INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT087-3-3-RTS

REAL-TIME SYSTEMS

APU3F2211CS(DA)

HAND OUT DATE: 19 DECEMBER 2022

HAND IN DATE: 22 MARCH 2023

WEIGHTAGE: 50%

STUDENT NAME: RYAN MARTIN

TP NUMBER: TP058091

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

Ryan Martin - TP058091

Abstract - **This paper looks into how program design and software factors affect the performance of real-time systems. The primary focus of this study is to compare the use of Java and Clojure, two programming languages that target the JVM, and how the language can affect the performance of the application. The comparison is done using a combination of performance benchmarking and system profiling. The results of the comparison shows that while the Java implementation has a faster execution speed, it also has higher memory usage, unlike the Clojure implementation which uses less memory but is slower. In general, Java is more suitable for real-time systems due to its more predictable garbage collection, however in the end the choice of software to use depends on the unique requirements of any system.**

I. INTRODUCTION

In modern computing systems, real-time performance is a critical aspect of many applications. A real-time system is a type of computer system that is designed to respond within a guaranteed time constraint, i.e. its actions should meet a specified deadline [1]. One example of a real-time system is a flight control system, which is at the centre of any aircrafts such as aeroplanes and drones. Flight control systems today use electronic controls that communicate with various parts of the aircraft such as its wing flaps and engines [2]. This system must operate in real-time to ensure the safety of the aircraft and its passengers.

The performance of a real-time system depends on several factors which will be explored in this paper. These factors include hardware factors such as processor speed and memory, as well as software factors such as software design and architecture. While performance is an important aspect of any system, it is especially crucial that a real-time system runs performantly due to the applied time constraints. As most real-time systems are implemented in critical areas such as a flight control system, they need to be highly optimised in order to run smoothly and meet all of the specified deadlines.

Regarding deadlines and time constraints, real-time systems can be categorised based on how important it is for them to meet their deadlines. Hard real-time systems must always meet their deadlines otherwise there will be serious consequences, while soft real-time systems can occasionally miss their deadlines [3]. A flight control system is an example of a hard real-time system, as any failure to meet its deadline can cause a whole aircraft to crash. On the other hand, first person shooter (FPS) games are an example of a soft real-time system. If the system failed to meet its deadline, the worst that could happen is the player not being able to have a smooth gaming experience.

In this paper, the software factors affecting the performance of real-time systems will be explored in detail and tested in an actual real-time program. The program is a simulation of a flight control system that uses concurrency concepts and messaging queues written in languages that target the Java Virtual Machine (JVM). The focus of this study will be to compare the real-time performance of the same program written in two different languages, Java and Clojure. To achieve this, the two programs will be benchmarked and profile, then compared against each other. Through this experiment, the choice and design of software will be explored to understand their effects on the performance of real-time systems.

The rest of the paper is organised as follows. Section two provides an overview of the previous research done in the topic of real-time computing. Literature review will be conducted to establish a knowledge base for this research. Section three will describe the methodology for this research including the implementation details, benchmarking and profiling tools used, and how the two programs will be compared against each other. Section four presents the results of the comparison as well as a discussion on these results. Finally, section six will conclude this paper and provide some suggestions for future research.

II. BACKGROUND

In this section, a comprehensive literature review on the effects of software choice and design will be conducted. A discussion on the use of the JVM for real-time systems will also be included here, as both Java and Clojure compile to Java bytecode. Concurrency will also be a major theme for this secondary research, as it is commonly used in real-time systems. Concurrency is the ability for a program to execute multiple tasks simultaneously, albeit not in parallel. By applying concurrency concepts, real-time systems are able to execute their tasks faster and meet their strict deadlines.

A. Software Factors Affecting Real-Time System Performance

Real-time systems are highly dependent on several software-related factors such as the software design, architecture, as well as the choice of the programming

language and concurrency model. One important aspect of any real-time system is the application of concurrency. As most real-time systems are embedded, they usually have access to only one processor. Therefore, it is really important to use concurrency to improve the responsiveness and effectiveness of the system [4]. By executing multiple tasks at the same time, tasks can be completed faster and on time. However, this does come with some drawbacks.

The use of concurrency involves adding complexity to the system. This can lead to issues such as blocked processes caused by the sharing of global resources, deadlocks, thread starvation, and race conditions [5]. It also increases the chances of logic errors in code, which even the best programmers might miss. One aspect of concurrency to consider for real-time systems are threads. A thread can be defined as a task executed within a process [6]. There are different types of threads, user-level threads which are implemented by a user-level library, and kernel-level threads which are managed by an operating system's (OS) kernel. For real-time systems in general, it is better to use user-level threads, as it is faster to create and it reduces various overheads on the OS. It also allows for concurrency in systems with only a single processor, which is common in embedded systems.

Most real-time systems utilise some form of real-time operating system (RTOS). An RTOS is a type of OS specialised for handling real-time systems. Most RTOS provide abstractions over concurrency concepts, simplifying the development process and reducing the risks for logic errors [4]. According to [7], there are several factors that might affect the performance of a RTOS which includes time management, task management or scheduling, interrupt management, synchronisation, and shared resource management. They also touched on memory management, however it is not a major factor affecting the performance of the OS.

Another factor that can affect the performance of a real-time system is the system design. [4] discussed two different system design paradigms for real-time systems, time-driven style and event-driven style with object-oriented models. The time-driven style is based on a simple concurrency model, where a task is activated by an event trigger and performs some computation before waiting for the next event. This model is suitable for regular and recurring events which are predictable in nature. An event-driven model focuses on event handling code, and is suitable for handling unpredictable and irregular events. These styles are suitable for different situations, so it is important to select the most suitable system design according to the task in hand. The flight control system used in this study is an example of a system that uses the event-driven paradigm.

B. The Use Of JVM For Real-Time Systems

The JVM is one of the most used language virtual machines today. It is a very robust language specification, leading to its use in many enterprise systems worldwide. While at first it was designed to be the VM for the Java programming language, now there are many different languages that target the JVM. Popular examples include Scala, Groovy, Kotlin, and Clojure, although there are probably hundreds of other language implementations that compile to Java bytecode.

While the JVM is suitable for developing multi-platform software and complex applications, it is not highly suited for developing real-time systems. Even though it is optimised for high execution speed and memory usage, the sole fact that it uses an automatic garbage collector (GC) makes the nature of its execution nondeterministic. The GC will run when the memory threshold is reached, which can lead to unpredictable delays in the program execution. This makes the JVM inappropriate for hard real-time systems because of the necessity of meeting the deadlines. For soft real-time systems though, the use of JVM can still be argued. In fact, there are also specifications such as the Real-Time Specification for Java (RTSJ), which provides a set of application programming interfaces (APIs) and rules for developing real-time applications using Java [8].

As such, the flight control system simulation programs to be used for this study were written in Java and Clojure, two languages that utilise the JVM. *"Java runs on three billion devices worldwide"* was the slogan used by Oracle to promote the use of Java. It is the sixth most popular programming language in 2022, and as such no introduction is needed for it [9]. On the other hand, Clojure is a newer language created in 2007 by Rich Hickey. It is a dynamically typed, functional, general purpose programming language and a modern dialect of Lisp. The main difference Clojure has with Java, besides the syntax, is the functional programming paradigm which encourages the use of immutable data structures as well as higher order functions. It is also designed to make concurrency easier for programmers by building it into the language constructs [10].

III. METHODOLOGY

This section will describe in detail the comparison of the performance of the flight control system implemented in both Java and Clojure. The same system design and architecture will be used for both languages, but tweaked to use the idioms and best practices of each language. The Java Microbenchmarking Harness (JMH) will be used to benchmark the Java code, while a manual method will be used for Clojure. Both programs will also be profiled for their system resource usage using JProfiler, a popular profiling tool for Java programs.

A. System Overview

The system consists of four main parts, the sensors, the flight controller (FC), the global state, and the main process. The main process starts the other parts and the simulation. The sensors and the FC are the senders and receivers of messages through RabbitMQ. RabbitMQ is an open-source message broker that provides a reliable service for many messaging applications [11]. The Java implementation will use the official client library provided by RabbitMQ, while the Clojure implementation will use a third-party wrapper of the RabbitMQ client called Langohr [12]. The sensor sends data about the global state to the FC, which activates certain parts of the plane depending on the data captured.

The global state stores the information about the simulated world, e.g., the current altitude, weather conditions, etc. The next few sections will explain in detail the implementation of these parts.

1) State: This is a global store of the state of the simulated world. It contains six fields*;* distance to destination, current altitude, current speed, cabin pressure, weather condition, and a boolean indicator of whether the plane is preparing for landing. In Java, it is implemented as a class, while in Clojure is implemented as a hash map.

```
public class State {
  private int distance = 1000; // distance to destination
  private int altitude = 11000;
  private int pressure = 540;
  private int speed = 50;
  private int weather = 10; // 0-10, bad to good
  private boolean isLanding = false;

  public State() {}

  public synchronized int getAltitude() {
    return altitude;
  }
  // ... the same for the other properties
}
```

Above is the code snippet for the Java implementation of the global state. As it is a shared resource, all access to its properties is done through synchronised functions.

```
(def state (ref {:distance 1000
                 :altitude 11000
                 :speed 50
                 :pressure 540
                 :weather 10
                 :landing? false}))
```

Above is the code snippet for the Clojure implementation of the global state. It is implemented as a "ref", which is a reference type provided by Clojure to manage shared resources across multiple threads [13]. This ref holds a hash map of the state values.

2) Sensors: There are five sensors that correspond with the first five values of the global state. The main function of the sensor is to make alterations to the state and publish the value as a message into each of their own queues in RabbitMQ. An example of this is the GPS sensor, which decrements the state's distance property by 50 and publishes it every two seconds. This is the most important part of the whole program as the simulation ends when this value reaches zero. The rest of the sensors will randomly generate values to trigger certain responses from the FC.

```
public class Sensor implements Runnable {
  private ConnectionFactory factory;
  private State state;
  private String name;
  private DataGenerator generator;

  public Sensor( ... ) { ... }

  @Override
  public void run() {
    try (Connection con = factory.newConnection()) {
      Channel ch = con.createChannel();
      ch.queueDeclare(name, false, false, false, null);

      try {
        while (state.getDistance() > 0) {
          int data = generator.generate();
          state.set(name, data);
          ch.basicPublish("", name, null,
           String.valueOf(data).getBytes());
          System.out.printf("measured X");
          Thread.sleep(2000);
        }
      } catch (InterruptedException e) {
        System.err.println(e.getMessage());
      }
     System.out.printf("closing connection");
    } catch (TimeoutException | IOException e) {
      System.err.println(e.getMessage());
    }
  }
  // ... other utility methods
}
```

The Java implementation of a sensor implements the *Runnable* interface. It takes in the state, a RabbitMQ *ConnectionFactory*, the queue name, as well as a *DataGenerator*. This generator is the function to generate new values for the state, and is created and provided in the main process. It will be discussed in the section about the main process in later sections.

```
(defn run [name generator]
  (let [con (rmq/connect)
        ch (lch/open con)
        unit (get-unit name)]
    (lq/declare ch name
                {:exclusive false :auto-delete false})
    (loop []
      (when (> (:distance @state) 0)
        (let [data (generator)]
          (dosync
            (alter state assoc (get-attr name) data))
          (lb/publish ch "" name (str data))
          (print "measured X")
           (Thread/sleep 2000)
           (recur))))
    (rmq/close ch)
    (rmq/close con)
    (print "closing connection")))
```

The Clojure implementation is essentially the same as the Java one. It takes in the queue name and the data generator. The state is defined in another file in the same namespace and is imported in this file. The state is updated using *dosync*, which runs a synchronous transaction to ensure only one operation can run on it at a time.

3) FC: The FC contains the main logic behind the whole simulation. It consumes the messages published by the sensors, and then handles them based on the message queue. Each state property has a corresponding handler. Same as the sensors, the handlers will keep running until the plane has landed, i.e. until the state's *distance* property reaches zero. The handler functions are quite similar to each other so only two of them will have their code snippets shown later.

```
public class FC {
  private State state;
  private ConnectionFactory factory;

  public FC(State state, ConnectionFactory factory) {
    this.state = state;
    this.factory = factory;
  }

  public void run() throws IOException, TimeoutException {
    Connection con = factory.newConnection();
    Channel chGPS = con.createChannel();
    Channel chAltitude = con.createChannel();
    // ... the same for the other channels

    chGPS.queueDeclare(Main.QUEUE_GPS, false, false, false,
null);
    chAltitude.queueDeclare(Main.QUEUE_ALTITUDE, false, false,
false, null);
    // ... the same for the other queues

    ExecutorService executor = Executors.newFixedThreadPool(5);
```

```
    executor.execute(() → handle(chGPS, Main.QUEUE_GPS));
    executor.execute(() → handle(chAltitude,
Main.QUEUE_ALTITUDE));
    // … the same for the other channels

    executor.shutdown();
    while (!executor.isTerminated()) {}

    chAltitude.close();
    chGPS.close();
    // … the same for the other channels
    con.close();
  }

  private void handle(Channel ch, String queue) {
    try {
      while (state.getDistance() > 0) {
        DeliverCallback callback = null;
        switch (queue) {
          case Main.QUEUE_GPS:
             callback = handleGPS; break;
           case Main.QUEUE_ALTITUDE:
             callback = handleAltitude; break;
          // … the same for the other queues
        }
        if (callback ≠ null) {
          ch.basicConsume(queue, true, callback, consumerTag →
{});
        }
      }
    } catch (IOException e) {
      System.err.println(e.getMessage());
    }
  // … handler functions
  }
```

The Java code snippet above contains a lot of repetitive code, so some parts have been commented out to keep it short and understandable. Like the *Sensor* class, the *FC* class also implements the *Runnable* interface. Its *run* function subscribes to all queues, then creates an executor service to run the different handler functions in different threads. The *handle* isn't a handler function, rather it allocates the handler functions to the channels based on the queue name.

```
private DeliverCallback handleGPS = new DeliverCallback() {
  @Override
  public void handle(String consumerTag, Delivery message)
throws IOException {
    String msg = new String(message.getBody(), "UTF-8");
    int distance = Integer.parseInt(msg);
    // arbitrary distance value to start landing
    if (distance ⩽ 250
        && distance > 0
        && !state.getIsLanding()) {
      System.out.println("initiating landing");
      state.setIsLanding(true);
    }
    if (distance == 0) {
      System.out.println("Successfully landed");
    }
  }
};
```

A handler function in Java implements the *DeliverCallback* class provided by the RabbitMQ client library. The code snippet above shows the GPS handler which sets the *isLanding* state property if the distance is 250 km from the destination and prints a success message when the plane has landed.

```
private DeliverCallback handlePressure = new DeliverCallback()
{
  @Override
  public void handle(String consumerTag, Delivery message)
throws IOException {
    String msg = new String(message.getBody(), "UTF-8");
    int pressure = Integer.parseInt(msg);
    if (pressure < 500) {
      System.out.println("low cabin pressure, deploy oxygen
masks");
      state.setPressure(pressure + 75);
    }
  }
};
```

Above is another example of a handler function, this time it is for cabin pressure. It checks if the pressure is below a certain threshold (in this case, 500) and deploys the oxygen masks, or rather just prints the message here. It then increases the value of the global state's *pressure* by 75 pascals, which simulates the plane's efforts to stabilise the plane's cabin pressure. After a few seconds, the pressure should get to a safe value of around 500.

```
(defn handle [ch queue handler]
  (loop []
    (when (> (:distance @state) 0)
      (lc/subscribe queue ch handler {:auto-ack true})
      (recur))))

(defn run []
  (let [con (rmq/connect)
        ch-gps (lch/open con)
        ch-altitude (lch/open con)
        ;; … the same for the other channels

        [ch q handler]
        [[ch-gps queue-gps handle-gps]
         [ch-altitude queue-altitude handle-altitude]
         ;; … the same for the others
         [ch-weather queue-weather handle-weather]]

        futures
        (map #(future (handle %1 %2 %3)) ch q handler)]

    (doseq [completion futures]
        @completion)
    (rmq/close ch-gps)
    (rmq/close ch-altitude)
    ;; … the same for the other channels
    (rmq/close con)))
```

The Clojure implementation of the FC is again very similar to its Java counterpart. The main difference is that this code uses Clojure's *futures* instead of Java's *ExecutorService* (which is also available for use in Clojure). A future in simple terms is a code that runs on its own and independent thread [14]. The code above runs the *handle* function on each of the channels and stores them in the *futures* array. The handle function simply calls the *subscribe* function (equivalent to *basicConsume* in Java) with the appropriate handlers for each channel. Finally, the function waits for all of the threads to complete (by dereferencing the futures using *@completion*) and then closes all connections to RabbitMQ [15].

```
(defn handle-gps [_ch _meta payload]
  (let [distance
        (Integer/parseInt (String. payload "UTF-8"))]
    (cond
      (and (⩽ distance 250)
           (> distance 0)
           (not (:landing? @state)))
      (do
        (println "initiating landing")
        (dosync (alter state assoc :landing? true)))

      (zero? distance)
      (println "successfully landed"))))

(defn handle-pressure [_ch _meta payload]
  (let [pressure
        (Integer/parseInt (String. payload "UTF-8"))]
    (when (< pressure 500)
      (println "low cabin pressure, deploying oxygen masks")
      (dosync
        (alter state assoc :pressure (+ pressure 75))))))
```

Above is the equivalent Clojure code for the *handleGPS* and *handlePressure* callbacks in Java. Again,

the difference is in the language constructs used, not the program logic.

4) Main Process: The main process is just the program entry-point. Its purpose is to initialise the global state and to set up the sensors, the FC, as well as the data generators mentioned earlier. In Java, they are implemented as an interface, while in clojure they are regular anonymous functions.

```
public class Main {
  public static final String Q_GPS = "gps";
  public static final String Q_ALTITUDE = "altitude";
  // ... the same for the other queues

  public static void main(String[] args) {
    State state = new State();
    ConnectionFactory factory = new ConnectionFactory();
    Random rand = new Random(SEED);

    Sensor g = new Sensor(Q_GPS, state, factory, () → {
      return state.getDistance() - 50;
    });
    Sensor a = new Sensor(Q_ALTITUDE, state, factory, () → {
      return state.getAltitude()
        + rand.nextInt(200)
        * (rand.nextBoolean() ? 1 : -1);
    });
    // ... the same for the oher generators

    ExecutorService executor = Executors.newFixedThreadPool(5);
    executor.execute(g);
    executor.execute(a);
    // ... the same for the other sensors

    System.out.println("simulation started");
    FC fc = new FC(state, factory);
    try {
      fc.run();
    } catch (TimeoutException | IOException e) {
      System.err.println(e.getMessage());
    }

    executor.shutdown();
    while (!executor.isTerminated()) {}
    System.out.println("simulation completed");
  }
}
```

Above is the main function for the Java implementation. It first initialises the sensors with their data generators. The *DataGenerator* interface only has one method; *generate*, which returns an integer. This allows for the use of arrow functions provided by Java to initialise the interface. After this, similar to the FC, an executor service will be used to run each sensor in their own thread. Then, it will run the FC, wait for all threads to finish and finally print out a message indicating the completion of the simulation program.

```
(defn start-sensors []
  (doseq [[queue generator]
          [[queue-gps #(- (:distance @state) 50)]
          [queue-altitude
          #(+ (:altitude @state)
              (* (rand-int 200) (rand-nth [1 -1])))]
          ;; ... the same for the other sensors
          [queue-weather #(rand-int 10)]]]
          (future (sensor/run queue generator))))

(defn -main [& _args]
  (println "simulation started")
  (start-sensors)
  (fc/run)
  (shutdown-agents)
  (println "simulation completed"))
```

The main function in Clojure also follows a similar pattern as its FC. It creates the data generators for the sensors, starts the sensors in their own threads using futures, starts the FC, waits for all threads to finish (through *shutdown-agents*), and finally prints the final message.

B. *Performance Benchmarking*

As mentioned earlier, the benchmarking will be done using JMH for Java and manually for Clojure. The average time per operation will be the main indicator of performance for this system. The benchmarking will be done in five iterations, preceded by two rounds of warm ups.

```
@BenchmarkMode(Mode.All)
@Fork(value = 1)
@Warmup(iterations = 2)
@Measurement(iterations = 5)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public class MyBenchmark {
  @Benchmark
  public void benchmark() {
  // ... same code as main
  }
}
```

The benchmark code for Java is implemented using the same code as the main function but in a separate MyBenchmark class with the JMH benchmark annotations.

```
(defn benchmark []
  (let [start (.truncatedTo (Instant/now)
                            ChronoUnit/MICROS)
        m (-main)
        end (.truncatedTo (Instant/now)
                          ChronoUnit/MICROS)]
        (.toMillis (Duration/between start end))))
(println (benchmark))
```

Above is the code for manually benchmarking the Clojure program. Although there are third-party libraries for benchmarking Clojure such as Criterium and JMH-Clojure, unfortunately the student couldn't get them to work with the existing code due to the *shutdown-agents* function which is called in the main function. When that function is called, not only will the threads from the main program stop, but the threads running for the benchmarking will also stop. However, without this code, the threads used in the future calls have to wait a few minutes before terminating. Therefore, albeit at the cost of benchmarking accuracy, a manual method was used.

C. *System Profiling*

Both programs will be compiled into Java Archives (JARs) and run through JProfiler to check their system resource usage. The main values to analyse through profiling are the CPU and memory usage, as well as the JVM's GC activity. These values are important to take into consideration when designing real time systems such as a flight control system.

IV. RESULTS & DISCUSSION

Below are the results of the Java program benchmarks. Since the benchmarking mode is set to all modes in the code, only the section on the average time will be shown for discussion.

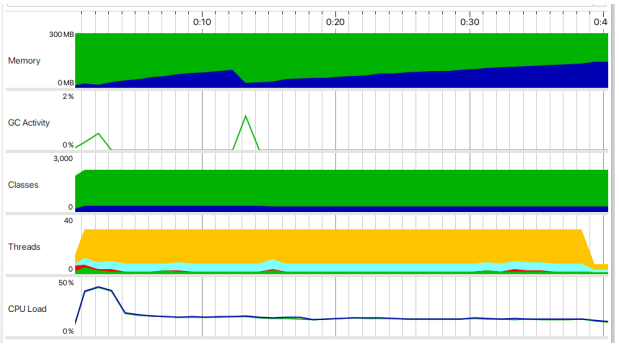
```
# JMH version: 1.36
# VM version: JDK 17.0.6, OpenJDK 64-Bit Server VM, 17.0.6+10
# VM invoker:
/usr/lib/jvm/java-17-openjdk-17.0.6.0.10-1.fc36.x86_64/bin/java
```

```
# VM options: -Dfile.encoding=UTF-8
-Djava.io.tmpdir=/home/rmrt1n/Documents/SEM5/RTS/me/app/build/t
mp/jmh -Duser.country=US -Duser.language=en -Duser.variant
# Blackhole mode: compiler (auto-detected, use
-Djmh.blackhole.autoDetect=false to disable)
# Warmup: 2 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Average time, time/op
# Benchmark: rts.MyBenchmark.benchmark

# Run progress: 33.33% complete, ETA 00:09:23
# Fork: 1 of 1
# Warmup Iteration   1: simulation started
//  ... skipped
simulation completed
40360.335 ms/op

# Warmup Iteration   2: simulation started
//  ... skipped
simulation completed
40096.945 ms/op

Iteration   1: simulation started
//  ... skipped
simulation completed
40064.460 ms/op

Iteration   2: simulation started
//  ... skipped
simulation completed
40136.996 ms/op

Iteration   3: simulation started
//  ... skipped
simulation completed
40622.023 ms/op

Iteration   4: simulation started
//  ... skipped
simulation completed
40075.916 ms/op

Iteration   5: simulation started
//  ... skipped
simulation completed
40046.654 ms/op

Result "rts.MyBenchmark.benchmark":
  40189.210 ±(99.9%) 940.799 ms/op [Average]
  (min, avg, max) = (40046.654, 40189.210, 40622.023), stdev =
244.322
  CI (99.9%): [39248.412, 41130.009] (assumes normal
distribution)
```

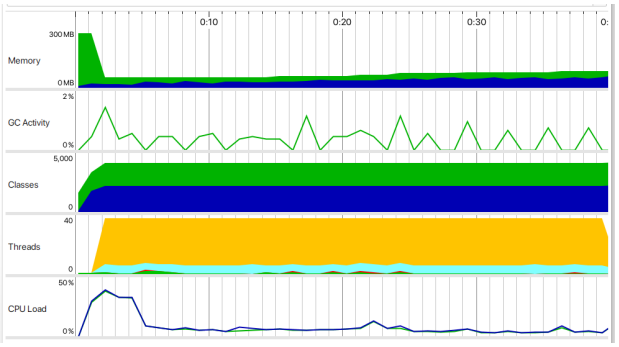
It can be seen that the average time of the Java program is around 40.189 seconds. There isn't much deviation between the iterations, only around 244 milliseconds.

```
warmup 1: 38848
warmup 2: 38601
iteration 1: 38617
iteration 2: 38586
iteration 3: 39166
iteration 4: 38641
iteration 5: 38741
avg: 38750.2
```

Above are the results for the Clojure benchmarking. Surprisingly it runs around 38.750 seconds on average, which is two seconds less than the Java program.



Above is the screenshot of the Java simulation under JProfiler. The program ran for 40.3 seconds. CPU usage reached a max of 43.29% at the start of the program, which falls down into a stable area of around 15-18%. Memory usage on the other hand starts out low, then once it hits around 94MB, the GC runs and sweeps the excess memory. It then continues to rise until the end of the program.



Above is the result of the profiling on the Clojure program. While the CPU usage is similar to its Java counterpart, its memory usage and GC activity is very different. While memory usage is way lower and more consistent than Java, its GC activity is very active and almost regular. This might be the reason why the memory usage is so low. However, the program ran for 45 seconds, which is far off from the benchmarking results.

V. CONCLUSION

Overall, the results are somewhat surprising. The memory usage of the Clojure program is lower, which may be caused by its more active GC. However, this leads to a slower execution time. Java on the other hand, has a faster execution speed but with more memory consumption. While the benchmarking results suggest that the Clojure implementation is better, as it is a manual method the results might not be very accurate. If given more time, a better benchmarking will be conducted for the Clojure code to ensure a more even comparison. For a real-time system, based purely on the comparison done in this study, Java might be the better choice due to its less active GC, which results in more predictable program flow. However, Clojure might be worth considering for low memory environments as its memory usage is very low due to its GC. Ultimately, both languages have their own unique pros and cons, and it is up to the developer to choose the best one for their unique use case.

REFERENCES

[1] GeeksforGeeks, “Real Time Systems,” GeeksforGeeks, Jan. 12, 2022. https://www.geeksforgeeks.org/real-time-systems/ (accessed Mar. 18, 2023).

[2] McMahon, “What is a Flight Control System?,” Wiki Motors, Feb. 23, 2023. https://www.wikimotors.org/what-is-a-flight-control-system.htm (accessed Mar. 18, 2023).

[3] GeeksforGeeks, “Difference between Hard real time and Soft real time system,” GeeksforGeeks, Feb. 05, 2023. https://www.geeksforgeeks.org/difference-between-hard-real-time-and-soft-real-time-system/ (accessed Mar. 18, 2023).

[4] M. Saksena and B. Selic, “Real-Time Software Design - State of the Art and Future Challenges,” IEEE Canadian Review, Tech. Rep., Jan. 1999, [Online]. Available: https://ewh.ieee.org/reg/7/canrev/canrev32/manas.pdf

[5] GeeksforGeeks, “Concurrency in Operating System,” GeeksforGeeks, Mar. 02, 2023. https://www.geeksforgeeks.org/concurrency-in-operating-system/ (accessed Mar. 18, 2023).

[6] P. Pedamkar, “Threads in Operating System,” EDUCBA, Jul. 13, 2021. https://www.educba.com/threads-in-operating-system/ (accessed Mar. 18, 2023).

[7] M. Huang, S. Guo, X. Liang, and X. Song, “Research on Real-time Performance Testing Methods for Robot Operating System,” Journal of Software, vol. 9, no. 10, pp. 2685–2692, Jan. 2014, doi: 10.4304/jsw.9.10.2685-2692.

[8] Oracle, “An Introduction to Real-Time Java Technology: Part 1, The Real-Time Specification for Java (JSR 1),” Oracle. https://www.oracle.com/technical-resources/articles/javase/jsr-1.html (accessed Mar. 19, 2023).

[9] Stack Overflow, “Stack Overflow Developer Survey 2022,” Stack Overflow. https://survey.stackoverflow.co/2022/#technology (accessed Mar. 19, 2023).

[10] Hickey and Visualmodo, “Clojure Vs. Java: Comparison Guide In 2021 - visualmodo - Medium,” Medium, Jan. 06, 2022. https://medium.com/visualmodo/clojure-vs-java-comparison-guide-in-2021-136d11a6478a (accessed Mar. 19, 2023).

[11] P. Pedamkar, “What is RabbitMQ?,” EDUCBA, Mar. 30, 2021. https://www.educba.com/what-is-rabbitmq (accessed Mar. 21, 2023).

[12] ClojureWerkz Team, “Langohr, an idiomatic Clojure RabbitMQ client | Clojure AMQP 0.9.1 client.” http://clojurerabbitmq.info/ (accessed Mar. 21, 2023).

[13] “Clojure - Refs and Transactions,” Clojure. https://clojure.org/reference/refs (accessed Mar. 21, 2023).

[14] J. Cruz, “Back to the Future in Clojure - Level Up Coding,” Medium, Dec. 13, 2021. https://levelup.gitconnected.com/back-to-the-future-in-clojure-934b85a3d08e (accessed Mar. 21, 2023).

[15] noisesmith, “Clojure Synchronize Futures with Await,” Stack Overflow. https://stackoverflow.com/questions/27347563/clojure-synchronize-futures-with-await (accessed Mar. 21, 2023).