INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT087-3-3-RTS

REALTIME SYSTEMS

APUF2211CS(DA)

HAND OUT DATE	**: 9 JANUARY 2023**
HAND IN DATE	**: 23 MARCH 2023**
WEIGHTAGE	**: 50%**

NAME	**:**	**MARCELL AGUNG WAHYUDI**
STUDENT ID	**:**	**TP058650**
INTAKE CODE	**:**	**APU3F2211CS(DA)**
MODULE CODE	**:**	**CT087-3-3-RTS**
ASSIGNMENT	**:**	**INDIVIDUAL ASSIGNMENT**
LECTURER	**:**	**ASSOC. PROF. DR. IMRAN MEDI**

Contents

Investigating How Programming Design Effects Real-Time Performance - A Simulation of A Flight Control System

Marcell Agung Wahyudi | TP058650

Abstract

A real-time system or RTS in short, is a computer system that is designed to respond to events or input signals in real-time, without delay. It is commonly used in applications that require time-critical processing, such as process control systems, robotics, and multimedia systems. The key characteristic of a real-time system is its ability to respond to an event or input signal within a specified time limit, known as a deadline (Abdallah & Mohamed, 2019). The proposed project aims to improve the safety and efficiency of passenger airliners by developing an intelligent real-time system to manage and coordinate the various subsystems and processes involved in a typical flight.

Keywords: Airline Flight Management System, Benchmarking, Java, Programming Language, Real-time system.

1.0 Introduction

Air travel has revolutionized transportation and connectivity around the world, enabling people to travel to distant places with speed and comfort. However, the increasing complexity of passenger airliners has resulted in greater challenges for flight crew and ground personnel in ensuring safe and efficient flight operations. The flight system of a typical passenger airliner comprises numerous subsystems and processes that need to be monitored and managed in real-time. These processes include monitoring weather conditions, altitude, engine performance, and ensuring that the aircraft stays on course. Any deviation from optimal performance can have severe consequences, including accidents and incidents that put the safety of passengers and crew at risk. To address these challenges, this paper proposes the development of an intelligent real-time system that can manage and coordinate the various subsystems and processes involved in a typical flight utilizing Java programming language. This system will use advanced algorithms and machine learning techniques to optimize the flight plan in real-time, ensuring safe and efficient flight operations. The paper will discuss the potential benefits of such a system and the technical challenges that need to be addressed in its development. The contribution of the researcher may improve the current air travel and/or improve another researcher’s work. In order to make the research paper viable, a sufficient literature reviews and research backgrounds are mandatory to compare to the present real-world airline flight management system, and also to identify any issues at the present moment regarding the airline system. After a sufficient literature review is done, the researcher then will discuss about the system itself and how the code works while also describing the methodology used, then results and discussion will be done to mention any issues faced and the desired results, and lastly, a conclusion chapter to conclude everything in the document.

2.0 Literature Review

2.1 The Evolution of Flight Management Systems.

The evolution of flight management systems (FMS) has been a subject of extensive research and development in the aviation industry. In this paper, the researcher reviewed the literature on the history and evolution of FMS, starting from the early days of mechanical instrumentation to the advanced computerized systems of today. The earliest FMS consisted of basic mechanical instrumentation, such as compasses, altimeters, and airspeed indicators, which provided the pilot with basic information about the aircraft's performance. The introduction of radio communication systems in the 1930s enabled pilots to receive information about weather conditions and air traffic, allowing for more precise navigation and control. In the 1960s, the first computerized FMS was developed, which enabled pilots to input the desired flight plan and receive real-time guidance and feedback on their progress. The introduction of GPS in the 1990s revolutionized FMS, allowing for more precise navigation and control. Today's FMS are highly advanced computer systems that integrate data from multiple sources, including weather sensors, radar, GPS, and aircraft systems. They provide pilots with real-time guidance on navigation, fuel consumption, engine performance, and other critical aspects of flight management. They also enable ground personnel to monitor the aircraft's performance and provide support in case of emergencies. Despite the significant progress made in FMS, challenges remain, including issues related to interoperability, data management, and cybersecurity. Future research is needed to address these challenges and further improve the safety and efficiency of flight operations. In conclusion, "The Evolution of Flight Management Systems" by David Avery provides a comprehensive overview of the development and evolution of flight management systems (FMS). The paper traces the history of FMS from its inception in the 1960s to the present day, highlighting the key technological advances that have contributed to its evolution. The author also examines the various challenges and complexities associated with the design, development, and implementation of FMS, as well as the ongoing efforts to improve their functionality and reliability. The paper provides valuable insights into the future of FMS, including the potential impact of emerging technologies such as artificial intelligence and machine learning. Overall, this paper is a valuable resource for anyone interested in the history and evolution of FMS, as well as the ongoing challenges and opportunities facing this important area of aviation technology. The author's deep knowledge of the subject matter is evident throughout the paper, making it a must-read for aviation professionals, researchers, and enthusiasts alike (Avery, 19994).

2.2 Scheduling Algorithms for Real-Time Systems

“Scheduling Algorithms for Real-Time Systems" by Arezou Mohammadi and Selim G. Akl presents an overview of the scheduling algorithms used in real-time systems. The paper discusses the challenges involved in scheduling tasks in real-time systems and highlights the importance of meeting timing constraints in such systems. The authors then describe the most used scheduling algorithms in real-time systems, including Rate Monotonic (RM), Earliest Deadline First (EDF), Deadline Monotonic (DM), and the Priority Inheritance Protocol (PIP). The paper goes on to provide a comparative analysis of these algorithms in terms of their performance metrics, such as response time, schedulability, and efficiency. The authors also discuss the limitations of these algorithms and the future research directions in this area. Overall, the paper provides a comprehensive overview of the scheduling algorithms used in real-time systems and can be a useful resource for researchers and practitioners working in this area. In conclusion, "Scheduling Algorithms for Real-Time Systems" is a valuable contribution to the field of real-time systems. The authors have provided an insightful analysis of the scheduling algorithms used in real-time systems and their performance characteristics. The paper is well-organized and written in a clear and concise manner, making it easy to understand for readers with different levels of expertise (Mohammadi & G. Akl, 2005).

2.3 Real-time communication protocols: an overview

In "Real-time communication protocols: an overview," authors Ferdy Hanssen and Pierre G. Jansen provide an extensive review of real-time communication protocols. The paper begins by discussing the importance of real-time communication and its role in modern society. The authors then introduce the concept of real-time communication protocols and their importance in various applications, including industrial automation and control systems, automotive systems, and multimedia streaming. The authors provide a comprehensive review of several commonly used real-time communication protocols, including CAN, FlexRay, Ethernet/IP, and EtherCAT. They discuss the features, benefits, and limitations of each protocol and compare them based on various criteria such as data rate, latency, and reliability. Furthermore, the authors present an overview of the Real-time Transport Protocol (RTP) and the Real-time Transport Control Protocol (RTCP) and their use in multimedia streaming applications. They discuss the benefits and challenges of using RTP and RTCP for real-time communication and provide examples of their implementation in various multimedia applications. In conclusion, the authors emphasize the importance of choosing the appropriate real-time communication protocol based on the specific requirements of the application. They also suggest that further research is needed to develop new real-time communication protocols that can address the challenges of emerging applications such as Internet of Things (IoT) and cyber-physical systems. Overall, the paper provides an informative overview of real-time communication protocols and their applications, making it a valuable resource for researchers, engineers, and anyone interested in the field of real-time communication (Ferdy & Jansen, 2003).

2.4 Java

Java is a high-level, object-oriented programming language developed by Sun Microsystems, which was later acquired by Oracle Corporation. It was designed to be platform-independent, meaning that Java programs can run on any system that has a Java Virtual Machine (JVM) installed, regardless of the underlying hardware and operating system. Java is widely used in the development of desktop applications, web applications, mobile applications, and embedded systems. One of the key features of Java is its "write once, run anywhere" philosophy. Java code is compiled into bytecode, which can run on any system that has a JVM installed. This makes it easier for developers to create and deploy applications across different platforms, without having to rewrite the code for each platform. Java is also known for its robustness, as it includes features such as automatic memory management and exception handling. The Java Virtual Machine (JVM) manages memory allocation and garbage collection, freeing the programmer from the need to manually manage memory. Additionally, Java's exception handling mechanism allows developers to write code that gracefully handles errors and exceptions, improving the overall reliability of Java programs. Java has a vast library of built-in classes and APIs that make it easier for developers to write complex programs. These libraries include everything from basic data structures to advanced network programming and cryptography. Java's standard libraries provide a comprehensive set of tools for developers to create applications of any size or complexity. Java is also highly popular in the enterprise world, with many large corporations using Java for mission-critical applications. Its popularity has also led to a large and active developer community, which provides support and resources to new and experienced Java developers alike. In conclusion, Java is a versatile and powerful programming language that is widely used for developing a variety of applications. Its platform independence, robustness, and extensive library of built-in classes and APIs make it a popular choice among developers. (Oracle Corporation, 2022).

2.5 Java Benchmarking Harness

Java Benchmarking Harness (JMH) is a powerful tool for benchmarking Java code. It is a Java-based tool that can be used to measure the performance of Java code accurately. JMH is designed to handle the issues that developers face when benchmarking Java code, such as JVM warm-up time, garbage collection, and accurate timing measurements. JMH provides a flexible and extensible framework for benchmarking Java code. It allows developers to write microbenchmarks that measure the performance of small code snippets, as well as larger benchmarks that measure the performance of entire applications. JMH can be used to measure the performance of code running on the JVM, including Java, Scala, and Kotlin code. The JMH benchmarking harness is built on top of the Java Micro benchmarking Benchmark (JMB) infrastructure. JMB is a set of Java classes and tools that provide support for benchmarking Java code. It includes the JMH tool for writing and running benchmarks, as well as a set of helper classes for measuring performance and collecting results. JMH provides several features that make it a powerful tool for benchmarking Java code. One of the key features is the ability to handle JVM warm-up time. JVM warm-up time can significantly impact the performance of Java code, as the JVM must compile the code and perform other initialization tasks before it can run at peak performance. JMH handles JVM warm-up time by running the benchmark multiple times and discarding the results of the warm-up runs. Another important feature of JMH is its ability to handle garbage collection. Garbage collection can also impact the performance of Java code, as it can cause interruptions in the execution of the code. JMH handles garbage collection by measuring the time taken for garbage collection and subtracting this time from the overall benchmarking results. JMH also provides accurate timing measurements. It uses several techniques to ensure that the timing measurements are as accurate as possible. For example, it uses a high-resolution timer to measure the time taken for code execution, and it uses statistical techniques to eliminate outliers and other sources of error. In conclusion, the Java Benchmarking Harness (JMH) is a powerful tool for benchmarking Java code. It provides a flexible and extensible framework for measuring the performance of Java code accurately. JMH handles the issues that developers face when benchmarking Java code, such as JVM warm-up time, garbage collection, and accurate timing measurements. With its many features, JMH is an essential tool for anyone looking to optimize the performance of Java applications (OpenJDK, 2023).

3.0 Methodology

3.1 Summary of the Simulation

The purpose of this project is to design a real-time system using Java Programming Language with the help of RabbitMQ that can manage and coordinate several processes, since the usual flight system is made up of several processes running simultaneously, responsible for every necessary aspect of the flight. The flight control system comprises of two systems, which is the sensory systems, and the actuator systems. The sensory systems check and manages the altitude variable, cabin, speed/direction variables, and weather/environment variables, while the actuator systems guide the engines, wing flags, tail flags, landing gear, and oxygen masks.

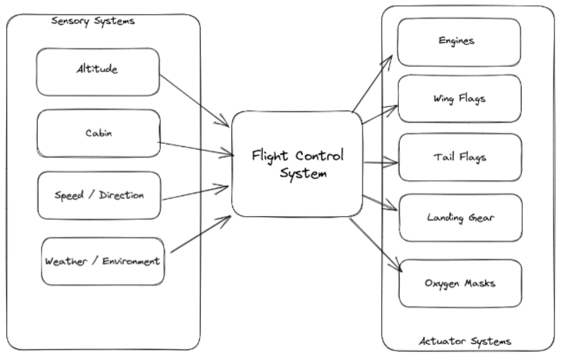


Figure 1: Sensory Systems and Actuator Systems (source:Assignment Question)

3.2 Resources

Resource	Explanation	Application
IDE	Integrated Development Environment	Apache NetBeans IDE 17
Messaging System	Manages communication between threads	RabbitMQ
Micro-Benchmarking	To Evaluate the Program's performance	Java Micro-benchmark Harness (JMH)

3.3 Flow of the Program

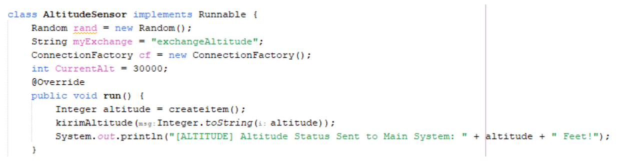
To make the system work, we must have three different logic system, the first one is the sensory system, which sends message to the main system, this system will have randomized variables since, reflecting real life, variables such as weather is not completely calculatable, and so is pressure, direction and etc. The second logic system, the main system, acts as a receiver and a sender, this system receives information from the sensory system, and proceeds it to the actuator system, actuator system is a system that help stabilize the uncontrollable variables, such as controlling the speed and altitude of the plane making the plane fly smoothly.

3.4 Methodology

The method to how the system is to be made is by utilizing RabbitMQ messaging services. RabbitMQ employs multithreading to manage more than one requests simultaneously. It accomplishes it by spawning numerous threads, with everyone in charge of dealing with a single program. This enables RabbitMQ to process multiple requests at once, leading to quicker response times and better productivity. To manage various interconnection, RabbitMQ also employs multithreading. This enables it to control more connections at once, leading to greater reliability as well as scalability (Bao, 2023).

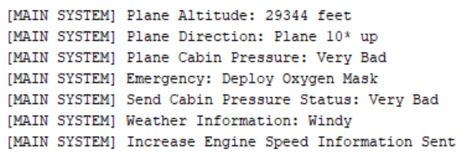
4.0 Results and discussion

Once the simulation is started, the code will simulate a currently flying plane, with the starting altitude of 30000.



```
class AltitudeSensor implements Runnable {
    Random rand = new Random();
    String myExchange = "exchangeAltitude";
    ConnectionFactory cf = new ConnectionFactory();
    int CurrentAlt = 30000;
    @Override
    public void run() {
        Integer altitude = createitem();
        kirimAltitude(msg:Integer.toString(i: altitude));
        System.out.println("[ALTITUDE] Altitude Status Sent to Main System: " + altitude + " Feet!");
    }
```

Figure 2: Sensory Altitude System Code



```
[MAIN SYSTEM] Plane Altitude: 29344 feet
[MAIN SYSTEM] Plane Direction: Plane 10* up
[MAIN SYSTEM] Plane Cabin Pressure: Very Bad
[MAIN SYSTEM] Emergency: Deploy Oxygen Mask
[MAIN SYSTEM] Send Cabin Pressure Status: Very Bad
[MAIN SYSTEM] Weather Information: Windy
[MAIN SYSTEM] Increase Engine Speed Information Sent
```

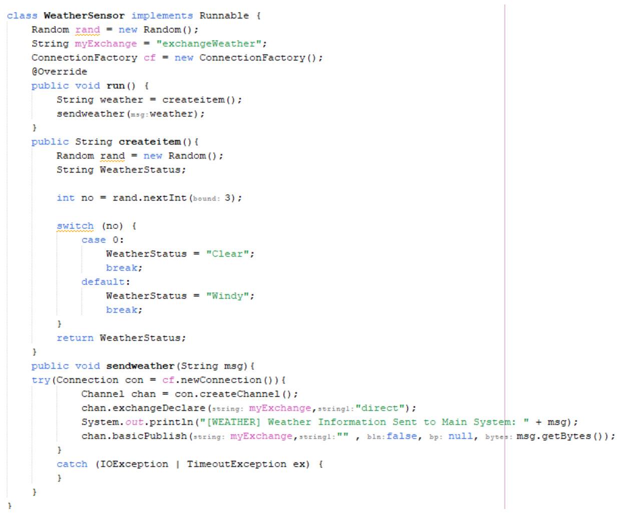
Figure 3: Code Snippet

As soon as the program launched, since the default altitude is 30000, when it dropped to 29344 feet seen in the snippet, the plane immediately detected the cabin pressure is very bad, hence the plane automatically triggered the emergency countermeasure which is deploying masks for the passengers, then the information of cabin pressure is sent to the actuator system. At the same time, the weather sensory system detected that it is windy, and simultaneously, the increase engine speed information is also sent to the engine actuator to correct the cabin pressure.

The following code snippets and outputs are from the sensory systems:

1. Weather/Environment Sensory System

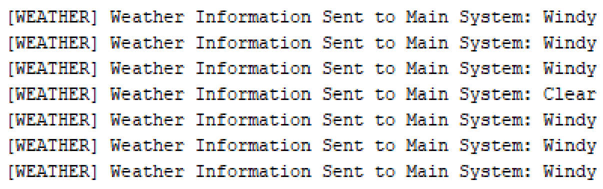
Weather class has two functions, one for randomizing the type of weather (Clear or Windy) and the other to send the weather information to the main system.



```
class WeatherSensor implements Runnable {
    Random rand = new Random();
    String myExchange = "exchangeWeather";
    ConnectionFactory cf = new ConnectionFactory();
    @Override
    public void run() {
        String weather = createitem();
        sendweather(msg:weather);
    }
    public String createitem(){
        Random rand = new Random();
        String WeatherStatus;

        int no = rand.nextInt(bound: 3);

        switch (no) {
            case 0:
                WeatherStatus = "Clear";
                break;
            default:
                WeatherStatus = "Windy";
                break;
        }
        return WeatherStatus;
    }
    public void sendweather(String msg){
    try(Connection con = cf.newConnection()){
            Channel chan = con.createChannel();
            chan.exchangeDeclare(string: myExchange,string1:"direct");
            System.out.println("[WEATHER] Weather Information Sent to Main System: " + msg);
            chan.basicPublish(string: myExchange,string1:"" , bln:false, bp: null, bytes: msg.getBytes());
        }
        catch (IOException | TimeoutException ex) {
        }
    }
}
```

Figure 4:Weather System Code Snippet

Running this class will give the following outcome:

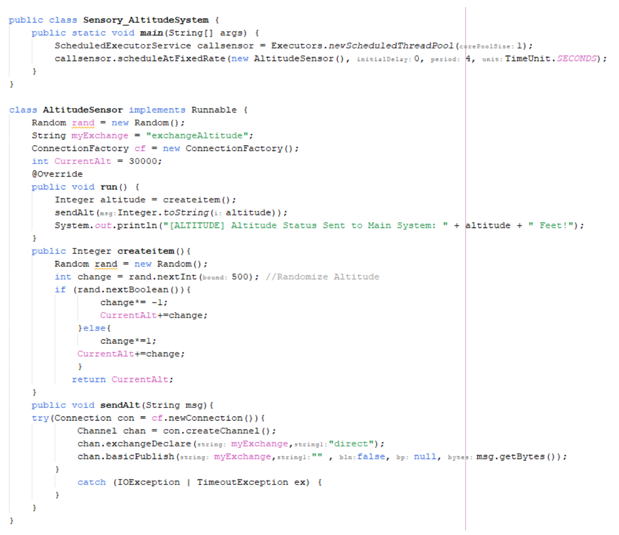


```
[WEATHER] Weather Information Sent to Main System: Windy
[WEATHER] Weather Information Sent to Main System: Windy
[WEATHER] Weather Information Sent to Main System: Windy
[WEATHER] Weather Information Sent to Main System: Clear
[WEATHER] Weather Information Sent to Main System: Windy
[WEATHER] Weather Information Sent to Main System: Windy
[WEATHER] Weather Information Sent to Main System: Windy
```

Figure 5: Weather System Class Output

2. Altitude Sensory System

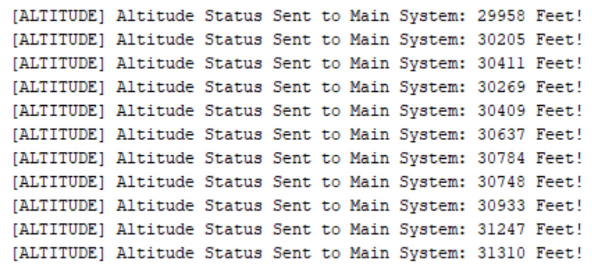
Like the weather sensory system, altitude system also has two functions, which is one for setting the initial altitude of the plane and also randomizing the next altitude, and the other function to send the altitude information to the main system.



```
public class Sensory_AltitudeSystem {
    public static void main(String[] args) {
        ScheduledExecutorService callsensor = Executors.newScheduledThreadPool(corePoolSize: 1);
        callsensor.scheduleAtFixedRate(new AltitudeSensor(), initialDelay: 0, period: 4, unit: TimeUnit.SECONDS);
    }
}

class AltitudeSensor implements Runnable {
    Random rand = new Random();
    String myExchange = "exchangeAltitude";
    ConnectionFactory cf = new ConnectionFactory();
    int CurrentAlt = 30000;
    @Override
    public void run() {
        Integer altitude = createitem();
        sendAlt(msg: Integer.toString(i: altitude));
        System.out.println("[ALTITUDE] Altitude Status Sent to Main System: " + altitude + " Feet!");
    }
    public Integer createitem(){
        Random rand = new Random();
        int change = rand.nextInt(bound: 500); //Randomize Altitude
        if (rand.nextBoolean()){
                change*= -1;
                CurrentAlt+=change;
            }else{
                change*=1;
            CurrentAlt+=change;
            }
          return CurrentAlt;
    }
    public void sendAlt(String msg){
    try(Connection con = cf.newConnection()){
            Channel chan = con.createChannel();
            chan.exchangeDeclare(string: myExchange,string1:"direct");
            chan.basicPublish(string: myExchange,string1:"" , bln:false, bp: null, bytes: msg.getBytes());
        }
            catch (IOException | TimeoutException ex) {
        }
    }
}
```

Figure 6: Altitude System Code Snippet

Running the code itself will only print out the altitudes that are supposed to be sent to the main system.

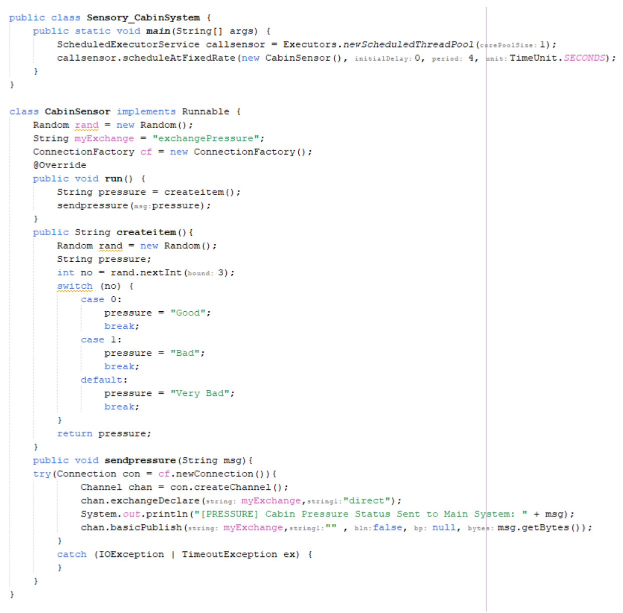


```
[ALTITUDE] Altitude Status Sent to Main System: 29958 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30205 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30411 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30269 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30409 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30637 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30784 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30748 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 30933 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 31247 Feet!
[ALTITUDE] Altitude Status Sent to Main System: 31310 Feet!
```

Figure 7: Altitude System Output

3. Cabin Pressure Sensory System

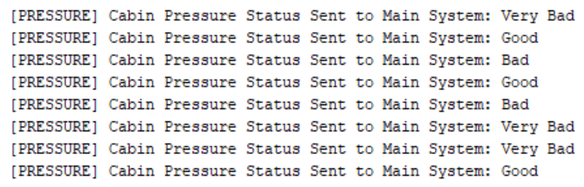
Cabin pressure also has a randomizer, which randomizes the pressure inside the plane cabin with it having 3 possibilities, good, bad, or very bad, with each information of the cabin pressure status being sent to the main system.



```
public class Sensory_CabinSystem {
    public static void main(String[] args) {
        ScheduledExecutorService callsensor = Executors.newScheduledThreadPool(corePoolSize: 1);
        callsensor.scheduleAtFixedRate(new CabinSensor(), initialDelay: 0, period: 4, unit: TimeUnit.SECONDS);
    }
}

class CabinSensor implements Runnable {
    Random rand = new Random();
    String myExchange = "exchangePressure";
    ConnectionFactory cf = new ConnectionFactory();
    @Override
    public void run() {
        String pressure = createitem();
        sendpressure(msg: pressure);
    }
    public String createitem(){
        Random rand = new Random();
        String pressure;
        int no = rand.nextInt(bound: 3);
        switch (no) {
            case 0:
                pressure = "Good";
                break;
            case 1:
                pressure = "Bad";
                break;
            default:
                pressure = "Very Bad";
                break;
        }
        return pressure;
    }
    public void sendpressure(String msg){
    try(Connection con = cf.newConnection()){
            Channel chan = con.createChannel();
            chan.exchangeDeclare(string: myExchange,string1:"direct");
            System.out.println("[PRESSURE] Cabin Pressure Status Sent to Main System: " + msg);
            chan.basicPublish(string: myExchange,string1:"" , bln:false, bp: null, bytes: msg.getBytes());
        }
        catch (IOException | TimeoutException ex) {
        }
    }
}
```

Figure 8: Cabin Pressure System Code Snippet

Running this class will randomize the three possible cabin pressure status.

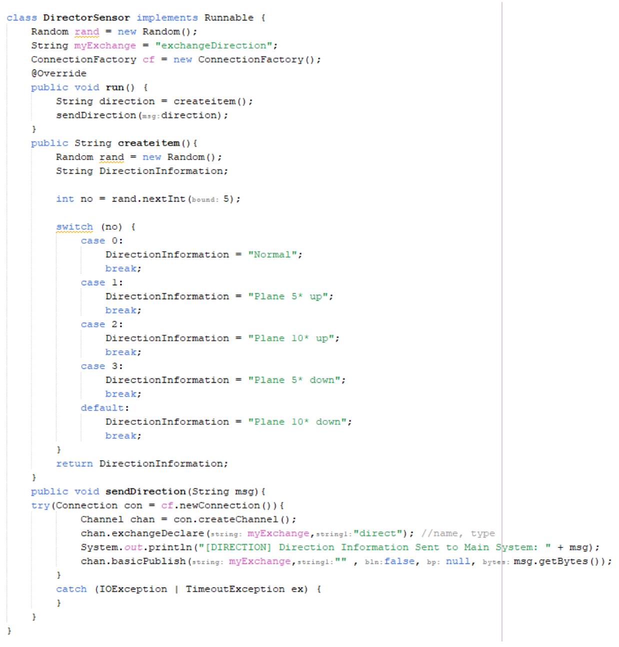


```
[PRESSURE] Cabin Pressure Status Sent to Main System: Very Bad
[PRESSURE] Cabin Pressure Status Sent to Main System: Good
[PRESSURE] Cabin Pressure Status Sent to Main System: Bad
[PRESSURE] Cabin Pressure Status Sent to Main System: Good
[PRESSURE] Cabin Pressure Status Sent to Main System: Bad
[PRESSURE] Cabin Pressure Status Sent to Main System: Very Bad
[PRESSURE] Cabin Pressure Status Sent to Main System: Very Bad
[PRESSURE] Cabin Pressure Status Sent to Main System: Good
```

Figure 9: Cabin Pressure System Outcome

4. Direction Sensory System

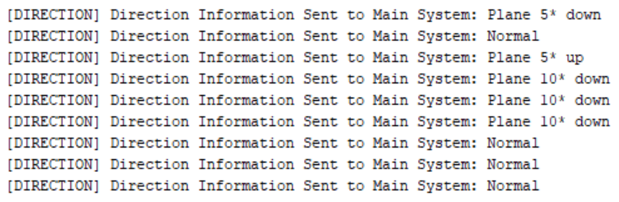
The final sensory system, controls and manages the direction of the plane, implementing a randomizer to randomize that either the plane is going on a straight line, or going up, or going down, the information then sent to the main system for further process.



```
class DirectorSensor implements Runnable {
    Random rand = new Random();
    String myExchange = "exchangeDirection";
    ConnectionFactory cf = new ConnectionFactory();
    @Override
    public void run() {
        String direction = createitem();
        sendDirection(msg:direction);
    }
    public String createitem(){
        Random rand = new Random();
        String DirectionInformation;

        int no = rand.nextInt(bound: 5);

        switch (no) {
            case 0:
                DirectionInformation = "Normal";
                break;
            case 1:
                DirectionInformation = "Plane 5* up";
                break;
            case 2:
                DirectionInformation = "Plane 10* up";
                break;
            case 3:
                DirectionInformation = "Plane 5* down";
                break;
            default:
                DirectionInformation = "Plane 10* down";
                break;
        }
        return DirectionInformation;
    }
    public void sendDirection(String msg){
    try(Connection con = cf.newConnection()){
            Channel chan = con.createChannel();
            chan.exchangeDeclare(string: myExchange,string1:"direct"); //name, type
            System.out.println("[DIRECTION] Direction Information Sent to Main System: " + msg);
            chan.basicPublish(string: myExchange,string1:"" , bln:false, bp: null, bytes: msg.getBytes());
        }
        catch (IOException | TimeoutException ex) {
        }
    }
}
```

Figure 10: Direction System Code Snippet

Running this class will have outputs printing the direction information sent to the main system as explained previously.

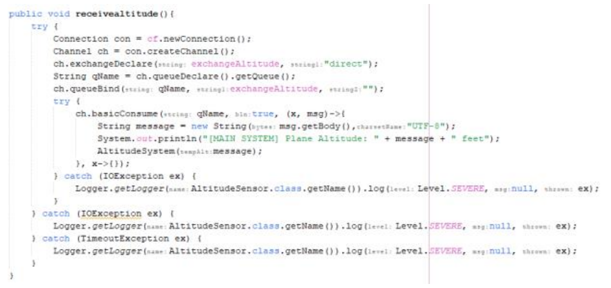


```
[DIRECTION] Direction Information Sent to Main System: Plane 5* down
[DIRECTION] Direction Information Sent to Main System: Normal
[DIRECTION] Direction Information Sent to Main System: Plane 5* up
[DIRECTION] Direction Information Sent to Main System: Plane 10* down
[DIRECTION] Direction Information Sent to Main System: Plane 10* down
[DIRECTION] Direction Information Sent to Main System: Plane 10* down
[DIRECTION] Direction Information Sent to Main System: Normal
[DIRECTION] Direction Information Sent to Main System: Normal
[DIRECTION] Direction Information Sent to Main System: Normal
```

Figure 11: Direction System Output

5. The Main System

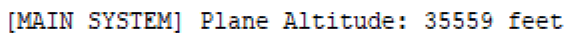
This is considered the primary system of the whole simulation since this system receives information from the sensory and will process and forward said information to the actuator system where the actuator will stabilize the whole plane. The following code snippet will give an example of a receiving function, in this case, the main system is receiving the altitude from the sensory system.



```
public void receivealtitude(){
    try {
        Connection con = cf.newConnection();
        Channel ch = con.createChannel();
        ch.exchangeDeclare(string: exchangeAltitude, string1:"direct");
        String qName = ch.queueDeclare().getQueue();
        ch.queueBind(string: qName, string1:exchangeAltitude, string2:"");
        try {
            ch.basicConsume(string: qName, bln:true, (x, msg)->{
                String message = new String(bytes: msg.getBody(),charsetName:"UTF-8");
                System.out.println("[MAIN SYSTEM] Plane Altitude: " + message + " feet");
                AltitudeSystem(tempAlt:message);
            }, x->{});
        } catch (IOException ex) {
            Logger.getLogger(name: AltitudeSensor.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        }
    } catch (IOException ex) {
        Logger.getLogger(name: AltitudeSensor.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
    } catch (TimeoutException ex) {
        Logger.getLogger(name: AltitudeSensor.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
    }
}
```

Figure 12: Receive Altitude Function

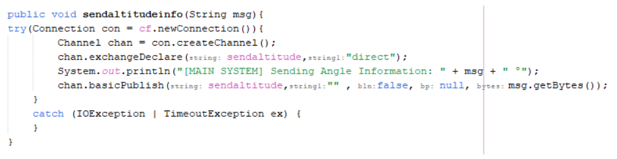
Then the main system will also print out a line describing the current altitude of the plane, shown in the following outcome:



```
[MAIN SYSTEM] Plane Altitude: 35559 feet
```

Figure 13: Main System Output

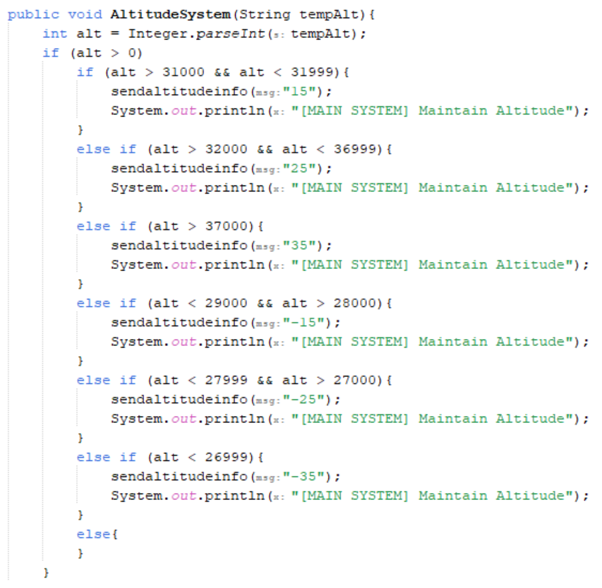
After receiving the altitude information, the main system will then send out a message to the actuator system with the following code:



```
public void sendaltitudeinfo(String msg){
try(Connection con = cf.newConnection()){
        Channel chan = con.createChannel();
        chan.exchangeDeclare(string: sendaltitude,string1:"direct");
        System.out.println("[MAIN SYSTEM] Sending Angle Information: " + msg + " °");
        chan.basicPublish(string: sendaltitude,string1:"" , bln:false, bp: null, bytes: msg.getBytes());
    }
    catch (IOException | TimeoutException ex) {
    }
}
```

Figure 14: Sending Altitude Code Snippet

Alongside that, the main system will also inform the plane to maintain altitude with the following coed snippet:

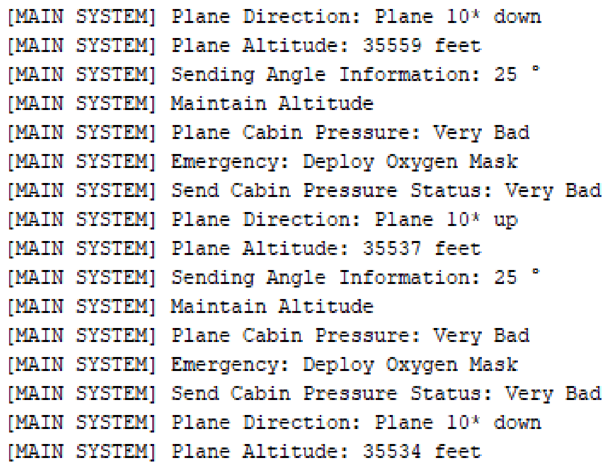


```
public void AltitudeSystem(String tempAlt){
    int alt = Integer.parseInt(s: tempAlt);
    if (alt > 0)
        if (alt > 31000 && alt < 31999){
            sendaltitudeinfo(msg:"15");
            System.out.println(x: "[MAIN SYSTEM] Maintain Altitude");
        }
        else if (alt > 32000 && alt < 36999){
            sendaltitudeinfo(msg:"25");
            System.out.println(x: "[MAIN SYSTEM] Maintain Altitude");
        }
        else if (alt > 37000){
            sendaltitudeinfo(msg:"35");
            System.out.println(x: "[MAIN SYSTEM] Maintain Altitude");
        }
        else if (alt < 29000 && alt > 28000){
            sendaltitudeinfo(msg:"-15");
            System.out.println(x: "[MAIN SYSTEM] Maintain Altitude");
        }
        else if (alt < 27999 && alt > 27000){
            sendaltitudeinfo(msg:"-25");
            System.out.println(x: "[MAIN SYSTEM] Maintain Altitude");
        }
        else if (alt < 26999){
            sendaltitudeinfo(msg:"-35");
            System.out.println(x: "[MAIN SYSTEM] Maintain Altitude");
        }
        else{
        }
    }
```

Figure 15: Altitude System Code Snippet

The rest of the functions follow almost the exact same flow, with the main system having a receiver, processes the received information, then sends the information to the actuator system, a more complete detailed functionality of each function can be seen in the code provided with the document itself.

The following screenshot will show the complete functionality of the main system working with the sensory systems and actuator systems all working simultaneously.

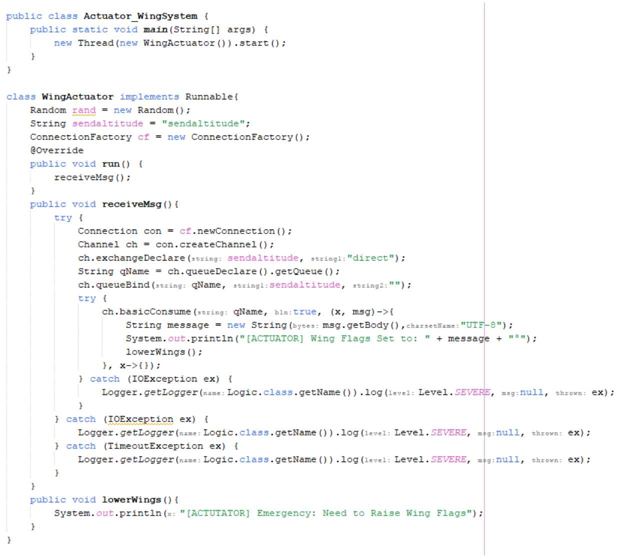


```
[MAIN SYSTEM] Plane Direction: Plane 10* down
[MAIN SYSTEM] Plane Altitude: 35559 feet
[MAIN SYSTEM] Sending Angle Information: 25 °
[MAIN SYSTEM] Maintain Altitude
[MAIN SYSTEM] Plane Cabin Pressure: Very Bad
[MAIN SYSTEM] Emergency: Deploy Oxygen Mask
[MAIN SYSTEM] Send Cabin Pressure Status: Very Bad
[MAIN SYSTEM] Plane Direction: Plane 10* up
[MAIN SYSTEM] Plane Altitude: 35537 feet
[MAIN SYSTEM] Sending Angle Information: 25 °
[MAIN SYSTEM] Maintain Altitude
[MAIN SYSTEM] Plane Cabin Pressure: Very Bad
[MAIN SYSTEM] Emergency: Deploy Oxygen Mask
[MAIN SYSTEM] Send Cabin Pressure Status: Very Bad
[MAIN SYSTEM] Plane Direction: Plane 10* down
[MAIN SYSTEM] Plane Altitude: 35534 feet
```

Figure 16: Main System Output

6. Wing Flags Actuator System

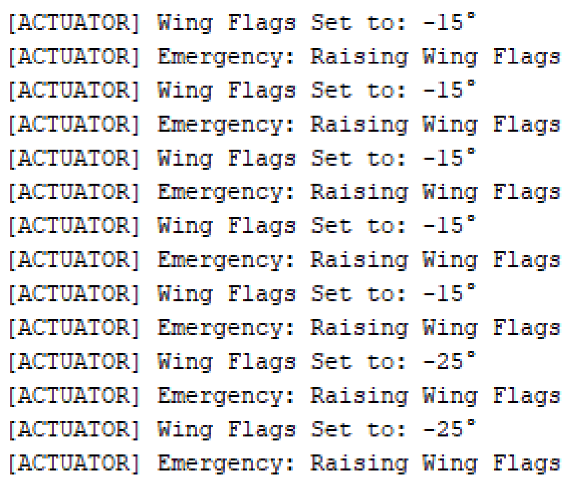
After receiving altitude information from the main system, the code for the wing flags actuator will then process the input and decide what to do with it. If the plane were to fly too low, the wings of the plane will have to be set higher, if the plane were to fly very low, then the system will send an emergency message and raise the wing flags.



```
public class Actuator_WingSystem {
    public static void main(String[] args) {
        new Thread(new WingActuator()).start();
    }
}

class WingActuator implements Runnable{
    Random rand = new Random();
    String sendaltitude = "sendaltitude";
    ConnectionFactory cf = new ConnectionFactory();
    @Override
    public void run() {
        receiveMsg();
    }
    public void receiveMsg(){
        try {
            Connection con = cf.newConnection();
            Channel ch = con.createChannel();
            ch.exchangeDeclare(string: sendaltitude, string1:"direct");
            String qName = ch.queueDeclare().getQueue();
            ch.queueBind(string: qName, string1:sendaltitude, string2:"");
            try {
                ch.basicConsume(string: qName, bln:true, (x, msg)->{
                    String message = new String(bytes: msg.getBody(),charsetName:"UTF-8");
                    System.out.println("[ACTUATOR] Wing Flags Set to: " + message + "°");
                    lowerWings();
                }, x->{});
            } catch (IOException ex) {
                Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
            }
        } catch (IOException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        } catch (TimeoutException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        }
    }
    public void lowerWings(){
        System.out.println(x: "[ACTUTATOR] Emergency: Need to Raise Wing Flags");
    }
}
```

Figure 17: Wing Flags Actuator System

The following outcome can be seen that the system is working very smoothly.

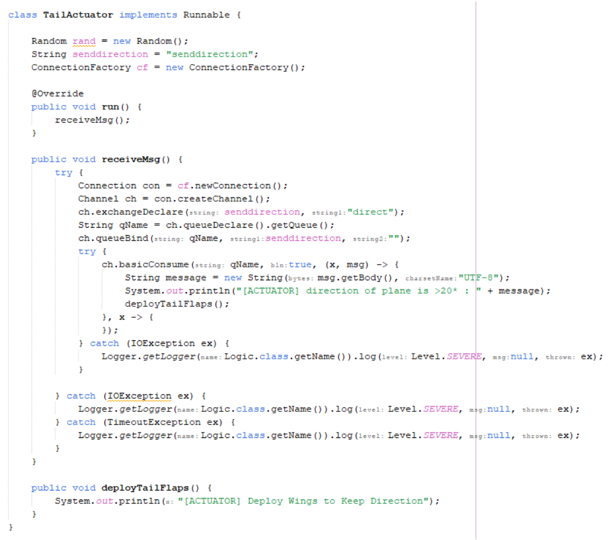


```
[ACTUATOR] Wing Flags Set to: -15°
[ACTUATOR] Emergency: Raising Wing Flags
[ACTUATOR] Wing Flags Set to: -15°
[ACTUATOR] Emergency: Raising Wing Flags
[ACTUATOR] Wing Flags Set to: -15°
[ACTUATOR] Emergency: Raising Wing Flags
[ACTUATOR] Wing Flags Set to: -15°
[ACTUATOR] Emergency: Raising Wing Flags
[ACTUATOR] Wing Flags Set to: -15°
[ACTUATOR] Emergency: Raising Wing Flags
[ACTUATOR] Wing Flags Set to: -25°
[ACTUATOR] Emergency: Raising Wing Flags
[ACTUATOR] Wing Flags Set to: -25°
[ACTUATOR] Emergency: Raising Wing Flags
```

Figure 18: Wing Flags System Outcome

7. Tail Flags Actuator System

The following code is for the tail flag system which works almost the same with all the actuator system.

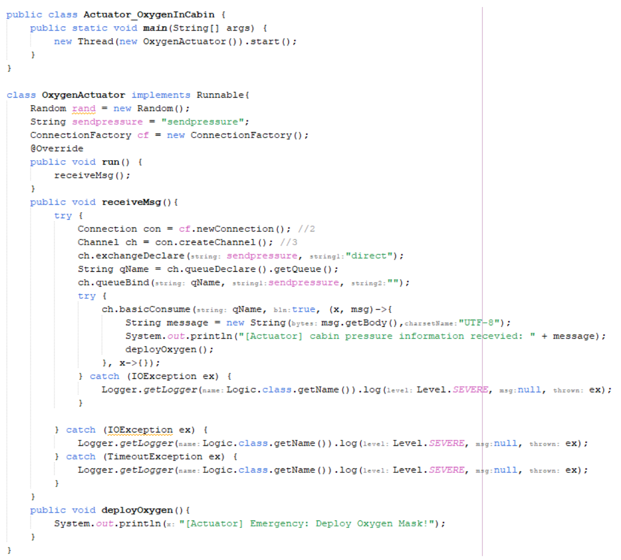


```java
class TailActuator implements Runnable {

    Random rand = new Random();
    String senddirection = "senddirection";
    ConnectionFactory cf = new ConnectionFactory();

    @Override
    public void run() {
        receiveMsg();
    }

    public void receiveMsg() {
        try {
            Connection con = cf.newConnection();
            Channel ch = con.createChannel();
            ch.exchangeDeclare(string: senddirection, string1:"direct");
            String qName = ch.queueDeclare().getQueue();
            ch.queueBind(string: qName, string1:senddirection, string2:"");
            try {
                ch.basicConsume(string: qName, bln:true, (x, msg) -> {
                    String message = new String(bytes: msg.getBody(), charsetName:"UTF-8");
                    System.out.println("[ACTUATOR] direction of plane is >20* : " + message);
                    deployTailFlaps();
                }, x -> {
                });
            } catch (IOException ex) {
                Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
            }

        } catch (IOException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        } catch (TimeoutException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        }
    }

    public void deployTailFlaps() {
        System.out.println(x: "[ACTUATOR] Deploy Wings to Keep Direction");
    }
}
```

Figure 19: Tail Flags Actuator System

8. Cabin Actuator System [Oxygen Mask]

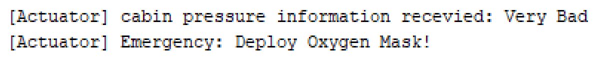
The Cabin actuator system will control manage the cabin pressure, and as a bonus mark, it will also deploy oxygen mask incase of a very low pressure inside the plane cabin.



```java
public class Actuator_OxygenInCabin {
    public static void main(String[] args) {
        new Thread(new OxygenActuator()).start();
    }
}

class OxygenActuator implements Runnable{
    Random rand = new Random();
    String sendpressure = "sendpressure";
    ConnectionFactory cf = new ConnectionFactory();
    @Override
    public void run() {
        receiveMsg();
    }
    public void receiveMsg(){
        try {
            Connection con = cf.newConnection(); //2
            Channel ch = con.createChannel(); //3
            ch.exchangeDeclare(string: sendpressure, string1:"direct");
            String qName = ch.queueDeclare().getQueue();
            ch.queueBind(string: qName, string1:sendpressure, string2:"");
            try {
                ch.basicConsume(string: qName, bln:true, (x, msg)->{
                    String message = new String(bytes: msg.getBody(),charsetName:"UTF-8");
                    System.out.println("[Actuator] cabin pressure information recevied: " + message);
                    deployOxygen();
                }, x->{});
            } catch (IOException ex) {
                Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
            }

        } catch (IOException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        } catch (TimeoutException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        }
    }
    public void deployOxygen(){
        System.out.println(x: "[Actuator] Emergency: Deploy Oxygen Mask!");
    }
}
```

Figure 20: Cabin Actuator System

If this class were to be ran, the following output will be printed:

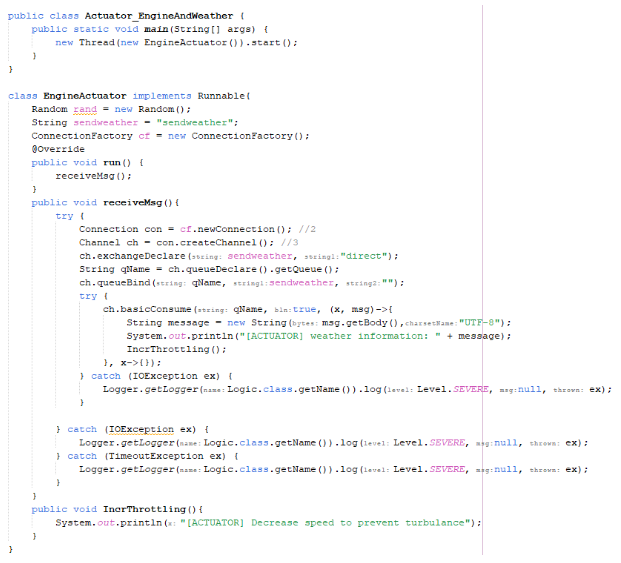


```
[Actuator] cabin pressure information recevied: Very Bad
[Actuator] Emergency: Deploy Oxygen Mask!
```

Figure 21: Cabin System Output

9. Engine Actuator System

The last actuator system, the engine will manage the speed of the plane, this works by receiving the weather information, and if the weather is currently Windy and the plane were to go full speed, the plane will go into turbulence which is considered inconvenience to the passengers, and vice versa, if the weather is clear, the function should speed up the plane.



```
public class Actuator_EngineAndWeather {
    public static void main(String[] args) {
        new Thread(new EngineActuator()).start();
    }
}

class EngineActuator implements Runnable{
    Random rand = new Random();
    String sendweather = "sendweather";
    ConnectionFactory cf = new ConnectionFactory();
    @Override
    public void run() {
        receiveMsg();
    }
    public void receiveMsg(){
        try {
            Connection con = cf.newConnection(); //2
            Channel ch = con.createChannel(); //3
            ch.exchangeDeclare(string: sendweather, string1:"direct");
            String qName = ch.queueDeclare().getQueue();
            ch.queueBind(string: qName, string1:sendweather, string2:"");
            try {
                ch.basicConsume(string: qName, bln:true, (x, msg)->{
                    String message = new String(bytes: msg.getBody(),charsetName:"UTF-8");
                    System.out.println("[ACTUATOR] weather information: " + message);
                    IncrThrottling();
                }, x->{});
            } catch (IOException ex) {
                Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
            }

        } catch (IOException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        } catch (TimeoutException ex) {
            Logger.getLogger(name: Logic.class.getName()).log(level: Level.SEVERE, msg:null, thrown: ex);
        }
    }
    public void IncrThrottling(){
        System.out.println(x: "[ACTUATOR] Decrease speed to prevent turbulance");
    }
}
```

Figure 22: Engine Actuator Code Snippet

In this section you provide a detailed explanation of the results of your tests and discuss your findings. Include screen shots, source code extracts, tables, and charts to support your discussion.

5.0 Conclusion

To conclude the project, the literature review done by the researcher has very much helped the researched to develop the system in the way the assignment question has requested by deepening the understanding of the researcher on various topics including real time systems, threads, and other important topics.

References

Abdallah, A. E., & Mohamed, A. E. (2019). Real-Time Systems: A Comprehensive Overview. *International Journal of Computer Applications (IJCA)*, 1-4.

Avery, D. (19994). Impact. *The Evolution of Flight Management Systems*, 1-4.

Bao, A. C. (2023, Mar 16). *Alibaba Cloud*. Retrieved from Is Rabbitmq Server Multithreaded: https://www.alibabacloud.com/tech-news/rabbitmq/1i8-is-rabbitmq-server-multithreaded#:~:text=RabbitMQ%20uses%20multithreading%20to%20handle,response%20times%20and%20better%20performance.

Ferdy, H., & Jansen, P. G. (2003). Journal of Control Engineering and Applied Informatics. *Real-time communication protocols: an overview*, 16-27.

Mohammadi, A., & G. Akl, S. (2005). Technical Report No. 2005-499. *Scheduling Algorithms for Real-Time Systems*, 5-11.

OpenJDK. (2023, Jan 12). *GitHub*. Retrieved from Java Microbenchmark Harness (JMH): https://github.com/openjdk/jmh

Oracle Corporation. (2022). *Oracle Java Documentation*. Retrieved from The Java Tutorials: https://www.oracle.com/java/