



INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT087-3-3-RTS

REALTIME SYSTEMS

APU3F2211CS(DA)

HAND OUT DATE : 19th DECEMBER 2022

HAND IN DATE : 19th MARCH 2023

WEIGHTAGE : 50%

NAME : EDWARD LEONARDO

STUDENT ID : TP058284

ASSIGNMENT : INDIVIDUAL ASSIGNMENT

LECTURER : ASSOC. PROF. DR. IMRAN MEDI

Table of Contents

Abstract	3
1. Introduction.....	3
2. Background (Literature Review)	4
2.1 Real-Time System.....	4
2.2 Latency	5
2.3 Java in Real-Time System.....	5
2.4 Benchmarking Techniques.....	6
2.5 Profiling Technique.....	6
3. Simulation Design (Investigation Methodology).....	7
3.1 Simulation Overview.....	7
3.2 Implementation of Method.....	7
3.3 Simulation	8
3.4 Performance Analysis	13
4. Results and Discussion	13
4.1 Tweaks Made	13
4.1.1 the FCS and Actuators System Thread Management.....	13
4.1.2 Changing the number of cores for ScheduledThreadPool.....	14
4.2 Execution Time	14
4.3 Profiling.....	15
5. Conclusion	15
6. References.....	15

Investigating How Programming Design Effects Real-Time Performance –

A Simulation of A Flight Control System

Edward Leonardo

TP058284

Abstract

A Real-Time System (RTS) is a system that works under a determined time constraint. A RTS results is determined based on its response time and the outcome of the system. There are multiple factors that can have an impact on RTS performance, such as CPU core counts, Memory size allocated, the programming language chosen, and the system design. This report objective is to gain an understanding of how system design can have an impact on RTS performance using Java Programming Language. Benchmarking and Profiling will be used to determine the results of the RTS system, which is a simulation based on a Flight Control System.

1. Introduction

A real-time system (RTS) covers a system that works under a time constraint. This means that in RTS, the system responsiveness is also an important factor in determining if the system is functioning correctly or not, not only the logical correctness of the system. An RTS's response should be within a specified deadline. RTS is mainly used in time-sensitive industries, such as Air Traffic Control, networked multimedia systems, and many more (Juvva, 1998).

As currently web applications is on the rise, a lot of this application's core involves around Real-Time programming, due to its latency-sensitivity. Chatting, live streaming, gaming, and video calls are the

prime example of applications that are very sensitive to latency, as the amount of latency that the application have can massively impacts the performance or the user experience of these applications. Thus shows the importance of RTS implementation in these applications' design, to mitigate the latency faced by the users. By using RTS, it can minimize the amount of latency by optimization in hardware and software area (Rana, 2017).

In this report, RTS will be applied towards a Flight Control System Simulation using Java Programming Language. This report includes a detailed review into RTS and its relevant areas. It will also contain an investigation methodology by doing the Flight Control System Simulation, and

lastly to look on the simulation results and discussion regarding it.

2. Background (Literature Review)

Before discussing the simulation case, a Literature review will be conducted about RTS and its knowledge areas that are relevant to this investigation.

2.1 Real-Time System

Real-Time System (RTS) can be defined as hardware and software that need to operate under a certain time limitation, where response time is an important factor in determining if the system is a success (Bejawada, 2019). This means that responsiveness and latency amount is also considered during the creation of a Real-Time Systems.

There are two main types of RTS, which is differentiated by its timing constraints. The first type is Hard RTS, which means that the system cannot miss its deadline, where a missed deadline can cause disastrous consequences. This means that if the usefulness of the RTS results decreased, even become negative based on its tardiness. Tardiness means how late a RTS completes its task, compared to its deadline. The best example of this type is Air Traffic Controller system, Flight Control System, and Vehicle System (GeeksforGeeks, 2022a).

The other type is called Soft RTS, where the system can occasionally miss its deadline without causing any catastrophic consequences. The result of the system gradually decreased in tandem with the increase of its tardiness. The example of this type is video chat and live streaming, where missing a deadline can decrease the user experience, but not completely ruined it (GeeksforGeeks, 2022a).

RTS can provide many benefits. The main benefit of RTS is a better utilization of hardware and software usage that is running RTS, meaning that if implemented correctly, RTS can be very efficient. This also means that RTS programs size can be small compared to non-RTS programs. RTS is also more likely to be error-free due to its nature. For all of these upside, RTS main downside are its complexity, where creating a RTS can be a very complex task. Due to its complex design, RTS can be very costly to develop (javatpoint, n.d.).

RTS is made up of many characteristics. The first one is the system can accomplish all tasks before the deadlines. Another one is the systems also need to produce precise outcomes. Furthermore, another vital characteristic is to support concurrency, meaning that the system can handle running multiple tasks at the same time and completing them on time. The last characteristic is reliability, where a RTS

system must be able to function for a long period of time without any error occurred. In the case of an error, the system also must be able to recover quickly (GeeksforGeeks, 2022b).

2.2 Latency

Latency can be defined as the time needed between an action taken into the result of that action being produced. Latency is a very integral part of RTS, where a real-time system aims for having the lowest latency possible (Floridi, 2021). A RTS can be measured based on its predictability to finish tasks. This predictability is evaluated based on the system's latency and jitter. Where latency is the time needed between the action taken to the results produced, jitter is the variation of latency between each iteration of the system. Based on this, a very low latency RTS value can decrease if it has a high jitter. Thus, reducing the inconsistency between iteration is also important in designing a RTS (Intel, n.d.). According to Lee et al. (2007), There are multiple factors that can contribute to latency in RTS, such as Hardware delays, where the speed of CPU and memory can have an affect to the latency. Software delay also can affect the latency, based on the design and implementation of the software. Another factor is Scheduling, depending on how the tasks scheduling can have an impact on the RTS latency.

2.3 Java in Real-Time System

Java mainly used as an Object-Oriented Programming language mainly used to create embedded system. Java code is compiled independently by Java Virtual Machine (JVM). Java is an Object-Oriented programming (OOP) language, which means it uses objects and classes for its code. It also supports many OOP concepts (Sierra & Bates, 2005). Java is often used for developing RTS. The main consideration why Java is used is due to its automatic memory management using Garbage Collection. By using Garbage Collection, Java will free up unused memory, thus preventing memory-based issues. GC works by scanning the program's heap size, identifying any data that is no longer used by the program, and then releasing that memory (Patil, 2021). Another benefit of using Java is the capability of Multi-Threading, which supports simultaneous execution of multiple tasks in a program, more commonly known as concurrent programming (Vogel, 2023). Using concurrency, it allows better performance, responsiveness, and resource utilization due to its concept of running multiple tasks independently at the same time. Due to these advantages, Java is a suitable language for developing Real-Time Systems for its Garbage Collection system and Multi-Threading support.

2.4 Benchmarking Techniques

Benchmarking is an important part of Real-Time System, where its evaluate the performance of a RTS by measuring the metrics, such as response time. In RTS, there is a type of benchmark called microbenchmark, where it focuses on measuring the performance of a specific part of the RTS system. The purpose of microbenchmark is to measure the performance of an isolated area of the system. This allows developers to see the performance of that specific area, allowing them to discover any causes that can cause performance issue of the system, thus allowing the potential of optimization of the RTS (Poggi, 2019).

Java programming language support micro benchmarking though the JMH (Java Microbenchmark Harness) framework. JMH is one of the most popular microbenchmark tools available for Java (Baeldung, 2022). It allows multiple different types of benchmarking, with varieties of parameters and results recorded, such as throughput, average time, single shot time, and sample time. It also allows warm-up iteration to make sure that the JVM is already warmed-up before running the benchmark, eliminating the cold start scenario (Jenkov, 2015). Overall, JMH is a very powerful tool to do benchmark in Java Programming

Language. To enable JMH, the JMH dependencies need to be added to the Java Project by declaring the dependency into the pom.xml file of the project (Mkyong, 2018).

Another way of doing benchmarking is using manual method. By recording the elapsed time manually, the average time results can be done by getting the elapsed time of multiple iterations of the system. Even though its more tedious, manual benchmarking can be an alternative for JMH.

2.5 Profiling Technique

Profiling can be defined as process to measure the performance of the system and identifying the resource used during the system's execution. Profiling allows developers to see how the hardware is operating the system during execution. The benefit of profiling is to gain understanding of internal performance, showing how the system is performing and where improvements can be made to the system to allow better hardware utilization. (Javatpoint, n.d.). Profiling involves monitoring system resources such as memory and CPU usage. In Java Programming, profiler also measure Garbage Collection execution.

In this simulation case, NetBeans Profiler will be the profiler of choice to do profiling

of the simulation. NetBeans Profiler can show the telemetry results of the simulation, such as CPU and GC time, Heap size usage, Garbage Collections, and Threads and Classes.

3. Simulation Design (Investigation Methodology)

3.1 Simulation Overview

The purpose of this simulation is to assess the performance of Real-Time systems in Java Programming Language. In this case, the simulation will be in the form of Flight Control System. In this simulation, there will be 4 sensor systems, 5 actuator systems, and a Flight Control System main module. The Flight Control System will do 3 loops of normal simulation, with a total of 12 data collection from the sensors to the actuators adjustments, with 1 loop consisting of 4 data collection and actuators adjustments. These loops will cover the normal simulation. After 3 loops of normal simulation, an emergency case will be conducted as well, with a loss of cabin pressure scenario happening, causing an emergency landing to happen as well. This emergency scenario will always be done after the normal simulation ends.

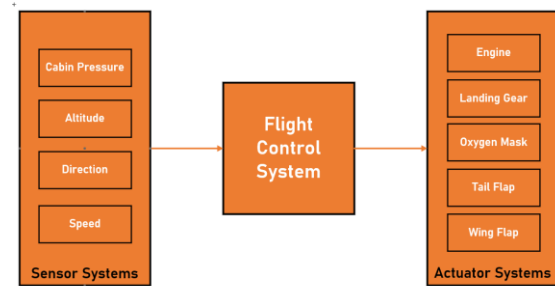


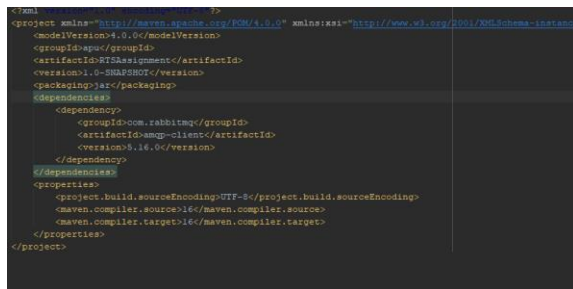
Figure 1 - Flowchart of the Simulation

The above figure shows the model of the simulation. Each sensor system (Producer) will generate a data that will be passed to the Flight Control System, which in turn will pass the data to the appropriate actuator system (Consumer). In the normal simulation, the Cabin Pressure sensor will send the data to the oxygen mask actuator, the Altitude sensor will pass its data to the Wing Flap actuator, the Direction sensor will pass its data to the Tail Flap actuator, and the Speed sensor will pass its data to the Engine actuator. During the emergency simulation, then the Altitude sensor will also take control of the Landing Gear and the Engine Actuators.

3.2 Implementation of Method

The simulation above will be done in the Apache NetBeans IDE version 12.5. The Thread management will be done using `ScheduledExecutorService`, specially used for the sensor systems. The synchronization aid will be done using `CountDownLatch`. The data will be passed from each systems using `RabbitMQ`. The benchmark will be done using manual benchmarking methods

for the timing, and NetBeans Profiler for the profiling. Any Dependencies will be added to the pom.xml file, with only RabbitMQ dependency being added to the file.



```

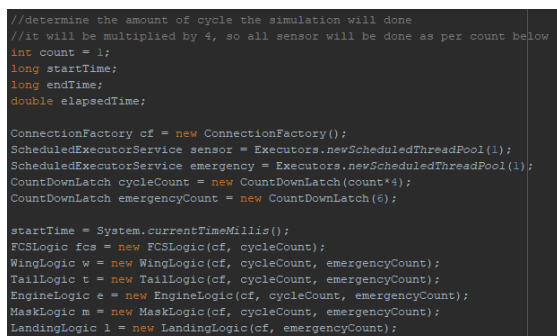
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>apc</groupId>
    <artifactId>RTSAssignment</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <dependencies>
        <dependency>
            <groupId>com.rabbitmq</groupId>
            <artifactId>amqp-client</artifactId>
            <version>5.16.0</version>
        </dependency>
    </dependencies>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>16</maven.compiler.source>
        <maven.compiler.target>16</maven.compiler.target>
    </properties>
</project>

```

Figure 2 - pom.xml file

3.3 Simulation

The simulation starting point will be in the FlightControlSystem file. It includes all simulation execution, thus executing this file will in turn execute all other java files. First in the file will be declaring all variables and classes that will be needed for the project.



```

//determine the amount of cycle the simulation will done
//it will be multiplied by 4, so all sensor will be done as per count below
int count = 1;
long startTime;
long endTime;
double elapsedTime;

ConnectionFactory cf = new ConnectionFactory();
ScheduledExecutorService sensor = Executors.newScheduledThreadPool(1);
ScheduledExecutorService emergency = Executors.newScheduledThreadPool(1);
CountDownLatch cycleCount = new CountDownLatch(count*4);
CountDownLatch emergencyCount = new CountDownLatch(6);

startTime = System.currentTimeMillis();
FCSLogic fcs = new FCSLogic(cf, cycleCount);
WingLogic w = new WingLogic(cf, cycleCount, emergencyCount);
TailLogic t = new TailLogic(cf, cycleCount, emergencyCount);
EngineLogic e = new EngineLogic(cf, cycleCount, emergencyCount);
MaskLogic m = new MaskLogic(cf, cycleCount, emergencyCount);
LandingLogic l = new LandingLogic(cf, emergencyCount);

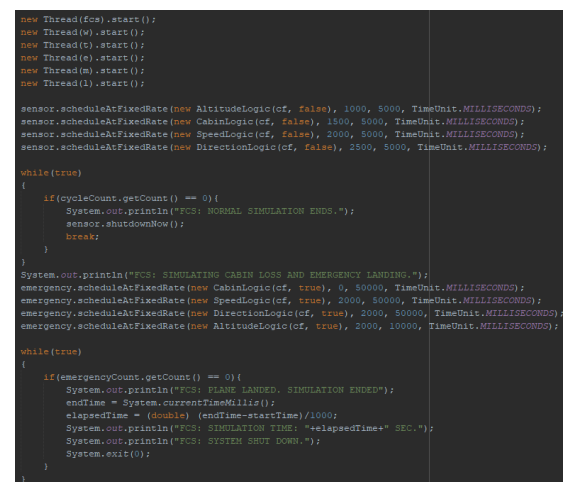
```

Figure 3 - Declaring Variables and Classes

The variables declared are count, which saves the number of loops for the simulation, and preparing the variables for elapsed time benchmarking. The classes declared are ConnectionFactory for

RabbitMQ, 2 ScheduledExecutorService for both simulation scenarios, and 2 CountDownLatch for both simulation scenarios. The emergencyCount is set to a fixed 6, while the cycleCount is count multiplied by 4.

Next is starting all the Threads for the all the functions.



```

new Thread(fcs).start();
new Thread(w).start();
new Thread(t).start();
new Thread(e).start();
new Thread(m).start();
new Thread(l).start();

sensor.scheduleAtFixedRate(new AltitudeLogic(cf, false), 1000, 5000, TimeUnit.MILLISECONDS);
sensor.scheduleAtFixedRate(new CabinLogic(cf, false), 1500, 5000, TimeUnit.MILLISECONDS);
sensor.scheduleAtFixedRate(new SpeedLogic(cf, false), 2000, 5000, TimeUnit.MILLISECONDS);
sensor.scheduleAtFixedRate(new DirectionLogic(cf, false), 2500, 5000, TimeUnit.MILLISECONDS);

while(true)
{
    if(cycleCount.getCount() == 0){
        System.out.println("FCS: NORMAL SIMULATION ENDS.");
        sensor.shutdownNow();
        break;
    }
    System.out.println("FCS: SIMULATING CABIN LOSS AND EMERGENCY LANDING.");
    emergency.scheduleAtFixedRate(new CabinLogic(cf, true), 0, 50000, TimeUnit.MILLISECONDS);
    emergency.scheduleAtFixedRate(new SpeedLogic(cf, true), 2000, 50000, TimeUnit.MILLISECONDS);
    emergency.scheduleAtFixedRate(new DirectionLogic(cf, true), 2000, 50000, TimeUnit.MILLISECONDS);
    emergency.scheduleAtFixedRate(new AltitudeLogic(cf, true), 2000, 10000, TimeUnit.MILLISECONDS);

    while(true)
    {
        if(emergencyCount.getCount() == 0){
            System.out.println("FCS: PLANE LANDED. SIMULATION ENDED.");
            endTime = System.currentTimeMillis();
            elapsedTime = (double) (endTime - startTime) / 1000;
            System.out.println("FCS: SIMULATION TIME: "+elapsedTime+" SEC.");
            System.out.println("FCS: SYSTEM SHUT DOWN.");
            System.exit(0);
        }
    }
}

```

Figure 4 - Process Execution Flow

The Flight Control System and the Actuator Systems will be only Threads running without any management, while the Sensor Systems will be managed by ScheduledExecutorService. For the normal simulation, it will run each sensors every 5 seconds, with 0.5 seconds interval between each sensor so the data can be shown in a more manageable way. If the cycleCount CountDownLatch is already done, then the simulation will start with the emergencyCount CountDownLatch, where the sensors will be managed by another form of ScheduledExecutorService. After

the emergencyCount is over, then the simulation will be terminated, and the elapsed time will be calculated.

Now for each system explanation. The Sensors systems are mostly identical in functionality, with each sensor having their own logic, where they generate a randomized data, and then the data will be sent towards the Flight Control System (FCS). using RabbitMQ. The difference between each sensor will be the output they print out, and the data that they sent to the FCS. The output of each sensor will then be forwarded to the appropriate actuators, where it will be interpreted. The sensors' code will all be shown below:

```
class AltitudeLogic implements Runnable {
    ConnectionFactory cf;
    boolean emergencyState;
    int emergencyCycle = 0;

    public AltitudeLogic(ConnectionFactory cf, boolean emergencyState) {
        this.cf = cf;
        this.emergencyState = emergencyState;
    }

    Random rand = new Random();
    String senderKey = "sensorToControl";

    @Override
    public void run() {
        //generate data
        String data = generateData();
        if(emergencyState){
            data = generateEmergencyData();
        }
        //send data
        sendData(senderKey, data);
        System.out.println("ALTITUDE SENSOR: DATA TRANSMITTED TO FCS. CODE: "+data);
    }
}
```

Figure 5 - Altitude Sensor Logic Part 1

```
public String generateData(){
    String data;

    int no = rand.nextInt(4);

    switch (no) {
        case 0:
            data = "ALT-tooHigh";
            break;
        case 1:
            data = "ALT-slightlyHigh";
            break;
        case 2:
            data = "ALT-tooLow";
            break;
        case 3:
            data = "ALT-slightlyLow";
            break;
        default:
            data = "ALT-ok";
            break;
    }
    return data;
}

public String generateEmergencyData() {
    String data;

    switch (emergencyCycle) {
        case 0:
            data = "ALT-emergency";
            emergencyCycle++;
            break;
        case 1:
            data = "LAND-open";
            emergencyCycle++;
            break;
        default:
            data = "SPD-land";
            break;
    }
    return data;
}
```

Figure 6 - Altitude Sensor Logic Part 2

```
public void sendData(String key, String data){
    try(Connection con = cf.newConnection()){
        Channel ch = con.createChannel();
        ch.queueDeclare(key, false, false, false, null);
        ch.basicPublish("", key, false, null, data.getBytes());
    }
    catch (Exception e){
        System.out.println("ALTITUDE SENSOR: ERROR, NO CONNECTION TO FCS.");
    }
}
```

Figure 7 - Altitude Sensor Logic Part 3

```
class CabinLogic implements Runnable {
    ConnectionFactory cf;
    boolean emergencyState;

    public CabinLogic(ConnectionFactory cf, boolean emergencyState) {
        this.cf = cf;
        this.emergencyState = emergencyState;
    }

    Random rand = new Random();
    String senderKey = "sensorToControl";

    @Override
    public void run() {
        //generate data
        String data = "CAB-ok";
        if(emergencyState){
            data = "CAB-breath";
        }
        //send data
        sendData(senderKey, data);
        System.out.println("CABIN PRESSURE SENSOR: DATA TRANSMITTED TO FCS. CODE: "+data);
    }

    public void sendData(String key, String data){
        try(Connection con = cf.newConnection()){
            Channel ch = con.createChannel();
            ch.queueDeclare(key, false, false, false, null);
            ch.basicPublish("", key, false, null, data.getBytes());
        }
        catch (Exception e){
            System.out.println("CABIN PRESSURE SENSOR: ERROR, NO CONNECTION TO FCS.");
        }
    }
}
```

Figure 8 - Cabin Pressure Sensor Logic

```

class DirectionLogic implements Runnable {
    ConnectionFactory cf;
    boolean emergencyState;

    public DirectionLogic(ConnectionFactory cf, boolean emergencyState) {
        this.cf = cf;
        this.emergencyState = emergencyState;
    }

    Random rand = new Random();
    String senderKey = "sensorToControl";

    @Override
    public void run() {
        //generate data
        String data = generateData();
        if(emergencyState){
            data = "DRT-emergency";
        }
        //send data
        sendData(senderKey, data);
        System.out.println("DIRECTION SENSOR: DATA TRANSMITTED TO FCS. CODE: "+data);
    }
}

```

Figure 9 - Direction Sensor Logic Part 1

```

public String generateData(){
    String data;

    int no = rand.nextInt(2);

    switch (no) {
        case 0:
            data = "DRT-tooLeft";
            break;
        case 1:
            data = "DRT-tooRight";
            break;
        default:
            data = "DRT-ok";
            break;
    }
    return data;
}

public void sendData(String key, String data){
    try(Connection con = cf.newConnection()){
        Channel ch = con.createChannel();
        ch.queueDeclare(key, false, false, false, null);
        ch.basicPublish("", key, false, null, data.getBytes());
    }
    catch (Exception e){
        System.out.println("DIRECTION SENSOR: ERROR, NO CONNECTION TO FCS.");
    }
}

```

Figure 10 - Direction Sensor Logic Part 2

```

class SpeedLogic implements Runnable {
    ConnectionFactory cf;
    boolean emergencyState;

    public SpeedLogic(ConnectionFactory cf, boolean emergencyState) {
        this.cf = cf;
        this.emergencyState = emergencyState;
    }

    Random rand = new Random();
    String senderKey = "sensorToControl";

    @Override
    public void run() {
        //generate data
        String data = generateData();
        if(emergencyState){
            data = "SPD-emergency";
        }
        //send data
        sendData(senderKey, data);
        System.out.println("SPEED SENSOR: DATA TRANSMITTED TO FCS. CODE: "+data);
    }
}

```

Figure 11 - Speed Sensor Logic Part 1

```

public String generateData(){
    String data;

    int no = rand.nextInt(2);

    switch (no) {
        case 0:
            data = "SPD-tooFast";
            break;
        case 1:
            data = "SPD-tooSlow";
            break;
        default:
            data = "SPD-ok";
            break;
    }
    return data;
}

public void sendData(String key, String data){
    try(Connection con = cf.newConnection()){
        Channel ch = con.createChannel();
        ch.queueDeclare(key, false, false, false, null);
        ch.basicPublish("", key, false, null, data.getBytes());
    }
    catch (Exception e){
        System.out.println("SPEED SENSOR: ERROR, NO CONNECTION TO FCS.");
    }
}

```

Figure 12 - Speed Sensor Logic Part 2

Next is the FCS itself. The logic in the FCS is the FCS will receive the data from the sensors. The data then will be decoded, as each data transferred also contains the Channel key for their respective linked actuators. The FCS will decode the data, and then will send the data to the appropriate actuators based on the Channel Key from the sensors' data.

```

class FCSLogic implements Runnable {
    ConnectionFactory cf;
    CountdownLatch cycleCount;

    public FCSLogic(ConnectionFactory cf, CountdownLatch cycleCount) {
        this.cf = cf;
        this.cycleCount = cycleCount;
    }

    String receiverKey = "sensorToControl";
    String senderKey = "controlToMachine";

    @Override
    public void run() {
        System.out.println("FCS: SYSTEM INITIATED.");
        receiveData();
        try{
            cycleCount.await();
        }
        catch (Exception e) {}
    }
}

```

Figure 13 - FCS Logic Part 1

```

public void receiveData(){
    try{
        Connection con = cf.newConnection();
        Channel ch = con.createChannel();
        ch.queueDeclare(receiverKey, false, false, false, null);
        try{
            ch.basicConsume(receiverKey, true, (x, msg)->{
                String message = new String(msg.getBody(),"UTF-8");
                String[] decoded = dataDecoding(message);
                try{
                    Thread.sleep(500);
                }
                catch (Exception e) {}
                sendData(decoded[0], decoded[1]);
            }, x->{});
        }
        catch (Exception e){
            System.out.println("FCS: ERROR, CANNOT RECEIVE DATA FROM SENSOR.");
        }
    }
    catch (Exception e){
        System.out.println("FCS: ERROR, NO CONNECTION TO SENSOR.");
    }
}

public String[] dataDecoding(String original){
    String[] decoded = original.split("-");
    return decoded;
}

```

Figure 14 - FCS Logic Part 2

```

public void receiveData(){
    try{
        Connection con = cf.newConnection();
        Channel ch = con.createChannel();
        ch.exchangeDeclare(receiverKey, "topic");

        String qName = ch.queueDeclare().getQueue();
        ch.queueBind(qName, receiverKey, machineKey);
        try {
            ch.basicConsume(qName, true, (x, msg)->{
                String data = new String(msg.getBody(),"UTF-8");
                System.out.println("TAIL: DATA RECEIVED FROM FCS: "+ data);
                try{
                    Thread.sleep(500);
                }
                catch (Exception e) {}
                adjustment(data);
                cycleCount.countDown();
            }, x->{});
        }
        catch (Exception e){
            System.out.println("TAIL: ERROR, CANNOT RECEIVE DATA FROM FCS.");
        }
    }
    catch (Exception e) {
        System.out.println("TAIL: ERROR, NO CONNECTION TO FCS.");
    }
}

```

Figure 17 - Tail Flap Actuator Logic Part 2

```

public void sendData(String key, String data){
    try{Connection con = cf.newConnection();
    Channel chan = con.createChannel();
    chan.exchangeDeclare(senderKey, "topic");
    chan.basicPublish(senderKey, key, false, null, data.getBytes());
    if(key.equals("ALT"))
        System.out.println("FCS: DATA TRANSMITTED TO THE WING. DATA: "+data);
    else if(key.equals("CAB"))
        System.out.println("FCS: CABIN PRESSURE REPORTED: "+data);
    else if (key.equals("SPD"))
        System.out.println("FCS: DATA TRANSMITTED TO THE ENGINE. DATA "+data);
    else if (key.equals("DST"))
        System.out.println("FCS: DATA TRANSMITTED TO THE TAIL. DATA "+data);
    else if (key.equals("LAND"))
        System.out.println("FCS: DATA TRANSMITTED TO THE LANDING GEAR. DATA "+data);
    }
    catch (Exception e){}
}

```

Figure 15 - FCS Logic Part 3

```

public void adjustment(String data){
    if(data.equals("tooRight"))
        System.out.println("TAIL: ADJUSTING TO THE LEFT.");
    else if(data.equals("tooLeft"))
        System.out.println("TAIL: ADJUSTING TO THE RIGHT.");
    else if (data.equals("ok"))
        System.out.println("TAIL: NO ADJUSTMENT NEEDED.");
    else if (data.equals("emergency")){
        System.out.println("TAIL: ADJUSTING TO THE DIRECTION OF A RUNWAY.");
        emergencyCount.countDown();
    }
}

```

Figure 18 - Tail Flap Actuator Logic Part 3

Finally, the Actuators system will receive the data that has been sent by the FCS. The data then will be used to adjust each actuator. The adjustment made will be based on the data received by from the sensors and FCS. Only Landing Gear Actuator have a difference, where it is only used during the emergency case.

```

class TailLogic implements Runnable {
    ConnectionFactory cf;
    CountDownLatch cycleCount;
    CountDownLatch emergencyCount;

    public TailLogic(ConnectionFactory cf, CountDownLatch cycleCount, CountDownLatch emergencyCount) {
        this.cf = cf;
        this.cycleCount = cycleCount;
        this.emergencyCount = emergencyCount;
    }

    String receiverKey = "controlToMachine";
    String machineKey = "DST";

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

```

Figure 16 - Tail Flap Actuator Logic Part 1

```

class WingLogic implements Runnable {
    ConnectionFactory cf;
    CountDownLatch cycleCount;
    CountDownLatch emergencyCount;

    public WingLogic(ConnectionFactory cf, CountDownLatch cycleCount, CountDownLatch emergencyCount) {
        this.cf = cf;
        this.cycleCount = cycleCount;
        this.emergencyCount = emergencyCount;
    }

    String receiverKey = "controlToMachine";
    String machineKey = "ALT";

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

```

Figure 19 - Wing Flap Actuator Logic Part 1

```

public void receiveData(){
    try{
        Connection con = cf.newConnection();
        Channel ch = con.createChannel();
        ch.exchangeDeclare(receiverKey, "topic");

        String qName = ch.queueDeclare().getQueue();
        ch.queueBind(qName, receiverKey, machineKey);
        try {
            ch.basicConsume(qName, true, (x, msg)->{
                String data = new String(msg.getBody(),"UTF-8");
                System.out.println("WING: DATA RECEIVED FROM FCS: "+ data);
                try{
                    Thread.sleep(500);
                }
                catch (Exception e) {}
                adjustment(data);
                cycleCount.countDown();
            }, x->{});
        }
        catch (Exception e){
            System.out.println("WING: ERROR, CANNOT RECEIVE DATA FROM FCS.");
        }
    }
    catch (Exception e) {
        System.out.println("WING: ERROR, NO CONNECTION TO FCS.");
    }
}

```

Figure 20 - Wing Flap Actuator Logic Part 2

```

public void adjustment(String data){
    if(data.equals("tooHigh"))
        System.out.println(("WING: RAISING FLAP."));
    else if(data.equals("slightlyHigh"))
        System.out.println(("WING: RAISING FLAP SLIGHTLY."));
    else if (data.equals("tooLow"))
        System.out.println(("WING: LOWERING FLAP."));
    else if (data.equals("slightlyLow"))
        System.out.println(("WING: LOWERING FLAP SLIGHTLY."));
    else if (data.equals("emergency")){
        System.out.println(("WING: RAISING FLAP FOR EMERGENCY."));
        emergencyCount.countDown();
    }
    else if (data.equals("ok"))
        System.out.println(("WING: NO ADJUSTMENT NEEDED."));
}

```

Figure 21 - Wing Flap Actuator Logic Part 3

```

class LandingLogic implements Runnable {
    ConnectionFactory cf;
    CountdownLatch emergencyCount;

    public LandingLogic(ConnectionFactory cf, CountdownLatch emergencyCount) {
        this.cf = cf;
        this.emergencyCount = emergencyCount;
    }

    String receiverKey = "controlToMachine";
    String machineKey = "LAND";

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

```

Figure 22 - Landing Gear Actuator Logic Part 1

```

public void receiveData(){
    try{
        Connection con = cf.newConnection();
        Channel ch = con.createChannel();

        ch.exchangeDeclare(receiverKey, "topic");

        String qName = ch.queueDeclare().getQueue();
        ch.queueBind(qName, receiverKey, machineKey);
        try {
            ch.basicConsume(qName, true, (x, msg)->{
                String data = new String(msg.getBody(),"UTF-8");
                System.out.println(("LANDING GEAR: DATA RECEIVED FROM FCS: "+ data));
                try{
                    Thread.sleep(500);
                }
                catch (Exception e) {}
                adjustment(data);
            }, x->{});
        }
        catch (Exception e){
            System.out.println(("LANDING GEAR: ERROR, CANNOT RECEIVE DATA FROM FCS."));
        }
    }
    catch (Exception e) {
        System.out.println(("LANDING GEAR: ERROR, NO CONNECTION TO FCS."));
    }
}

public void adjustment(String data){
    if(data.equals("open")){
        System.out.println(("LANDING GEAR: OPENING LANDING GEAR."));
        try {
            Thread.sleep(500);
        } catch (InterruptedException ex) {}
        System.out.println(("LANDING GEAR: LANDING GEAR OPENED."));
        emergencyCount.countDown();
    }
}

```

Figure 23 - Landing Gear Actuator Logic Part 2

```

class MaskLogic implements Runnable {
    ConnectionFactory cf;
    CountdownLatch cycleCount;
    CountdownLatch emergencyCount;

    public MaskLogic(ConnectionFactory cf, CountdownLatch cycleCount, CountdownLatch emergencyCount) {
        this.cf = cf;
        this.cycleCount = cycleCount;
        this.emergencyCount = emergencyCount;
    }

    String receiverKey = "controlToMachine";
    String machineKey = "CAB";

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

```

Figure 24 - Oxygen Mask Actuator Logic Part 1

```

public void receiveData(){
    try{
        Connection con = cf.newConnection();
        Channel ch = con.createChannel();

        ch.exchangeDeclare(receiverKey, "topic");

        String qName = ch.queueDeclare().getQueue();
        ch.queueBind(qName, receiverKey, machineKey);
        try {
            ch.basicConsume(qName, true, (x, msg)->{
                String data = new String(msg.getBody(),"UTF-8");
                System.out.println(("MASK: DATA RECEIVED FROM FCS: "+ data));
                try{
                    Thread.sleep(500);
                }
                catch (Exception e) {}
                adjustment(data);
                cycleCount.countDown();
            }, x->{});
        }
        catch (Exception e){
            System.out.println(("MASK: ERROR, CANNOT RECEIVE DATA FROM FCS."));
        }
    }
    catch (Exception e) {
        System.out.println(("MASK: ERROR, NO CONNECTION TO FCS."));
    }
}

public void adjustment(String data){
    if(data.equals("breach")){
        System.out.println(("MASK: CABIN PRESSURE LOSS. OXYGEN MASK DROPPING."));
        emergencyCount.countDown();
    }
    else if (data.equals("ok"))
        System.out.println(("MASK: NO ADJUSTMENT NEEDED."));
}

```

Figure 25 - Oxygen Mask Actuator Logic Part 2

```

class EngineLogic implements Runnable {
    ConnectionFactory cf;
    CountdownLatch cycleCount;
    CountdownLatch emergencyCount;

    public EngineLogic(ConnectionFactory cf, CountdownLatch cycleCount, CountdownLatch emergencyCount) {
        this.cf = cf;
        this.cycleCount = cycleCount;
        this.emergencyCount = emergencyCount;
    }

    String receiverKey = "controlToMachine";
    String machineKey = "SPD";

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

```

Figure 26 - Engine Actuator Logic Part 1

```

public void receiveData(){
    try{
        Connection con = cf.newConnection();
        Channel ch = con.createChannel();

        ch.exchangeDeclare(receiverKey, "topic");

        String qName = ch.queueDeclare().getQueue();
        ch.queueBind(qName, receiverKey, machineKey);
        try {
            ch.basicConsume(qName, true, (x, msg)->{
                String data = new String(msg.getBody(),"UTF-8");
                System.out.println(("ENGINE: DATA RECEIVED FROM FCS: "+ data));
                try{
                    Thread.sleep(500);
                }
                catch (Exception e) {}
                adjustment(data);
                cycleCount.countDown();
            }, x->{});
        }
        catch (Exception e){
            System.out.println(("ENGINE: ERROR, CANNOT RECEIVE DATA FROM FCS."));
        }
    }
    catch (Exception e) {
        System.out.println(("ENGINE: ERROR, NO CONNECTION TO FCS."));
    }
}

```

Figure 27 - Engine Actuator Logic Part 2

```

public void adjustment(String data){
    if(data.equals("tooFast"))
        System.out.println(("ENGINE: DECREASING ENGINE POWER."));
    else if(data.equals("tooSlow"))
        System.out.println(("ENGINE: INCREASING ENGINE POWER."));
    else if (data.equals("ok"))
        System.out.println(("ENGINE: NO ADJUSTMENT NEEDED."));
    else if (data.equals("land")){
        emergencyCount.countDown();
        System.out.println(("ENGINE: TURNING OFF ENGINE."));
    }
    else if (data.equals("emergency")){
        System.out.println(("ENGINE: ADJUSTING ENGINE FOR EMERGENCY LANDING"));
        emergencyCount.countDown();
    }
}

```

Figure 28 - Engine Actuator Logic Part 3

3.4 Performance Analysis

For the benchmark, since the manual benchmarking method will be used, the elapsed time will be calculated by subtracting the start time, that will be recorded when the simulation starts, with the end time, that will be recorded when the emergency simulation ends. From there, the developer can calculate the average time for the whole simulation. The simulation will be done for 10 times (iterations) to get the appropriate data, with each iteration will be 3 data cycle loops, meaning that each sensor will be producing 3 data before the simulation will ends.

```
startTime = System.currentTimeMillis();
```

Figure 29 - Start Time during the start of the system

```
endTime = System.currentTimeMillis();
elapsedTime = (double) (endTime - startTime) / 1000;
System.out.println("FCS: SIMULATION TIME: "+elapsedTime+" SEC.");
```

Figure 30 - End Time, Elapsed Time calculation and showing the result.

```
//determine the amount of cycle the simulation will done
//it will be multiplied by 4, so all sensor will be done as per count below
int count = 3;
```

Figure 31 - Amount of Cycle set

For profiling, the NetBeans profiler will be used to monitor the resource usage of the simulation. The profiler will record CPU and Garbage Collection time. It will also record the Memory and heap usage, the Thread usage, and the Garbage collection Intervals. From the results of the profiler, it will show the RTS effects on the hardware.

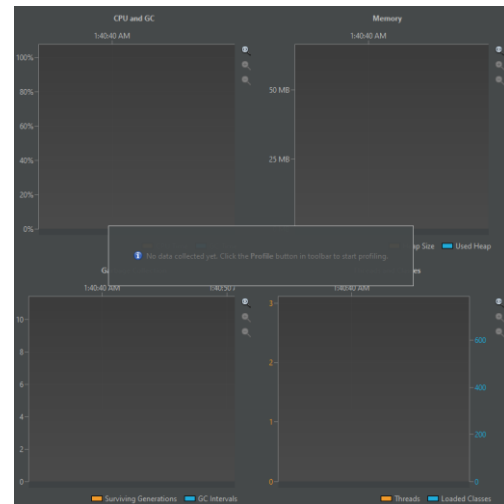


Figure 32 - NetBeans Profiler GUI

4. Results and Discussion

4.1 Tweaks Made

In the code for the simulation above, some tweaks can be made with the possibility of changing the results of the benchmark and profile, in turn for better or worse performance compared to the original code. There will be 2 tweaks that will be made towards the original code, which will be shown in their respective section.

4.1.1 the FCS and Actuators System Thread Management

In the original code, the FCS and Actuators System threads are activated independently without any use of a Thread Management. As the first tweak, the changes will be made in this specific area, where FCS and actuators system threads will be managed by `ExecutorService`, using the `CachedThreadPool`. The purpose of this tweak is by using a thread management, the

execution time or the resource usage can be improved.

```
startTime = System.currentTimeMillis();
FCSLogic fcs = new FCSLogic(cf, cycleCount);
WingLogic w = new WingLogic(cf, cycleCount, emergencyCount);
TailLogic t = new TailLogic(cf, cycleCount, emergencyCount);
EngineLogic e = new EngineLogic(cf, cycleCount, emergencyCount);
MaskLogic m = new MaskLogic(cf, cycleCount, emergencyCount);
LandingLogic l = new LandingLogic(cf, emergencyCount);

new Thread(fcs).start();
new Thread(w).start();
new Thread(t).start();
new Thread(e).start();
new Thread(m).start();
new Thread(l).start();
```

Figure 33 - Original Code

```
ExecutorService system = Executors.newCachedThreadPool();

startTime = System.currentTimeMillis();
system.submit(new FCSLogic(cf, cycleCount));
system.submit(new WingLogic(cf, cycleCount, emergencyCount));
system.submit(new TailLogic(cf, cycleCount, emergencyCount));
system.submit(new EngineLogic(cf, cycleCount, emergencyCount));
system.submit(new MaskLogic(cf, cycleCount, emergencyCount));
system.submit(new LandingLogic(cf, emergencyCount));
```

Figure 34 - Tweak 1

4.1.2 Changing the number of cores for ScheduledThreadPool

In the original code, the ScheduledThreadPool used for controlling the Sensor Systems is only set to 1. In this tweak, the cores used for the ScheduledThreadPool will be set to follow the amount of cores available in the test system CPU. With this change, it is hoped that the simulation will have better CPU time usage and better execution time result.

```
ConnectionFactory cf = new ConnectionFactory();
ScheduledExecutorService sensor = Executors.newScheduledThreadPool(1);
ScheduledExecutorService emergency = Executors.newScheduledThreadPool(1);
CountDownLatch cycleCount = new CountDownLatch(count*4);
CountDownLatch emergencyCount = new CountDownLatch(e);
```

Figure 35 - Original Code

```
int core = Runtime.getRuntime().availableProcessors();

ConnectionFactory cf = new ConnectionFactory();
ScheduledExecutorService sensor = Executors.newScheduledThreadPool(core);
ScheduledExecutorService emergency = Executors.newScheduledThreadPool(core);
CountDownLatch cycleCount = new CountDownLatch(count*4);
CountDownLatch emergencyCount = new CountDownLatch(e);
```

Figure 36 - Tweak 2

4.2 Execution Time

For benchmarks, the average time will be measured. As previously mentioned, manual benchmarking will be used to calculate the average execution time of the simulation based on 10 iterations. The result for each iteration will be recorded in the table below. The time will be recorded in second.

Iteration	Original	Tweak 1	Tweak 2
1	36.592	36.59	36.597
2	36.57	36.597	36.582
3	36.603	36.588	36.586
4	36.624	36.594	36.591
5	36.599	36.57	36.576
6	36.606	36.588	36.585
7	36.592	36.588	36.595
8	36.586	36.584	36.599
9	36.603	36.574	36.59
10	36.589	36.602	36.59
Average	36.59	36.58	36.58
Min	36.57	36.57	36.576
Max	36.624	36.602	36.599

Table 1 – Execution Time Results

Based on the results above, the tweaks only cause a very small amount of change in execution time, which can be considered as jitter between each iteration. This can be due to the minimal changes being made towards the original code, thus there is no noticeable differences between each version.

4.3 Profiling

Using the NetBeans Profilers, Profiling will be done towards the simulation project

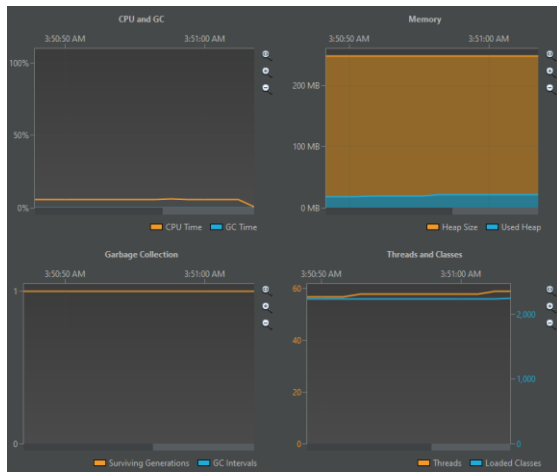


Figure 37 - Profiling Result

The above figure shows the result of profiling into the simulation systems. The maximum CPU time is 7.1%, with 0.1% GC time. The maximum used heap is 24.45 MB. The Garbage Collection has only 1 surviving generations, thus no Garbage Collection is conducted. The original simulation has 59 threads and 2259 loaded classes in total.

5. Conclusion

For Conclusion, the study is a investigation towards how programming design can have an impact on Real-Time system (RTS) performance, using the Flight Control System simulation. The study already covered domain research regarding relevant topics in according to the study's title. The study also managed to conduct a simulation successfully, with tweaks applied to the

original simulation. The simulation is then benchmarked and profiled to see if the tweaks have a noticeable impact on the simulation. Based on the results gathered, no improvement or loss happen towards the RTS's performance. This can be due to the minimal number of tweaks being applied, or the tweaks applied did not produce any noticeable impact towards the performance.

6. References

Baeldung. (2022, August 26). *Microbenchmarking with Java*. Retrieved March 15, 2023, from <https://www.baeldung.com/java-microbenchmark-harness>

Baeldung. (2023, March 17). *A Guide to Java Profilers*. Retrieved March 17, 2023, from <https://www.baeldung.com/java-profilers>

Bejawada, S. (2019). An Analysis to Identify the Factors that Impact the Performance of Real-Time Software Systems. *Faculty of Computing Blekinge Institute of Technology*. <https://www.diva-portal.org/smash/get/diva2:1422467/FULLTEXT02>

Floridi, L. (2021). Digital Time: Latency, Real-time, and the Onlife Experience of Everyday Time. *Philosophy & Technology*, 34(3), 407–412. <https://doi.org/10.1007/s13347-021-00472-5>

GeeksforGeeks. (2022a, January 12). *Real Time Systems*. Retrieved March 15, 2023, from <https://www.geeksforgeeks.org/real-time-systems/>

GeeksforGeeks. (2022b, December 28). *Characteristics of Real time Systems*. Retrieved March 15, 2023, from

<https://www.geeksforgeeks.org/characteristics-of-real-time-systems/>

Gillis, A. S. (2022, February 14). *real-time operating system (RTOS)*. Data Center. Retrieved March 15, 2023, from <https://www.techtarget.com/searchdatacenter/definition/real-time-operating-system>

Intel. (n.d.). *Real-Time Systems Overview and Examples*. Retrieved March 15, 2023, from <https://www.intel.com/content/www/us/en/robotics/real-time-systems.html>

Javatpoint. (n.d.). *Java Profilers*. Retrieved March 17, 2023, from <https://www.javatpoint.com/java-profilers>

javatpoint. (n.d.). *Real-Time operating system*. Javatpoint. Retrieved March 15, 2023, from <https://www.javatpoint.com/real-time-operating-system>

Jenkov, J. (2015, September 16). *JMH - Java Microbenchmark Harness*. Jenkov. Retrieved March 15, 2023, from <https://jenkov.com/tutorials/java-performance/jmh.html>

Juvva, K. (1998). *Real-Time Systems*. Carnegie Mellon University. Retrieved March 15, 2023, from https://users.ece.cmu.edu/~koopman/des_s99/real_time/

Lee, I., Leung, J. Y., & Son, S. H. (2007). *Handbook of Real-Time and Embedded Systems*. CRC Press.

Mkyong, M. (2018, November 29). *Java JMH Benchmark Tutorial - Mkyong.com*. Mkyong.com. Retrieved March 15, 2023, from <https://mkyong.com/java/java-jmh-benchmark-tutorial/>

Patil, A. (2021, August 21). *All You Need To Know About Garbage Collection in Java*. dzone.com. Retrieved March 15,

2023, from <https://dzone.com/articles/all-you-need-to-know-about-garbage-collection-in-j>

Poggi, N. (2019). *Microbenchmark*. Springer EBooks, 1143–1152. https://doi.org/10.1007/978-3-319-77525-8_111

Rana, M. E. (2017). The Effect of Applying Software Design Patterns on Real Time Software Efficiency. *Future Technologies Conference (FTC)*. https://www.researchgate.net/publication/332556324_The_Effect_of_Applying_Software_Design_Patterns_on_Real_Time_Software_Efficiency

Sierra, K., & Bates, B. (2005). *Head First Java: A Brain-Friendly Guide*. “O’Reilly Media, Inc.”

Vogel, L. (2023, March 2). *Java concurrency (multi-threading) - Tutorial*. Vogella. Retrieved March 15, 2023, from <https://www.vogella.com/tutorials/JavaConcurrency/article.html>

Zafar, S. (2022, March 31). *Real-time Systems: Full Importance of Latency, Scheduling and Memory*. Data Fifty. <https://datafifty.com/real-time-systems-overview/>