INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT087-3-3-RTS

REALTIME SYSTEMS

APU3F2211CS

HAND OUT DATE	**: 19th December 2022**
HAND IN DATE	**: 22nd March 2023**
WEIGHTAGE	**: 50%**

NAME	**:**	**David Wee Ming Junn**
STUDENT ID	**:**	**TP062496**
INTAKE CODE	**:**	**APU3F2211CS**
MODULE CODE	**:**	**CT087-3-3-RTS**
ASSIGNMENT	**:**	**INDIVIDUAL ASSIGNMENT**
LECTURER	**:**	**ASSOC. PROF. DR. IMRAN MEDI**

Investigating How Programming Design Effects Real-Time Performance

A Simulation of A Flight Control System

David Wee Ming Junn

TP062496

Abstract

This paper aims to explore the impact of programming design on the performance of flight management systems. The study includes the creation of a simulation of the system's core components - sensors, actuators, and the flight control system - with an explanation of the implementation and workflow. The systems mainly communicate with RabbitMQ tool, allowing data to be transferred from distinct thread to thread. The developer then analyzes and evaluates the simulation's performance based on different programming designs, utilizing benchmarks and JDK Mission Control software. For results that have little difference in performance, the sample size is expanded two folds or more. Additionally, a literature review is also conducted, to get a further in-depth understanding on real-time systems, issues of latency and solutions to measure performance of a software system. Overall, this paper demonstrates the methodology and performance outcomes resulting from the programming changes.

1.0 Introduction

According to TheHealthyJournal, roughly 100,000 flights take off and land every day all over the globe. Say an average length of a flight is two hours; that would mean that six million people fly somewhere every day. (TheHealthyJournal, n.d.) It was something that was once deemed impossible, yet it continued to grow and became a part of everyone's lives. But of course, soaring through the skies is no easy feat, tallying up to 229 fatalities from 12 fatal accidents in just 2022. (Learmount, 2023) Which is why passenger airplanes are required by law to have a flight management system.

A flight management system is a real-time systems that manages and coordinates the sensory part of the airplane which measures important attributes such as the altitude, cabin pressure and airplane speed, with the actuator part of airplanes which manages the plane by making a rotary or linear motion based on the stimuli input, such as the wing flaps, engine and cabin pressurization system. Or in a simple term, the flight management system is a virtual pilot that gets the data from the sensors, decide, and pass off those decision to the actuator systems. However, there is a catch, the system is considered as half of a hard real-time and half of a soft real-time. This is because the flight management system has both hard and soft real-time components, depending on the specific data involved and the consequences it brings if time constraints are not obeyed. For example, the altitude is considered as a data of high value, if the altitude is too low and it was subjected to delay, unable to meet the deadline, a collision might occur just because the wing flaps did not receive the instruction to move the airplane up in time, devastating untold lives. As a result, a developer is tasked to investigate the affect of programming design towards real-time performance. The main reason for this investigation is to explore different techniques and strategies that can be used to reduce latency within the flight management system to improve the performance of a passenger airliner flight management system.

The developer is required to create a simulation of a flight management system and analyse the speed, efficiency and reliability of the system in addition to documenting any modification that helped in improving performance. Moreover, the developer is expected to utilise a messaging system to facilitate communication between various components of the flight control system, sensor systems and actuator systems. All in all, this report exemplifies the developers attempts in simulating a flight management system, refining the performance of the system and detecting performance obstacles.

2.0 Research Background/Literature Review

2.1 What is latency?

(Bakar & Jamal, 2020) offers a comparison of studies and solutions related to latency issues in the Internet of Things (IoT) resource discovery. The study defines latency as one-way delay, which is the amount of time for data to move from one point to another. According to the study, hardware-produced latency is relatively constant, while software-produced latency varies greatly and is therefore difficult to measure directly.

The study emphasizes the importance of IoT latency solutions in enabling latency-sensitive applications to operate as planned. By filtering and evaluating latency issues, the IoT can be tailored to ensure that critical IoT latency requirements are met, as latency cannot be completely eliminated. The study also introduces methods for measuring latency, such as using timestamping packets to determine delays accurately without additional overhead.
However, the study raises questions that need to be addressed, such as "Why is it so important to reduce latency?" "Why can't latency-sensitive applications operate as expected because of latency?" and "How can an IoT be tailored to meet latency requirements?"

2.2 Hard & Soft Real Time System

The term “real-time system” refers to any information processing system with hardware and software components that perform real-time application functions and can respond to events within predictable and specific time constraints. (Intel, n.d.) Real Time system is categorized into hard and soft. Hard real time system(HRTS) has extremely strict and high standards on time constraints, as failure to meet deadline results in serious consequences. (GeeksForGeeks, n.d.) On the contrary, soft real time system (SRTS) offers some leeway in terms of deadlines. This is because although the operation or data starts to degrade as the deadline passes, no major repercussion would occur as a result of it.

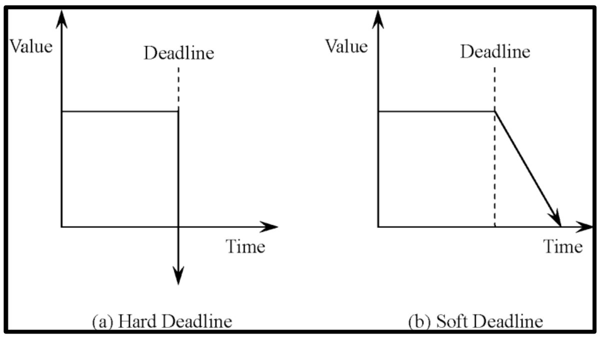


Figure 1: Hard vs Soft Deadline

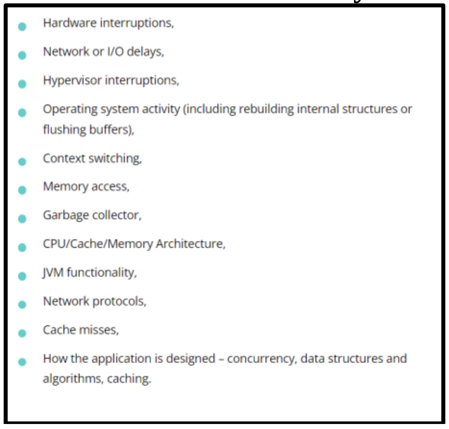
2.3 Understanding the effect of latency

(Claypool, 2005) researches the effect of latency on user performance in Real-Time Strategy games. The study conducts experiments with human participants playing RTS games with varying levels of network latency. With that, the study presents evidences that latency has a significant effect on user performance and user experience in the games conducted, with the user with high latency leading to slower user response times and decreased user performance. However, it is important to note that events that happen are consistent among all users, all users witness the same events unfolding with the high latency user seeing the event a little later, meaning latency does not stop data from being transferred, it just delays it. However, with the latency, it is noted that a latency of 500ms or more had a significant impact on the users’ performance in RTS games. Thus, to relate it to this report, if latency is low and at an acceptable range, the flight management system will be able to run smoothly. If latency is high, the flight management system will not be able to react properly according to the real-time stimuli from sensors.

2.4 Performance Evaluation with Benchmarking

There exist various methods to evaluate and gauge the performance of a system. Among these techniques, benchmarking holds significant importance in system development, a process of measuring products, services and processes. (ASQ, n.d.) However, selecting the appropriate benchmarking tool can be a challenging task. This is because the developer uses threads that are not expected to stop and RabbitMQ for the threads to communicate with each other. Therefore, in addition to testing the performance of the interaction between the threads and the processor, the developer needs to test the performance of said communication. But of course, due to the nature of the code, the thought of applying java micro-benchmarking(JMH) might be out of the question. This is because to apply JMH, specific rules need to be followed in regards to the annotations. Unfortunately, the starting point is in one thread while the stopping point is in another thread, making it difficult to use JMH. With that in mind, it is worth mentioning that benchmarking primarily provides a standardized and reproducible way to compare performance across different systems or components. Therefore, it should be considered to utilize performance monitoring tool instead, and compare the results manually. Although it may be less standardized than benchmarking, it is not inferior in terms of accuracy.

2.5 How to reduce Latency



- Hardware interruptions,
- Network or I/O delays,
- Hypervisor interruptions,
- Operating system activity (including rebuilding internal structures or flushing buffers),
- Context switching,
- Memory access,
- Garbage collector,
- CPU/Cache/Memory Architecture,
- JVM functionality,
- Network protocols,
- Cache misses,
- How the application is designed – concurrency, data structures and algorithms, caching.

Figure 2: Reasons for Latency (StratoFlow, n.d.)

According to (StratoFlow, n.d.), the factors listed in Figure 2 are the main causes of delays in applications or systems. Moreover, solutions to decrease latency include…

1. Choosing the correct programming language
2. Remember about memory and garbage collection.
3. Provide a system buffer.
4. Pay attention to caching.
5. Keep your code simple.

Among these solutions, solution 2 and 5 deserve a closer attention. In Java, garbage collection is an automatic process, the programmer does not need to explicitly mark objects to be deleted. (Nayak, 2021) However, it’s worth noting that operations that are running but does nothing are essentially garbage. These garbage are not marked by the garbage collector as it is assumed to be doing “something”. Therefore, it is the job of the developer to ensure such operations cease to run when they finish their tasks. For example, a thread awaiting messages that are never coming. Next in line, the code should be kept simple, avoiding any unnecessary actions. In other words, the code should be optimized in a way that finishes a requirement with the least of lines.

3.0 Methodology

3.1 Overview of Simulation

The purpose of this study is to analyse the performance of a simulated flight management system and search for ways to increase the performance of the system. As instructed by the case study, the flight management system is categorized into three sections, the sensory systems, flight control system and actuator systems as shown in Figure 3.

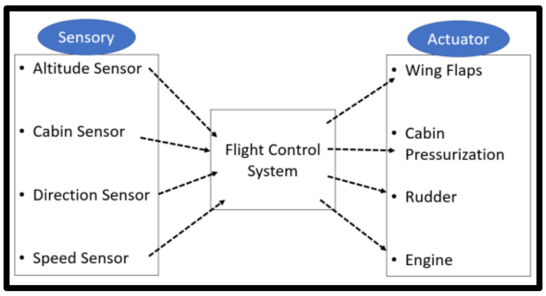
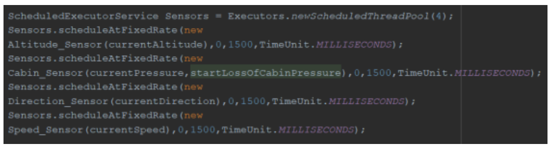


Figure 3: Visualization of Simulation's workflow

The simulation starts with the sensory systems generating inputs. The inputs are then sent to flight control system via RabbitMQ. When the flight control system receives the input, it analyses the data and subsequently arrives at a decision and sends the instruction to the intended actuator systems with RabbitMQ. In the end, the actuator systems receive and perform accordance to the acquired instruction.

3.2 Implementation

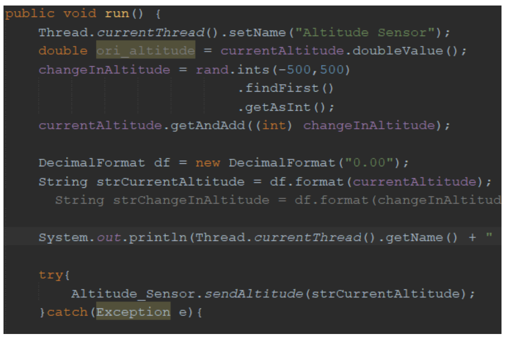
The entire code of file necessary for understanding the flow of the simulation will be available in the appendix. This section is merely to show the workflow of the simulation. Therefore, instead of the entire sensory systems, the entire flight control systems, the entire actuator systems, snippets of each systems will be shown and explained in details.



```
ScheduledExecutorService Sensors = Executors.newScheduledThreadPool(4);
Sensors.scheduleAtFixedRate(new
Altitude_Sensor(currentAltitude),0,1500,TimeUnit.MILLISECONDS);
Sensors.scheduleAtFixedRate(new
Cabin_Sensor(currentPressure,startLossOfCabinPressure),0,1500,TimeUnit.MILLISECONDS);
Sensors.scheduleAtFixedRate(new
Direction_Sensor(currentDirection),0,1500,TimeUnit.MILLISECONDS);
Sensors.scheduleAtFixedRate(new
Speed_Sensor(currentSpeed),0,1500,TimeUnit.MILLISECONDS);
```

Figure 4: Thread Execution

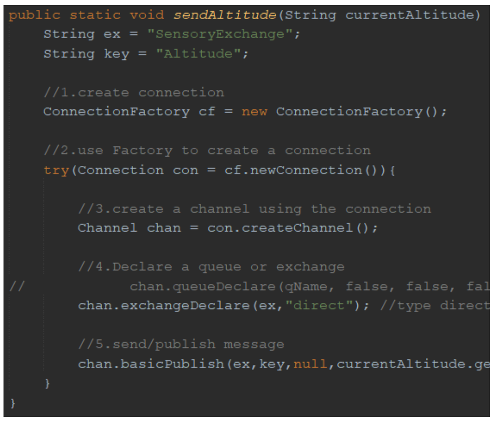
Figure 4 shows a snippet of the Main file, where threads of the sensory system is activated. Since each sensory is distinct from each other, the simulation has 4 threads for sensory system, 4 threads for actuator system and 2 threads for flight control system. As indicated, the sensory threads are executed once every 1500 milliseconds for reasons that will be explained below.



```
public void run() {
    Thread.currentThread().setName("Altitude Sensor");
    double ori_altitude = currentAltitude.doubleValue();
    changeInAltitude = rand.ints(-500,500)
                            .findFirst()
                            .getAsInt();
    currentAltitude.getAndAdd((int) changeInAltitude);

    DecimalFormat df = new DecimalFormat("0.00");
    String strCurrentAltitude = df.format(currentAltitude);
      String strChangeInAltitude = df.format(changeInAltitud

    System.out.println(Thread.currentThread().getName() + "

    try{
        Altitude_Sensor.sendAltitude(strCurrentAltitude);
    }catch(Exception e){
```

Figure 5:Modification of Altitude

Figure 5 is a snippet from the altitude sensor file. It mainly shows what the sensor does when it is executed. Since the simulation is primarily meant to simulate the reaction of the actuators depending on the stimuli, the sensors generate random measurements. In other words, the actions of the actuators are based off the sensors and not the other way round. The sensor works by first generating a random altitude between 500ft to -500ft and adding it to the current altitude. The current altitude is then sent to the flight control system with the function "sendAltitude()".



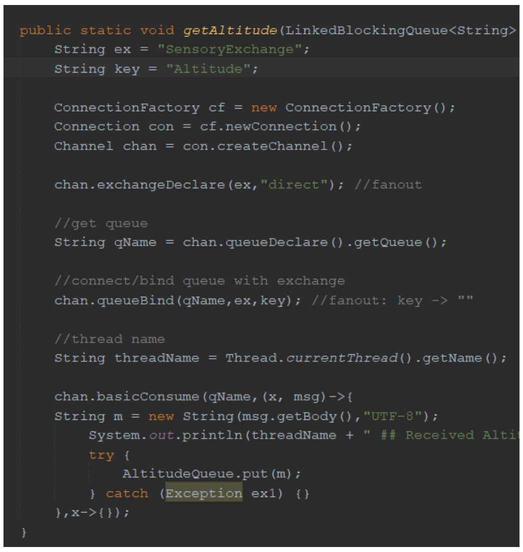
```
public static void sendAltitude(String currentAltitude)
    String ex = "SensoryExchange";
    String key = "Altitude";

    //1.create connection
    ConnectionFactory cf = new ConnectionFactory();

    //2.use Factory to create a connection
    try(Connection con = cf.newConnection()){

        //3.create a channel using the connection
        Channel chan = con.createChannel();

        //4.Declare a queue or exchange
//              chan.queueDeclare(qName, false, false, fal
        chan.exchangeDeclare(ex,"direct"); //type direct

        //5.send/publish message
        chan.basicPublish(ex,key,null,currentAltitude.ge
    }
}
```

Figure 6: Sending of Altitude

Before sending the data to flight control system, a connection to the flight control system's receiver must be made. The procedures are as of below…

1. Create an instance of 'Connection Factory'
   - Connection Factory is a class needed to help create a connection to a message broker.

2. Establish a connection to the message broker.
   - A message broker is software that enables applications, systems, and services to communicate with each other and exchange information. (IBM, n.d.)
3. Create a new channel using the connection.
   - Channel is a virtual connection inside a connection and publishing or consuming a message from queue will happen over a channel. (Tutlane, n.d.)
4. Declare an exchange of direct type.
   - The direct exchange will route the message to a specific queue based on exchange key which is "SensoryExchange".
5. The message is published to the exchange.
   - In addition to the routing key, a name is assigned to this message. Allowing only the intended receiver with the same key and name to access this message.



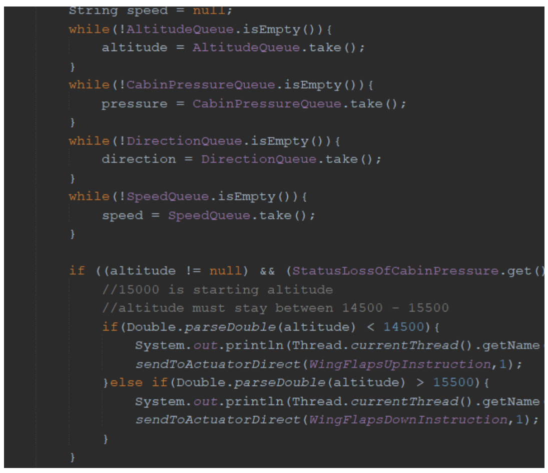
```java
public static void getAltitude(LinkedBlockingQueue<String>
    String ex = "SensoryExchange";
    String key = "Altitude";

    ConnectionFactory cf = new ConnectionFactory();
    Connection con = cf.newConnection();
    Channel chan = con.createChannel();

    chan.exchangeDeclare(ex,"direct"); //fanout

    //get queue
    String qName = chan.queueDeclare().getQueue();

    //connect/bind queue with exchange
    chan.queueBind(qName,ex,key); //fanout: key -> ""

    //thread name
    String threadName = Thread.currentThread().getName();

    chan.basicConsume(qName,(x, msg)->{
    String m = new String(msg.getBody(),"UTF-8");
        System.out.println(threadName + " ## Received Alti
        try {
            AltitudeQueue.put(m);
        } catch (Exception ex1) {}
    },x->{});
}
```

Figure 7: Flight Control System Altitude Receiver

Since data is sent from the sensory system, the flight control system must have receiver to receive these data. The procedure of receiving the data excluding connection creation is as follow…

1. Declare a new message queue on the channel and get its name.
2. Binds the message queue to the exchange using the routing key, name of queue, and name of message.
3. Register a consumer to receive messages from the message queue. The received messages are then processed by the lambda expression, which then retrieves the message.

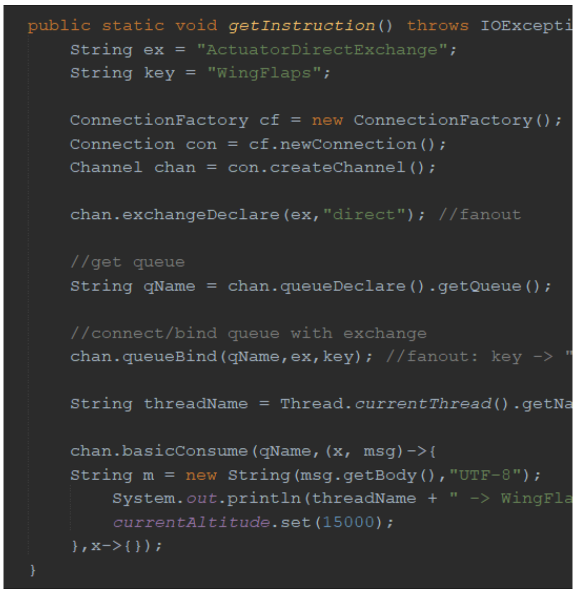
When the message is received, it is put into a linked blocking queue named AltitudeQueue.



```java
        String speed = null;
        while(!AltitudeQueue.isEmpty()){
            altitude = AltitudeQueue.take();
        }
        while(!CabinPressureQueue.isEmpty()){
            pressure = CabinPressureQueue.take();
        }
        while(!DirectionQueue.isEmpty()){
            direction = DirectionQueue.take();
        }
        while(!SpeedQueue.isEmpty()){
            speed = SpeedQueue.take();
        }

        if ((altitude != null) && (StatusLossOfCabinPressure.get()
            //15000 is starting altitude
            //altitude must stay between 14500 - 15500
            if(Double.parseDouble(altitude) < 14500){
                System.out.println(Thread.currentThread().getName
                sendToActuatorDirect(WingFlapsUpInstruction,1);
            }else if(Double.parseDouble(altitude) > 15500){
                System.out.println(Thread.currentThread().getName
                sendToActuatorDirect(WingFlapsDownInstruction,1);
            }
        }
```

Figure 8: Flight Control System Decision Making for Altitude

Just like altitude sensor, the other three sensory system does the same. Hence, every time the thread representing the flight control system activates, it clears the queues until the very last or recent measurement. If altitude is below or above the intended rate, the flight control system sends an instruction towards the wing flaps. The method of sending the instruction to the actuator system is no different from the altitude sensor sending the measurement to the flight control system. The only difference in this transaction is the exchange key and the name of the message.



```
public static void getInstruction() throws IOExcepti
    String ex = "ActuatorDirectExchange";
    String key = "WingFlaps";

    ConnectionFactory cf = new ConnectionFactory();
    Connection con = cf.newConnection();
    Channel chan = con.createChannel();

    chan.exchangeDeclare(ex,"direct"); //fanout

    //get queue
    String qName = chan.queueDeclare().getQueue();

    //connect/bind queue with exchange
    chan.queueBind(qName,ex,key); //fanout: key -> "

    String threadName = Thread.currentThread().getNa

    chan.basicConsume(qName,(x, msg)->{
    String m = new String(msg.getBody(),"UTF-8");
        System.out.println(threadName + " -> WingFla
        currentAltitude.set(15000);
    },x->{});
}
```

Figure 9: Wing Flaps Response

When an instruction is received, the wing flaps follow the instruction and reset the altitude to its original value of 15000 feet. Although this reset may seem unorthodox, it is crucial for the simulation because the altitude sensor randomly increments and decrements the altitude, which may result in continuous indication of low altitude despite the wing flaps' constant efforts to increase altitude. For the sample output, please refer to the appendix.

3.3 Performance Analysis

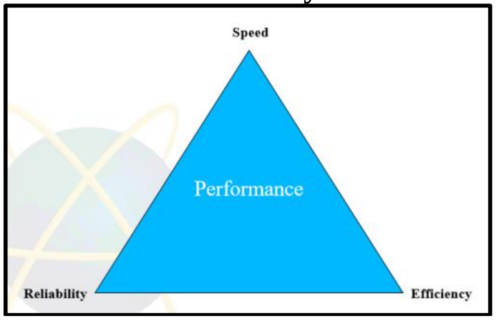
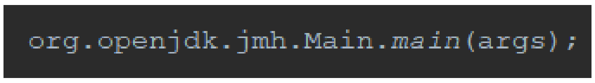


Figure 10: Realtime Development Considerations

The performance of this system is categorized into three separate category; speed, reliability and efficiency. Before improvement can be made to the simulation, the current performance of the simulation must be measured and recorded.

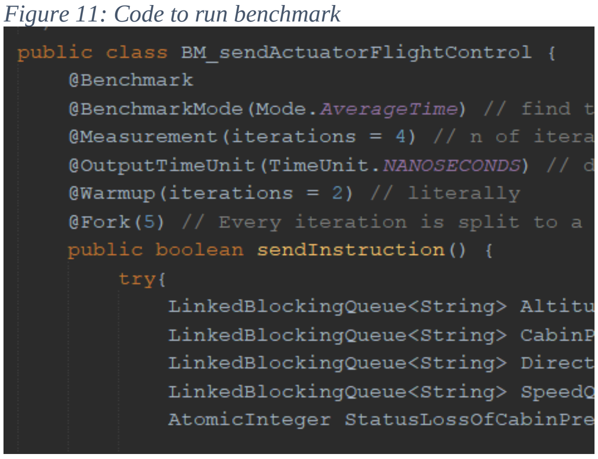
3.3.1 Micro-benchmarking

Micro-benchmarking is used in this simulation to evaluate the execution time of the threads. However, as previously mentioned, benchmarking thread is difficult or is above the ability of the developer. Therefore, a separate replica of a thread is created for benchmarking. Additionally, instead of creating a replica for every threads, for the purpose of identifying obstruction and improving performance, only a replica is created on the thread that has the most operation; the flight control system.



```
org.openjdk.jmh.Main.main(args);
```

Figure 11: Code to run benchmark



```
public class BM_sendActuatorFlightControl {
    @Benchmark
    @BenchmarkMode(Mode.AverageTime) // find t
    @Measurement(iterations = 4) // n of itera
    @OutputTimeUnit(TimeUnit.NANOSECONDS) // d
    @Warmup(iterations = 2) // literally
    @Fork(5) // Every iteration is split to a
    public boolean sendInstruction() {
        try{
            LinkedBlockingQueue<String> Altitu
            LinkedBlockingQueue<String> CabinP
            LinkedBlockingQueue<String> Direct
            LinkedBlockingQueue<String> SpeedQ
            AtomicInteger StatusLossOfCabinPre
```

Figure 12: Micro-Benchmarking Configuration in Flight Control System

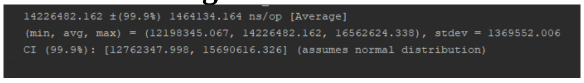
1. @BenchmarkMode(Mode.Average Time) notifies the Java Microbenchmark (JMH) to compute the average time taken for each benchmark execution. In other words, to get accurate measurement, the benchmark method is ran 4 times as stated in the @Measurement.
2. @Measurement(iterations = 4). As stated, this annotation specifies the number of iterations the benchmark will be ran.
3. @OutputTimeUnit(TimeUnit.NANOSECONDS). This annotation displays the results in nanoseconds. It could also be changed to another units such as seconds or milliseconds depending on the sensitivity required for the program.
4. @Warmup(iterations = 2). This annotation shows that the benchmark will be executed once as

warmup. This is to minimize inaccuracy of the benchmark by warming up the java virtual machine in advance.

5. @Fork(5). According to the information given above, the program will be officially iterated 4 times and ran 5 times in separate JVM instances to help smooth out any variation in performance caused by factors such as caching, CPU scheduling and other system-level effects.

4.0 Results and Changes

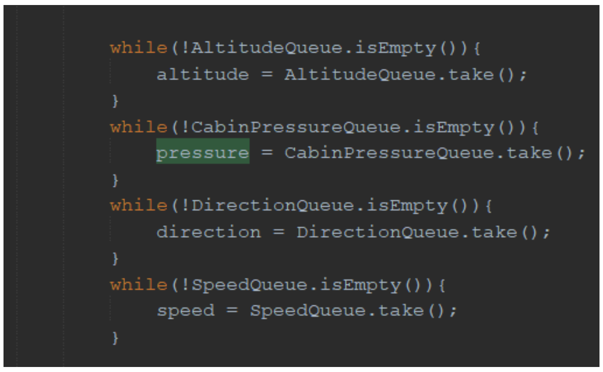
4.1 First Configuration - Threads



```
14226482.162 ±(99.9%) 1464134.164 ns/op [Average]
(min, avg, max) = (12198345.067, 14226482.162, 16562624.338), stdev = 1369552.006
CI (99.9%): [12762347.998, 15690616.326] (assumes normal distribution)
```

Figure 13: Benchmarking result of Flight Control System Replica

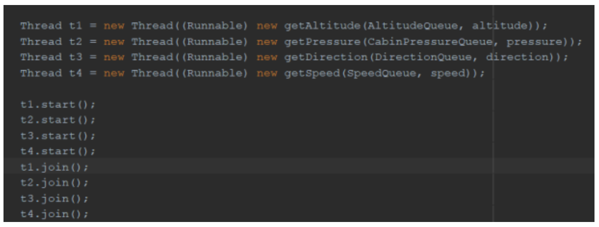
According to Figure 13, the average time taken to complete one operation is of 14226482.162 nanoseconds. With the lowest time at 12198345.067ns and the highest time at 16562624.338ns. Additionally, the standard deviation 1369552.006 indicates that the results are somewhat varied but not excessively so.



```
while(!AltitudeQueue.isEmpty()){
    altitude = AltitudeQueue.take();
}
while(!CabinPressureQueue.isEmpty()){
    pressure = CabinPressureQueue.take();
}
while(!DirectionQueue.isEmpty()){
    direction = DirectionQueue.take();
}
while(!SpeedQueue.isEmpty()){
    speed = SpeedQueue.take();
}
```

Figure 14: Snippet to be changed (4.1)

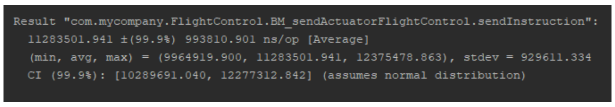
This snippet is from flight control system that yields data from Linked Blocking Queues, attempting to get the tail data or also known as the most recent data. This snippet of code is chosen to be modified as it loops numerous times until the queues are empty.



```
Thread t1 = new Thread((Runnable) new getAltitude(AltitudeQueue, altitude));
Thread t2 = new Thread((Runnable) new getPressure(CabinPressureQueue, pressure));
Thread t3 = new Thread((Runnable) new getDirection(DirectionQueue, direction));
Thread t4 = new Thread((Runnable) new getSpeed(SpeedQueue, speed));

t1.start();
t2.start();
t3.start();
t4.start();
t1.join();
t2.join();
t3.join();
t4.join();
```

Figure 15: Utilization of threads in the snippet

Instead of sequential execution, threads can be created to run the 4 distinct data procurement snippet in concurrent or depending on the core, parallelly. Either way, the utilization of threads significantly decreases the time required to acquire these data.



```
Result "com.mycompany.FlightControl.BM_sendActuatorFlightControl.sendInstruction":
  11283501.941 ±(99.9%) 993810.901 ns/op [Average]
  (min, avg, max) = (9964919.900, 11283501.941, 12375478.863), stdev = 929611.334
  CI (99.9%): [10289691.040, 12277312.842] (assumes normal distribution)
```

Figure 16: Result after addition of threads

According to Figure 16, the average time taken to complete one operation is of 11283501.941 nanoseconds. With the lowest time at 10289691.040ns and the highest time at 12277312.842ns. Additionally, the standard deviation 929611.334 indicates lesser fluctuation or less jitter as compared to "before". With this result, it is evident that by transitioning the tasks into numerous threads, the operation completes at a much faster rate. In this scenario however, efficiency is sacrificed for the price of increasing speed. 14226482.162

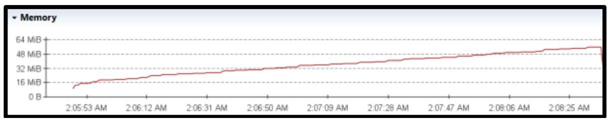


Figure 17: Memory usage of benchmarking before snippet modification

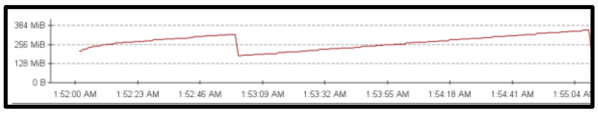
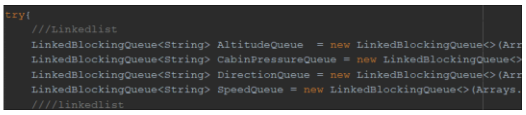


Figure 18: Memory usage of benchmarking after snippet modification

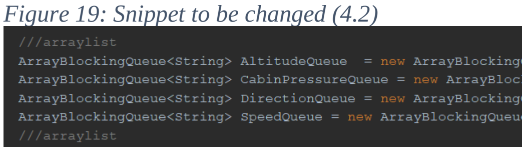
By comparing the memory usage of Figure 17 and Figure 18, it is clearly obvious that the operation with threads uses x3 or x4 the amount of memory to complete as compared to the operation without threads. Therefore, efficiency may be compromised in this situation. Flight control system can be considered as a hard real-time system. Therefore, high memory usage might be regarded as irrelevant if reducing latency is the number 1 concern. However, based on the tests, the average difference in completion time between the two is 2942980.221ns, which is 2.942980221ms. Due to the small difference in completion time, low memory usage may be the most efficient method as memory is not infinite and needs to be utilized efficiently. In summary, depending on the priority and hardware capabilities of the flight management system, the choice may vary.

4.2 Second Configuration – Data Structure



```
try{
    ///Linkedlist
    LinkedBlockingQueue<String> AltitudeQueue  = new LinkedBlockingQueue<>(Arr
    LinkedBlockingQueue<String> CabinPressureQueue = new LinkedBlockingQueue<>
    LinkedBlockingQueue<String> DirectionQueue = new LinkedBlockingQueue<>(Arr
    LinkedBlockingQueue<String> SpeedQueue = new LinkedBlockingQueue<>(Arrays.
    ////linkedlist
```

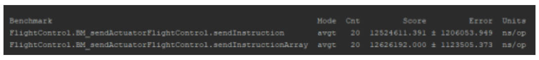
Figure 19: Snippet to be changed (4.2)



```
///arraylist
ArrayBlockingQueue<String> AltitudeQueue  = new ArrayBlocking
ArrayBlockingQueue<String> CabinPressureQueue = new ArrayBloc
ArrayBlockingQueue<String> DirectionQueue = new ArrayBlocking
ArrayBlockingQueue<String> SpeedQueue = new ArrayBlockingQueu
///arraylist
```

Figure 20: Snippet changed to array blocking queue

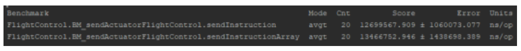
In this configuration, linked list queue is replaced to array list queue. A linked list is a sequence of data structures, which are connected together via links. (TutorialsPoint, n.d.) In other words, each element in the list maintains a reference to the pervious and next element. Therefore, making it suitable for removal and addition operations to the beginning and end of the list. Array list provides faster execution time for operations of accessing elements at random positions in the list. However, the addition and removal of element at the beginning or middle of the array list can be slow as it requires shifting all subsequent elements. As a result, arrays are theoretically not a good choice for queues, which will be tested below.



```
Benchmark                                                   Mode  Cnt         Score         Error  Units
FlightControl.BM_sendActuatorFlightControl.sendInstruction       avgt   20  12524611.391 ± 1206053.949  ns/op
FlightControl.BM_sendActuatorFlightControl.sendInstructionArray  avgt   20  12626192.000 ± 1123505.373  ns/op
```

Figure 21: Array vs Linked Benchmark Result

According to Figure 21, the average execution time with a linked list is 12,524,611.391ns, while the array list is at 12,626,192.000ns. Simple subtraction shows that the linked list is quicker by 101,580.609ns or 0.101580609ms. The tests were performed on four lists, each with 100 elements, and both operations were run simultaneously to avoid inaccuracies. Based on this result, it can be concluded that in this specific situation, the linked list is indeed faster than the array list, with a small margin difference. However, it is expected that this difference would increase exponentially as the data sample size increases.To prove the theory above, the benchmark is ran again after increasing the lists to 200 elements each, totalling up to 800 elements in total.



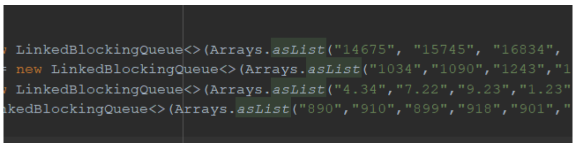
```
Benchmark                                                   Mode  Cnt         Score         Error  Units
FlightControl.BM_sendActuatorFlightControl.sendInstruction       avgt   20  12699567.909 ± 1060073.077  ns/op
FlightControl.BM_sendActuatorFlightControl.sendInstructionArray  avgt   20  13466752.946 ± 1438698.389  ns/op
```

Figure 22: Array vs Linked Benchmark Result (800elements)

After increasing the data size two folds, the difference in average execution time is 767,185.037. This is 665604.428ns more than the previous time difference. Hence, it is proven that linked list is indeed more suitable than array list in this operation, when the data size increases, the execution time difference increases as well.

In conclusion, by utilizing different data structure, the speed of the simulation can be increased. Both lists are data structures used to store and manipulate collections of elements. However, they possess distinct characteristics and approaches, resulting in significant changes in program behaviour depending on the situation.

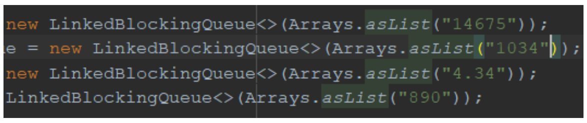
4.3 Third Configuration – Optimize Workflow



```
w LinkedBlockingQueue<>(Arrays.asList("14675", "15745", "16834",
= new LinkedBlockingQueue<>(Arrays.asList("1034","1090","1243","1
w LinkedBlockingQueue<>(Arrays.asList("4.34","7.22","9.23","1.23"
nkedBlockingQueue<>(Arrays.asList("890","910","899","918","901","
```

Figure 23: Snippet to be changed (4.3)

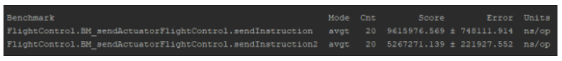
Figure 23 is the linked blocking queue’s declaration. The size of the queue is currently set to 100 elements for a better visualization of the difference in execution time. However, what if the maximum size of the queue is set to 1? What will the difference in execution time be?



```
new LinkedBlockingQueue<>(Arrays.asList("14675"));
e = new LinkedBlockingQueue<>(Arrays.asList("1034"));
new LinkedBlockingQueue<>(Arrays.asList("4.34"));
LinkedBlockingQueue<>(Arrays.asList("890"));
```

Figure 24: Configured snippet (4.3)

In Figure 24, it shows that each queue only has one element each. In summary, the third configuration imitates a situation whereas elements within the queue is constantly replaced with the latest value, drastically reducing the amount of iteration needed for the operation to acquire the desired data.

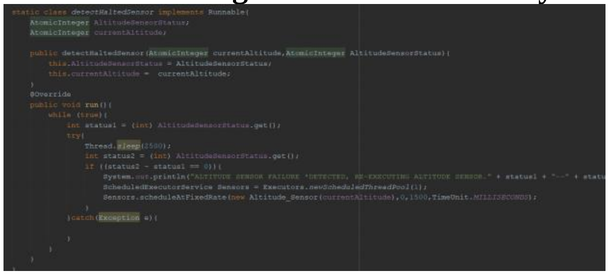


```
Benchmark                                                  Mode  Cnt        Score        Error  Units
FlightControl.BM_sendActuatorFlightControl.sendInstruction  avgt   20  9615976.569 ± 748111.914  ns/op
FlightControl.BM_sendActuatorFlightControl.sendInstruction2 avgt   20  5267271.139 ± 221927.552  ns/op
```

Figure 25: Result of Optimized Code Benchmark

When iteration is reduced to one for each queue, a whopping 45% decrease in execution time, with 4348705.43ns. Overall, optimized workflow refers to the action of using alternative coding approach to acquire desired result in a shorter time frame. One of the primary examples is loops. For every iteration, the execution time increases. Thus, by reducing the number of iteration, the speed is bound to be shortened.

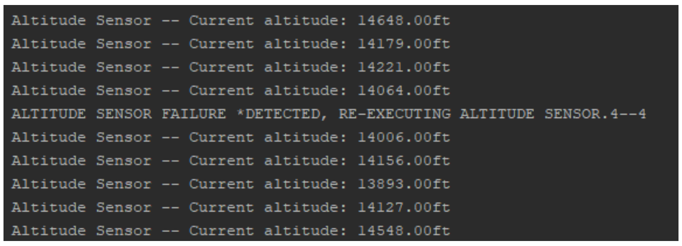
4.4 Fourth Configuration – Redundancy



```
static class detectHaltedSensor implements Runnable{
    AtomicInteger AltitudeSensorStatus;
    AtomicInteger currentAltitude;

    public detectHaltedSensor(AtomicInteger currentAltitude,AtomicInteger AltitudeSensorStatus){
        this.AltitudeSensorStatus = AltitudeSensorStatus;
        this.currentAltitude =  currentAltitude;
    }
    @Override
    public void run(){
        while (true){
            int status1 = (int) AltitudeSensorStatus.get();
            try{
                Thread.sleep(2500);
                int status2 = (int) AltitudeSensorStatus.get();
                if ((status2 - status1 == 0)){
                    System.out.println("ALTITUDE SENSOR FAILURE *DETECTED, RE-EXECUTING ALTITUDE SENSOR." + status1 + "--" + statu
                    ScheduledExecutorService Sensors = Executors.newScheduledThreadPool(1);
                    Sensors.scheduleAtFixedRate(new Altitude_Sensor(currentAltitude),0,1500,TimeUnit.MILLISECONDS);
                }
            }catch(Exception e){

            }
        }
    }
}
```

Figure 26: Altitude_SensorBackup Snippet

The backup mechanism works by first listening to messages sent by the altitude sensor. When a message is received, it increments an atomic integer called AltitudeSensorStatus. It also creates a thread (see Figure 26) that retrieves the current element from AltitudeSensorStatus and waits for 2500 milliseconds before retrieving it again. This is because the altitude sensor sends a message every 1500 milliseconds. If the sensor fails to send a message within 2500 milliseconds, the element within AltitudeSensorStatus will remain the same. By simple arithmetic subtraction, it can be determined whether the altitude sensor is functioning properly or has crashed prematurely. Subsequently, when it detects that the altitude sensor thread has stopped running for 2500ms, it creates a new altitude sensor thread, to keep the simulation running.



```
Altitude Sensor -- Current altitude: 14648.00ft
Altitude Sensor -- Current altitude: 14179.00ft
Altitude Sensor -- Current altitude: 14221.00ft
Altitude Sensor -- Current altitude: 14064.00ft
ALTITUDE SENSOR FAILURE *DETECTED, RE-EXECUTING ALTITUDE SENSOR.4--4
Altitude Sensor -- Current altitude: 14006.00ft
Altitude Sensor -- Current altitude: 14156.00ft
Altitude Sensor -- Current altitude: 13893.00ft
Altitude Sensor -- Current altitude: 14127.00ft
Altitude Sensor -- Current altitude: 14548.00ft
```

Figure 27: Altitude_SensorBackup Result

In this example (Figure 27), the altitude sensor is set to stop after 4 iteration. The Altitude_SensorBackup has successfully picked up on that and ran the altitude sensor thread again.

This configuration is known as redundancy. By creating a backup thread, the reliability aspect of the simulation can be protected. However, as illustrated in the first configuration, the use of additional threads leads to higher usage of memory. Hence, efficiency is significantly impacted if each sensor, actuator and the flight control system need to have a backup thread each. The memory usage is bound to increase by two folds or more.

5.0 Discussion

As shown in Figure 10, speed, reliability and efficiency is inter-related to performance. When one is improved, the others are compromised, such is the nature of software systems. In the result findings, when attempting to improve speed and reliability of the simulation, the memory would be significantly affected, efficiency is breached. Memory and processing power of hardware is limited, forcing a middle ground to be created between the three aspect of performance. One can’t only focus on a single aspect of the performance and expects the system to work as expected. All in all, above are the developer’s attempt to improve performance of the simulation, detailing the positive and negative effects of the configurations, signifying the impact of programming designs towards performance.

References

ASQ, n.d. *WHAT IS BENCHMARKING?.* [Online]
Available at: https://asq.org/quality-resources/benchmarking
[Accessed 17 March 2023].

Bakar, M. T. A. & Jamal, A. A., 2020. *Latency Issues in Internet of Things: A Review of Literature and Solution,* s.l.: s.n.

Claypool, M., 2005. *531e87a16aa501d3c3f7745071d082ba73d89792,* Worcester: s.n.

GeeksForGeeks, n.d. *Difference between Hard real time and Soft real time system.* [Online]
Available at: https://www.geeksforgeeks.org/difference-between-hard-real-time-and-soft-real-time-system/
[Accessed 17 March 2023].

IBM, n.d. *Message Brokers.* [Online]
Available at: https://www.ibm.com/my-en/topics/message-brokers
[Accessed 17 March 2023].

Intel, n.d. *The Need for Real-Time Systems.* [Online]
Available at: https://www.intel.com/content/www/us/en/robotics/real-time-systems.html
[Accessed 17 March 2023].

Learmount, D., 2023. *Deadly airline accidents reduced, but fatalities climbed in 2022.* [Online]
Available at: https://www.flightglobal.com/air-transport/deadly-airline-accidents-reduced-but-fatalities-climbed-in-2022/151508.article
[Accessed 3 June 2023].

Nayak, S., 2021. *Garbage Collection in Java – What is GC and How it Works in the JVM.* [Online]
Available at: https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm/#:~:text=Java%20garbage%20collection%20is%20an,implementation%20lives%20in%20the%20JVM.
[Accessed 17 March 2023].

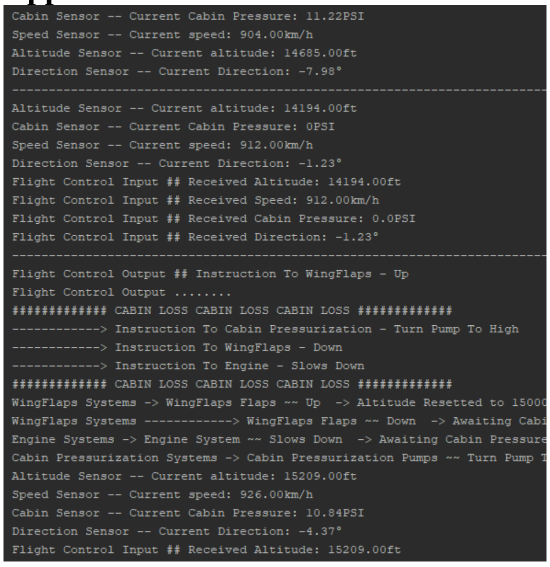
StratoFlow, n.d. *How to Build Low Latency Java Applications.* [Online]
Available at: https://stratoflow.com/low-latency-java-applications/
[Accessed 17 March 2023].

TheHealthyJournal, n.d. *How many planes take off a day?.* [Online]
Available at: https://www.thehealthyjournal.com/faq/how-many-planes-take-off-a-day#:~:text=Roughly%20100%2C000%20flights%20take%20off,people%20fly%20somewhere%20every%20day.
[Accessed 6 March 2023].

Tutlane, n.d. *RabbitMQ Channels.* [Online]
Available at: https://www.tutlane.com/tutorial/rabbitmq/rabbitmq-channels
[Accessed 17 March 2023].

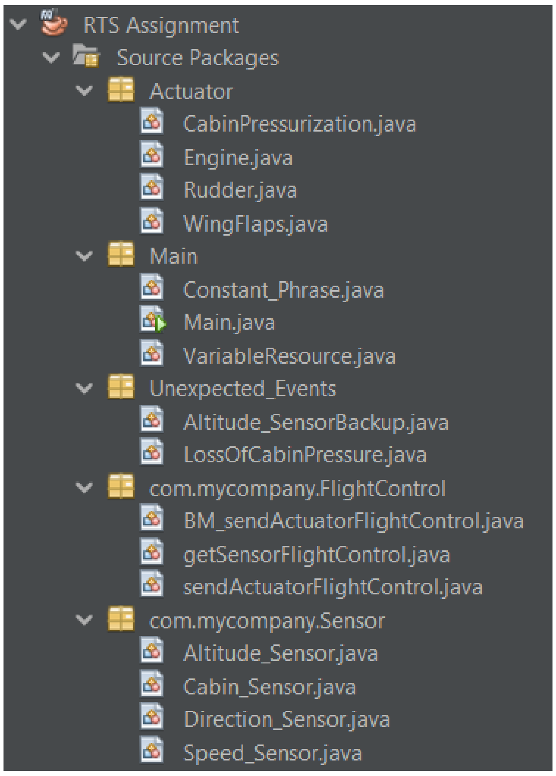
TutorialsPoint, n.d. *Data Structure and Algorithms - Linked List.* [Online]
Available at: https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorithms.htm
[Accessed 19 March 2023].

Appendix



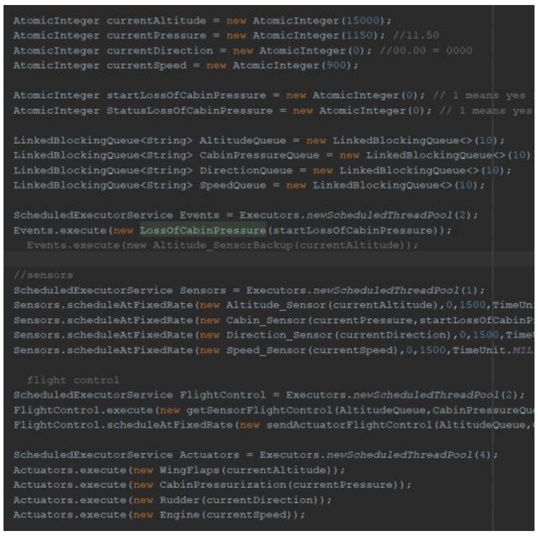
```
Cabin Sensor -- Current Cabin Pressure: 11.22PSI
Speed Sensor -- Current speed: 904.00km/h
Altitude Sensor -- Current altitude: 14685.00ft
Direction Sensor -- Current Direction: -7.98°
---------------------------------------------------------------------
Altitude Sensor -- Current altitude: 14194.00ft
Cabin Sensor -- Current Cabin Pressure: 0PSI
Speed Sensor -- Current speed: 912.00km/h
Direction Sensor -- Current Direction: -1.23°
Flight Control Input ## Received Altitude: 14194.00ft
Flight Control Input ## Received Speed: 912.00km/h
Flight Control Input ## Received Cabin Pressure: 0.0PSI
Flight Control Input ## Received Direction: -1.23°
---------------------------------------------------------------------
Flight Control Output ## Instruction To WingFlaps - Up
Flight Control Output ........
############# CABIN LOSS CABIN LOSS CABIN LOSS #############
------------> Instruction To Cabin Pressurization - Turn Pump To High
------------> Instruction To WingFlaps - Down
------------> Instruction To Engine - Slows Down
############# CABIN LOSS CABIN LOSS CABIN LOSS #############
WingFlaps Systems -> WingFlaps Flaps ~~ Up  -> Altitude Resetted to 15000
WingFlaps Systems ------------> WingFlaps Flaps ~~ Down  -> Awaiting Cabi
Engine Systems -> Engine System ~~ Slows Down  -> Awaiting Cabin Pressure
Cabin Pressurization Systems -> Cabin Pressurization Pumps ~~ Turn Pump T
Altitude Sensor -- Current altitude: 15209.00ft
Speed Sensor -- Current speed: 926.00km/h
Cabin Sensor -- Current Cabin Pressure: 10.84PSI
Direction Sensor -- Current Direction: -4.37°
Flight Control Input ## Received Altitude: 15209.00ft
```

Figure 28: Sample Output



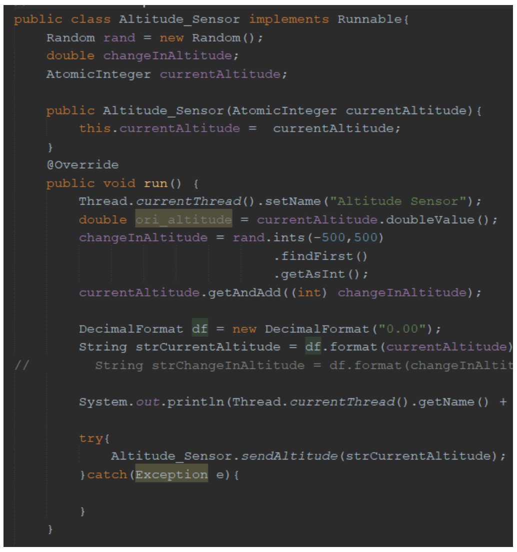
```
RTS Assignment
  Source Packages
    Actuator
      CabinPressurization.java
      Engine.java
      Rudder.java
      WingFlaps.java
    Main
      Constant_Phrase.java
      Main.java
      VariableResource.java
    Unexpected_Events
      Altitude_SensorBackup.java
      LossOfCabinPressure.java
    com.mycompany.FlightControl
      BM_sendActuatorFlightControl.java
      getSensorFlightControl.java
      sendActuatorFlightControl.java
    com.mycompany.Sensor
      Altitude_Sensor.java
      Cabin_Sensor.java
      Direction_Sensor.java
      Speed_Sensor.java
```

Figure 29: Files associated



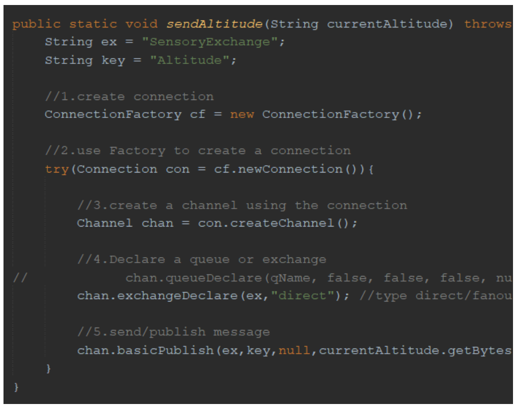
```java
AtomicInteger currentAltitude = new AtomicInteger(15000);
AtomicInteger currentPressure = new AtomicInteger(1150); //11.50
AtomicInteger currentDirection = new AtomicInteger(0); //00.00 = 0000
AtomicInteger currentSpeed = new AtomicInteger(900);

AtomicInteger startLossOfCabinPressure = new AtomicInteger(0); // 1 means yes
AtomicInteger StatusLossOfCabinPressure = new AtomicInteger(0); // 1 means yes

LinkedBlockingQueue<String> AltitudeQueue = new LinkedBlockingQueue<>(10);
LinkedBlockingQueue<String> CabinPressureQueue = new LinkedBlockingQueue<>(10)
LinkedBlockingQueue<String> DirectionQueue = new LinkedBlockingQueue<>(10);
LinkedBlockingQueue<String> SpeedQueue = new LinkedBlockingQueue<>(10);

ScheduledExecutorService Events = Executors.newScheduledThreadPool(2);
Events.execute(new LossOfCabinPressure(startLossOfCabinPressure));
  Events.execute(new Altitude_SensorBackup(currentAltitude));

//sensors
ScheduledExecutorService Sensors = Executors.newScheduledThreadPool(1);
Sensors.scheduleAtFixedRate(new Altitude_Sensor(currentAltitude),0,1500,TimeUn
Sensors.scheduleAtFixedRate(new Cabin_Sensor(currentPressure,startLossOfCabinP
Sensors.scheduleAtFixedRate(new Direction_Sensor(currentDirection),0,1500,Time
Sensors.scheduleAtFixedRate(new Speed_Sensor(currentSpeed),0,1500,TimeUnit.MIL

  flight control
ScheduledExecutorService FlightControl = Executors.newScheduledThreadPool(2);
FlightControl.execute(new getSensorFlightControl(AltitudeQueue,CabinPressureQu
FlightControl.scheduleAtFixedRate(new sendActuatorFlightControl(AltitudeQueue,

ScheduledExecutorService Actuators = Executors.newScheduledThreadPool(4);
Actuators.execute(new WingFlaps(currentAltitude));
Actuators.execute(new CabinPressurization(currentPressure));
Actuators.execute(new Rudder(currentDirection));
Actuators.execute(new Engine(currentSpeed));
```

Figure 30: Main File Source Code



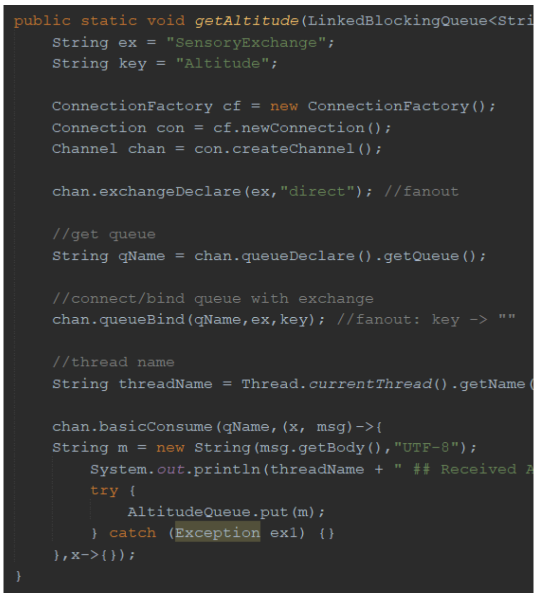
```java
public class Altitude_Sensor implements Runnable{
    Random rand = new Random();
    double changeInAltitude;
    AtomicInteger currentAltitude;

    public Altitude_Sensor(AtomicInteger currentAltitude){
        this.currentAltitude =  currentAltitude;
    }
    @Override
    public void run() {
        Thread.currentThread().setName("Altitude Sensor");
        double ori_altitude = currentAltitude.doubleValue();
        changeInAltitude = rand.ints(-500,500)
                                    .findFirst()
                                    .getAsInt();
        currentAltitude.getAndAdd((int) changeInAltitude);

        DecimalFormat df = new DecimalFormat("0.00");
        String strCurrentAltitude = df.format(currentAltitude)
//          String strChangeInAltitude = df.format(changeInAltit

        System.out.println(Thread.currentThread().getName() +

        try{
            Altitude_Sensor.sendAltitude(strCurrentAltitude);
        }catch(Exception e){

        }
    }
```

Figure 31: Altitude_Sensor file Run Method



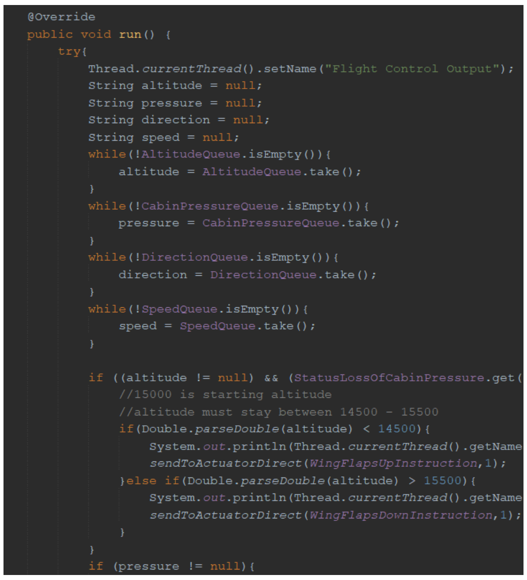
```java
public static void sendAltitude(String currentAltitude) throws
    String ex = "SensoryExchange";
    String key = "Altitude";

    //1.create connection
    ConnectionFactory cf = new ConnectionFactory();

    //2.use Factory to create a connection
    try(Connection con = cf.newConnection()){

        //3.create a channel using the connection
        Channel chan = con.createChannel();

        //4.Declare a queue or exchange
//              chan.queueDeclare(qName, false, false, false, nu
        chan.exchangeDeclare(ex,"direct"); //type direct/fanou

        //5.send/publish message
        chan.basicPublish(ex,key,null,currentAltitude.getBytes
    }
}
```

Figure 32: Altitude_Sensor file sendAltitude Method



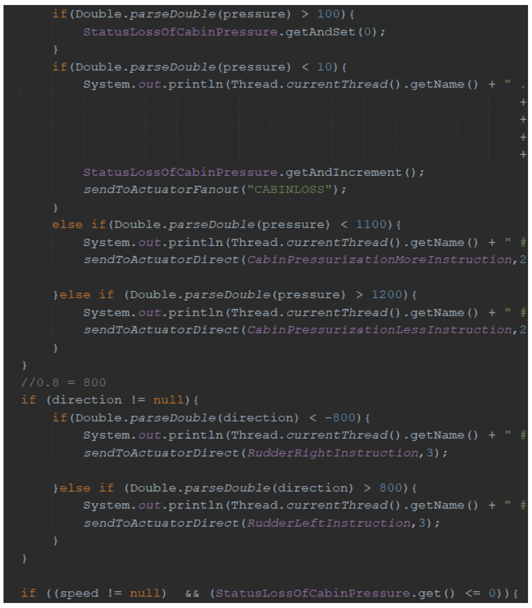
```java
public static void getAltitude(LinkedBlockingQueue<Stri
    String ex = "SensoryExchange";
    String key = "Altitude";

    ConnectionFactory cf = new ConnectionFactory();
    Connection con = cf.newConnection();
    Channel chan = con.createChannel();

    chan.exchangeDeclare(ex,"direct"); //fanout

    //get queue
    String qName = chan.queueDeclare().getQueue();

    //connect/bind queue with exchange
    chan.queueBind(qName,ex,key); //fanout: key -> ""

    //thread name
    String threadName = Thread.currentThread().getName(

    chan.basicConsume(qName,(x, msg)->{
    String m = new String(msg.getBody(),"UTF-8");
        System.out.println(threadName + " ## Received A
        try {
            AltitudeQueue.put(m);
        } catch (Exception ex1) {}
    },x->{});
}
```

Figure 33: FlightControlSystem file getAltitude Method



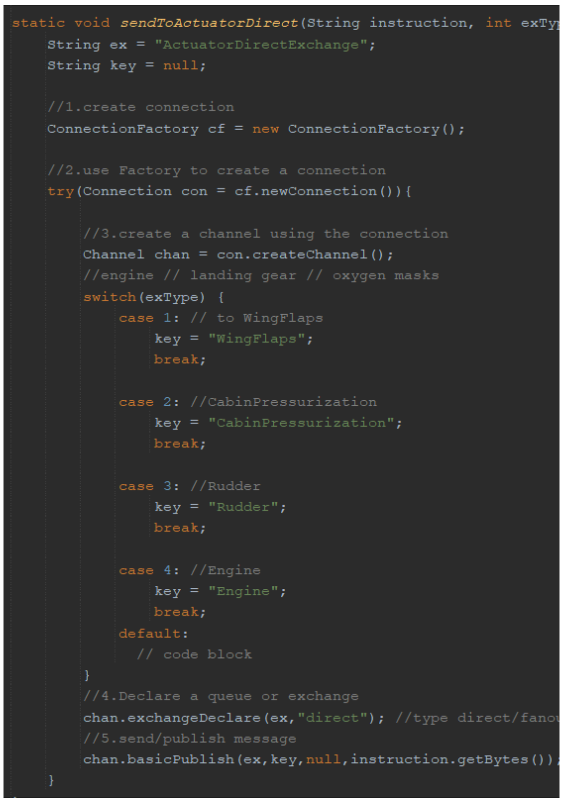
```
@Override
public void run() {
    try{
        Thread.currentThread().setName("Flight Control Output");
        String altitude = null;
        String pressure = null;
        String direction = null;
        String speed = null;
        while(!AltitudeQueue.isEmpty()){
            altitude = AltitudeQueue.take();
        }
        while(!CabinPressureQueue.isEmpty()){
            pressure = CabinPressureQueue.take();
        }
        while(!DirectionQueue.isEmpty()){
            direction = DirectionQueue.take();
        }
        while(!SpeedQueue.isEmpty()){
            speed = SpeedQueue.take();
        }

        if ((altitude != null) && (StatusLossOfCabinPressure.get(
            //15000 is starting altitude
            //altitude must stay between 14500 - 15500
            if(Double.parseDouble(altitude) < 14500){
                System.out.println(Thread.currentThread().getName
                sendToActuatorDirect(WingFlapsUpInstruction,1);
            }else if(Double.parseDouble(altitude) > 15500){
                System.out.println(Thread.currentThread().getName
                sendToActuatorDirect(WingFlapsDownInstruction,1);
            }
        }
        if (pressure != null){
```

Figure 34: FlightControlSystem file Run Method(1)



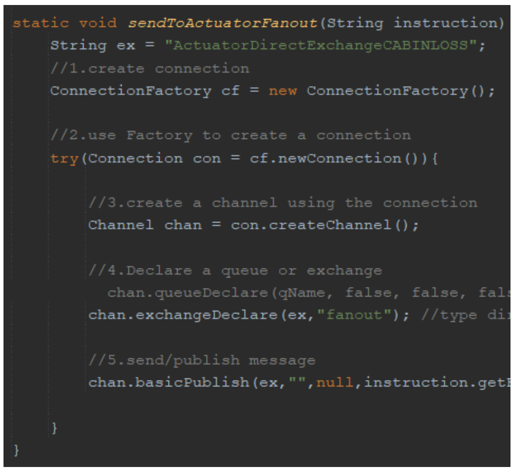
```
    if(Double.parseDouble(pressure) > 100){
        StatusLossOfCabinPressure.getAndSet(0);
    }
    if(Double.parseDouble(pressure) < 10){
        System.out.println(Thread.currentThread().getName() + " .
                                                                  +
                                                                  +
                                                                  +
                                                                  +
        StatusLossOfCabinPressure.getAndIncrement();
        sendToActuatorFanout("CABINLOSS");
    }
    else if(Double.parseDouble(pressure) < 1100){
        System.out.println(Thread.currentThread().getName() + " #
        sendToActuatorDirect(CabinPressurizationMoreInstruction,2

    }else if (Double.parseDouble(pressure) > 1200){
        System.out.println(Thread.currentThread().getName() + " #
        sendToActuatorDirect(CabinPressurizationLessInstruction,2
    }
}
//0.8 = 800
if (direction != null){
    if(Double.parseDouble(direction) < -800){
        System.out.println(Thread.currentThread().getName() + " #
        sendToActuatorDirect(RudderRightInstruction,3);

    }else if (Double.parseDouble(direction) > 800){
        System.out.println(Thread.currentThread().getName() + " #
        sendToActuatorDirect(RudderLeftInstruction,3);
    }
}

if ((speed != null)  && (StatusLossOfCabinPressure.get() <= 0)){
```

Figure 35: FlightControlSystem file Run Method(2)



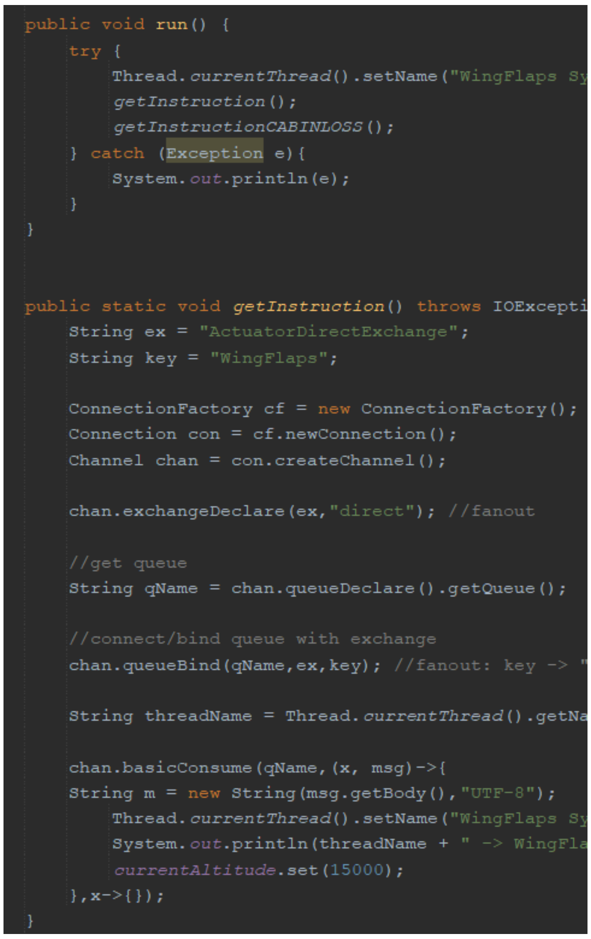
```
static void sendToActuatorDirect(String instruction, int exTyp
    String ex = "ActuatorDirectExchange";
    String key = null;

    //1.create connection
    ConnectionFactory cf = new ConnectionFactory();

    //2.use Factory to create a connection
    try(Connection con = cf.newConnection()){

        //3.create a channel using the connection
        Channel chan = con.createChannel();
        //engine // landing gear // oxygen masks
        switch(exType) {
            case 1: // to WingFlaps
                key = "WingFlaps";
                break;

            case 2: //CabinPressurization
                key = "CabinPressurization";
                break;

            case 3: //Rudder
                key = "Rudder";
                break;

            case 4: //Engine
                key = "Engine";
                break;
            default:
              // code block
        }
        //4.Declare a queue or exchange
        chan.exchangeDeclare(ex,"direct"); //type direct/fanou
        //5.send/publish message
        chan.basicPublish(ex,key,null,instruction.getBytes());
    }
```

Figure 36: FlightControlSystem file sendToActuatorDirect Method



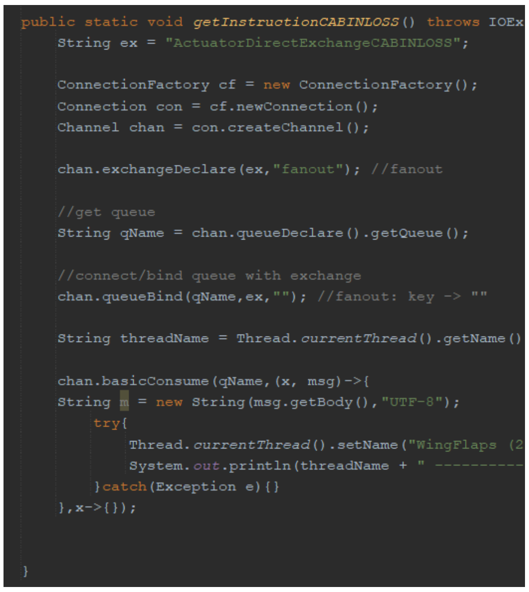
```
static void sendToActuatorFanout(String instruction)
    String ex = "ActuatorDirectExchangeCABINLOSS";
    //1.create connection
    ConnectionFactory cf = new ConnectionFactory();

    //2.use Factory to create a connection
    try(Connection con = cf.newConnection()){

        //3.create a channel using the connection
        Channel chan = con.createChannel();

        //4.Declare a queue or exchange
          chan.queueDeclare(qName, false, false, fal
        chan.exchangeDeclare(ex,"fanout"); //type di

        //5.send/publish message
        chan.basicPublish(ex,"",null,instruction.get

    }
}
```

Figure 37: FlightControlSystem file sendToActuatorFanout Method



```
public void run() {
    try {
        Thread.currentThread().setName("WingFlaps Sy
        getInstruction();
        getInstructionCABINLOSS();
    } catch (Exception e){
        System.out.println(e);
    }
}


public static void getInstruction() throws IOExcepti
    String ex = "ActuatorDirectExchange";
    String key = "WingFlaps";

    ConnectionFactory cf = new ConnectionFactory();
    Connection con = cf.newConnection();
    Channel chan = con.createChannel();

    chan.exchangeDeclare(ex,"direct"); //fanout

    //get queue
    String qName = chan.queueDeclare().getQueue();

    //connect/bind queue with exchange
    chan.queueBind(qName,ex,key); //fanout: key -> "

    String threadName = Thread.currentThread().getNa

    chan.basicConsume(qName,(x, msg)->{
    String m = new String(msg.getBody(),"UTF-8");
        Thread.currentThread().setName("WingFlaps Sy
        System.out.println(threadName + " -> WingFla
        currentAltitude.set(15000);
    },x->{});
}
```

Figure 38: WingFlaps file Run and getInstruction Method



```
public static void getInstructionCABINLOSS() throws IOEx
    String ex = "ActuatorDirectExchangeCABINLOSS";

    ConnectionFactory cf = new ConnectionFactory();
    Connection con = cf.newConnection();
    Channel chan = con.createChannel();

    chan.exchangeDeclare(ex,"fanout"); //fanout

    //get queue
    String qName = chan.queueDeclare().getQueue();

    //connect/bind queue with exchange
    chan.queueBind(qName,ex,""); //fanout: key -> ""

    String threadName = Thread.currentThread().getName()

    chan.basicConsume(qName,(x, msg)->{
    String m = new String(msg.getBody(),"UTF-8");
        try{
            Thread.currentThread().setName("WingFlaps (2
            System.out.println(threadName + " ---------
        }catch(Exception e){}
    },x->{});


}
```

Figure 39: WingFlaps file getInstructionCABINLOSS method (fanout ex)