loads. Since those loads are still open to interpretation, we calculated the number of airplanes that could fit in the airspace without separation violations and got around 1,200 airplanes. This meant that depending on the load, we could go up to about 25% of that value and still relatively stay inside a proper range. Because testing an application requires to think about every possible scenario, we also decided to automate the input file creation based on the user's desired load with the help of pseudo-random number generations. This would allow us to run the code on repeat and spot errors that we might not be able to think about if the input was static. We also decided to ask the user to input what load they want for the current simulation at the very beginning of the program.

Inter-process communication

In terms of task communication with each other and sharing data, there were two options we could have chosen from: shared memory or message passing. In the end we opted for the latter, since it fit elegantly with our design, and avoided some problems that could have arisen with the former. One problem we wanted to avoid was that with shared memory, a finite region of memory must be allocated to both tasks, and with a variable number of planes it would have been hard to manage exactly what size that shared memory region must have been. By using message passing, we circumvented this problem as all messages are of a fixed size and we just had to create algorithms to handle sending/receiving lists of data. Message passing also allows threads to listen as "servers" and increases responsiveness of the threads. They wait for a message, wake up and handle the message, then block again and wait for the next message, allowing other threads access to the CPU in a fair and as-needed basis.

Implementation

Simultaneous output and input

The first solution we tried for this problem was opening another console. However, after a long period of research, trials and errors, we realized that the complex interaction between the compiler, the QNX OS and the IDE did not allow to do this easily and the documentation was always either very vague about where to implement the changes or was simply outdated. After searching for other alternatives, we realized that the QNX OS came with the ncurses library (which allows the programmer to output/read without scrolling), so reading many forum explanations on how to set it up allowed us to make it work. It was not easy at first, since we soon discovered that we needed to link the library to the code's objects inside the Makefile when building.

In order to do so, we had to add an extra line of code to the Makefile for the project, specifically telling the linker where to look for the "ncurses.h" library since its not a standard library, but luckily came pre-installed on the QNX system. The edit was quite simple and is shown in the Figure 2 screenshot below. The downside is that Ncurses was not designed to function in the QNX IDE console/output window, and therefore we had to employ a work-around to run and test our application. The fix is not too complicated, an SSH connection must be made to the QNX target through a terminal application like Putty, which supports Ncurses unlike the QNX IDE console. Because we didn't know how to permanently upload the executable for our app onto the target QNX (QNX documentation is vague and out-of-date), we ended up having to run the code in the IDE on the VM, which uploads the program to the QNX VM in the /tmp/ directory, then ignoring the IDE and running the program in the Putty terminal by calling the temporary executable: /tmp/COEN320-Project (executable name). In this manner we achieved concurrent I/O on the terminal for our applications, but next came the problem of making the I/O operations thread safe.

```
61  #Linking rule
62  $(TARGET):$(OBJS)
63      $(LD) -o $(TARGET) $(LDFLAGS_all) $(LDFLAGS) $(OBJS) $(LIBS_all) $(LIBS) -l ncurses
64
```

Figure 2: Screenshot of the project Makefile, lines 61-64 which include the Linking rules. The '-l ncurses' option can be seen at the end of the line.

The solution to achieving simultaneous Input and Output was to use the Ncurses library in conjunction with a mutex lock on input/output related code, protecting the shared resource: the terminal/console window. Threads that use this mutex for I/O are the Display and Console threads, which print to the screen and read input from the screen, respectively. When the Console is modifying the screen or reading data, it locks the mutex for its Critical Section, thus blocking the Display from making changes to the cursor location if it tries to output data concurrently. Similarly, when the Display thread needs to write to the screen, it locks the mutex (so the Console can't make changes) but also stores the current cursor position (for input) before executing its Critical Section and finally resetting the cursor to its initial position before it unlocks the mutex. In this manner the mutex enabled us to achieve bug-free, readable and responsive I/O operations for the application when one thread is responsible for input and another for the output.

During our project, we received new information regarding the requirements of the data display system. We were told that since we have very minimal output space and control over a normal QNX console, whether it be the IDE's or the OS's, it is not needed to show an actual radar top-down plane view of the airspace. Instead, we can list all relevant information about the airplanes that are currently in the airspace at the time of output.

Airspace load creation

The randomized creation of the input load file was more complicated than what was initially anticipated because we had to learn how to work with QNX and navigate through its storage system. We eventually ended up having to create directories on some of our team members' VMs that were using 7.0 instead of 7.1, which did not have a "data" directory.

Aside from the technical difficulties, our team came up with an algorithm that would use the pseudo-random number generation from a C++ function to generate every parameter contained in the input file with randomized scenarios. The algorithm starts by choosing how many planes there will be in total from intervals based on the user's initially desired load. It then enters a loop that creates InitialPlane objects with pseudo-random entry times based on the user's desired load. The object initially generates its own randomised ID and constant properties like the vertical speed, then decides which side of the airspace the airplane is going to come from as well as a random speed magnitude between an interval that we determined from researching on average airplane speeds. Based on the side of entry that was just determined, the object will calculate its vectorial horizontal speeds with a random angle (with the horizontal axis fixed on that side) from -45 to 45 degrees as well as a randomized specific entry point.

Every airplane that is created is stored in a vector that is used to iteratively print all the planes' information onto a file on the QNX virtual machine. Afterwards, the program reads the file back to create temporary PlaneInfo struct instances and starts the system, then it creates threaded Plane objects representing an airplane inside the system, but only when the time corresponds to one of the entry times in the file that is conveniently sorted in ascending order.

Inter-process communication

In order for the threads to recognize data passed from one to another, we created a data type consisting of a QNX pulse header (with type and subtype fields) combined with a custom PlaneInfo_t containing plane information (ID, x, y, z, dx, dy, dz, FL) and some extra data (integer and double) needed by certain

commands. Then almost all of the threads created in the application (except for the Operator console, which functions by periodic polling of the input terminal) run in a while() loop with a MsgReceive() function for message passing right at the start of the loop. Once the thread receives a message, it will read the message Type and Subtype and call the appropriate logic for the received message, be it a command type, timeout type, exit type, or other. After reacting to the message, the thread loops back to the MsgReceive() and blocks, allowing other threads the chance to use the CPU. Care must be taken when sending messages, as a MsgSend() blocks and waits for a MsgReply(), the latter which must be sent as quickly as possible to allow the sending thread to unblock. When lists of data need to be sent, like multiple planes info from the Radar to the central CPU (for computation) and then to the Display task, an index is simply passed in the message subtype along with the data in order for the receiving thread to recognize more data is incoming. The IPC developed for this application also follows the model of the system component diagram of Figure 1 exactly: all radar data get passed through the CPU, which performs computations before passing it to the Display; The operator console gets an command input and passes it to the CPU, which then sends it to the Comms thread, which passes it to the plane thread, which replies back with the data to the Comms thread, before it gets routed through the CPU to the display task for output.

Also, a quick word on periodic polling for sporadic tasks. Only a single task in this system does this, the Command Console thread. Since we used the Ncurses library, cin does not work as expected with the thread. Therefore we went with Ncurses' non-blocking getchar() function. Every 50ms (pretty responsive, but some delay), the Console tasks reads the value of getchar(). If it is garbage, it is ignored. But if it's an acceptable char (alphanumeric, spaces, dots, and minus) it gets added to the input string, or if its backspace it deletes a char. The periodic polling was used to handle the only sporadic tasks (Console) in the system, the User can enter a command at any time and it must be processed quickly. All other tasks had hard deadlines, like the Radar pulses every 5s and constraints violation calculation of CPU thread before outputting to the operator as quickly as possible.

Operator commands and airplane calculations

As mentioned before, the operator commands in one of the requirements are somewhat open to interpretation. This means that we had to come up with our own implementation of the operator commands to either change an airplane's altitude, its horizontal direction or its speed. Displaying a plane's data was not as open to interpretation and only needed data passing between processes. Every implementation of the operator commands was based on ease of use for an average human operator and always needs the plane ID as the first argument to work.

The speed change command was the simplest one of the three to implement, as it only required us to change the current speed. We decided to go with the user having to enter a new speed that respects the average speed boundaries determined before and when the command will reach the airplane, said plane will calculate new vectorial speeds for every horizontal direction based on the percentage which represents the difference between the new speed and the old one. It is important to note that since our system is supposed to be an oversimplification of a real ATC system, planes are changing their positional and directional values almost instantly, unlike in a real-world application.

For the altitude change, we first had many implementations that revolved around giving the plane a certain vertical angle to face to calculate the different vectorial speeds. This approach was unreliable as

we did not know exactly how plane physics worked and it also resulted in a lot of unnecessarily complicated calculations. After careful research, we found out that planes all had average lift speeds and decided to go with 50 as a constant upward or downward speed. For simplicity's sake, we decided to not modify any of the horizontal speeds.

Lastly, the position change was quite the trigonometrical challenge. Because we interpreted the command as a change in horizontal direction, we ended up having a calculation system consisting of two different cartesian systems: the airspace and the airplane. After solving this problem, we managed to get it to work by having the operator input an angle in degrees between -25 and 25. This angle is used to make the plane turn either left if the value is positive, or right if the value is negative. These arbitrary values are since planes usually don't turn on hard angles unless they need to make an evasive maneuver. It also simplifies the calculations as the plane object can calculate its current angle and simply add the angle input to it to recalculate the vectorial speeds.

Lessons learned

There were not many lessons learned during the making of this project as most of the coding knowledge needed, such as thread management and resource sharing, had already been acquired from previous classes in our curriculum. However, we have to admit that the tutorials of this class during the semester helped us understand new things such as using message passing for inter-process communication, but it was long before we started working on the project. The only thing that needed adaptation was learning how to work with the QNX system and IDE. If anything, the hardest parts of the projects were only made hard by having to adapt our desired implementation to the QNX environment.

If we were to repeat this experiment, we would definitely use another system than QNX to run our application. The overall system is extremely buggy and seems to have a lot of compatibility and customization issues. Most of the documentation is outdated or very poorly detailed with no relevant examples whatsoever and as engineers, we know that good documentation determines how well a system can evolve and be used. We would probably use a host-based C++ compiler on our host system, simulate everything using only one core (using either OS-specific system commands or by making the entire system run sequentially) and make the overall program compatible with Windows, Mac and Linux-based systems.

Conclusion

For this project, we managed to successfully implement every desired requirement to simulate a proper real-time system. The multi-thread system used to simulate every component of an ATC system has been implemented in a way that perfectly mimics the behaviour of a single-core RTOS by using message passing for inter-process communication. The most important issue that we faced was having to adapt every implementation to the unreliable QNX environment. In the future, we would like to explore other environments for RTOS such as programming on physical microcontroller modules to produce real-life applications.