INDIVIDUAL ASSIGNMENT

Real Time Systems (RTS) CT087-3-3-RTS

Assessment Title	:	SMART WATCH SIMULATION
Intake Code	:	UC3F2011CS(DA)
Module Code	:	CT087-3-3-RTS
Student Name	:	CHAN JIA LE
TP Number	:	TP049952
Submission Date	:	19th March 2021
Lecturer Name	:	Dr. Imran Medi

Table of Contents

Smart Watch Simulation

Chan Jia Le TP049952

Abstract

The research conducted examines the performance impact caused by design of the real time system. Background studies has been conducted from numerous sources and articles to understand the important aspect that should be taken for consideration when evaluating performance of the system like response time, garbage collection etc. 16 Iterations of different implementations has been developed and will be explained thoroughly on how each design differs amongst each of them. Furthermore, two tests are conducted for examining the response time, memory & CPU usage, and garbage collection for different iterations of designs. The study will conclude the finding of how design has impacted the performance.

1.0 Introduction

Real Time Systems (RTS) is systems which has behaviors that are dependent on time elapsed. The output or response time of the system must be limited and predictable and can be classified as several forms (Zambrano, et al., 2017). In addition to that, it crucial to understand the key terms that defines a real time system which includes determinism, deadline, and quality of service. Determinism refers to the system being consistent on providing the same output from a known input, whereas deadline refers to the time the system is required to complete the task and lastly, quality of service refers to the overall quality or performance of the system which takes factors like throughput, jitter, error rates, etc. into consideration (Kay, 2020). Furthermore, The classification of Real Time System are either Hard Real Time Systems or Soft Real Time Systems which could be differentiate based on the consequences of missing a scheduled deadline (Doug, 2018).

Soft Real-time system

Soft Real Time Systems are systems whereby satisfying scheduled deadline is not compulsory and a miss in the scheduled deadline will not cause any catastrophe events. However, if deadline is often not satisfied, the performance of the system will be affect the experience of the user (Electronics Hub, 2015). Systems like Web browsing, Mobile communication, DVD player are considered as soft real time systems (VIVADIFFERENCES, 2020).

Hard Real-time system

Hard Real Time Systems on the other hand is the opposite of soft real time system, whereby it is deterministic and time constraint. Hence, meeting a scheduled deadline is compulsory, even a slight delay might cause a serious impact or catastrophe.

Hence, it is mandatory for Hard Real Time Systems to meet its scheduled deadline (Electronics Hub, 2015). Some references of hard real time systems are Air Traffic Control Systems, Railway signaling system, etc. (VIVADIFFERENCES, 2020).

The Smart Watch Simulation System is a system that will monitor and track user's heart rate and blood oxygen level in a regular interval. Upon completion of the checking, the vital details will be written inside the system and a notification will be prompted for the user on whether there are irregularity in the vitals, if there are any irregular details that is detected, the checking will end and the notification will alert the user while at the same time call the ambulance.

This study will take a deep dive into the implementation of a real time system, developing the Smart Watch Simulation System using Java along with additional analysis on the issues that is faced by the developer when conducting the development of the real time system. Furthermore, implementing different concurrent concepts and approach to seek the most efficient method as well as the design methods to avoid when constructing a real time system. Overall, the research conducted will outline the importance of design for constructing an efficient and reliable real-time system.

The research is arranged in the following structure. Section 2 will provide a background research by conducting literature review for answering research questions. Furthermore, the following section will be the methodology section which will provide a detailed explanation on the methods, benchmarking and test conducted to achieve the overall objective. Other than that, section 4 will provide a discussion on the results and findings of those tests and benchmarking. In the end, a conclusion will round up the whole research on the findings of the research.

2.0 Literature Review

2.1 Performance of a real time system

According to (Halang, et al., 2000), Performance measurement of real time systems or in short benchmarking is conducted based on the characteristics of the real time system taking into consideration the latency, timeliness, deadlock prevention, safety, etc. and further categorizes it into Qualitative binary criteria, Qualitative gradual criteria and quantitative criteria. Similarly, (Mc Graw Hill, 2017) takes a deep dive into methods of evaluating real time system performance and design, and further emphasizes on requirements for the analysis of a real time system which is homogeneous to the research stated above, including interrupt

handling, response time, data transfer rate, priority handling and task synchronization.

In addition to that, the statement is further backed up by research conducted by (Claypool, 2005) and (Keycdn, 2018) stating that latency (the time of delay in terms of sending one piece of information or processes from a point to another) would significantly affect the response time of the system, hence concluding that a good latency is crucial for a real time system. In addition to the statement, research conducted by (Chen, et al., 2019) further adds on that the timeliness of the system is crucial and purposes recursive algorithms with the understanding and analysis of schedulable conditions to optimize the start time and allocation of processors reduce the chances of missing deadlines. A research conducted by (Wolf, 2014) purposes that multiprocessors are a crucial element for real-time computation and further emphasizes to adapt and optimize the components to provide the best overall results (efficiency/ reliability).

In addition to that, research conducted by (Heide & Halang, 1991) concluded a series of metrics to be considered when benchmarking which involves in understanding the measures, comparison between different versions, search for balanced benchmarks, etc. These metrics had served as a basis for measurement for many other researches. (Sharma, et al., 2015) further adds details about measuring real time system, stating the limitation and challenges faced when measuring Worst Case Execution Time (WCET) and further explores the different approaches using tools like aiT, Bound-T, Chronos, etc. which adds value on the proposition of using tools for monitoring and measuring real time systems.

To conclude, the following studies has outlined the methods or tools needed to perform an evaluation on the real time systems and emphasis on how design and methods of constructing the system could affect the performance of the development for the real time systems along with the attributes or characteristics to pay attention on during evaluation.

2.2 Real Time Systems concepts on Java

Real-Time Specification for Java (RTSJ) is developed by professionals on real-time domains to make developing real time systems using Java possible. It introduces a variety of new features including memory management models, thread types and framework (Oracle, 2008). As mentioned in the statement above, real time systems involves in a series of deadlines for task and It can be categorized as follows, periodic, sporadic and aperiodic (Bench Partner, 2019). **Periodic Task** is task that happens in a regular interval like sensor reading data

every five seconds, **Aperiodic Task** is task that happens randomly, like sensor stops work when emergency is detected and **Sporadic Task** is similar to Aperiodic task but contains maximum frequency (GeeksforGeeks, 2019).

Application of Java is possible, but it contains many unpredictable events that may cause the system to miss its deadline according to (Oracle, 2008). One of the events includes in the unpredictable scheduling of the operating system, the creation of thread can be affected by the unpredictability of operating systems scheduling policy (Gupta, et al., 1991). Furthermore, as threads can be prioritize using Java, the different priorities might cause priority interventions causing higher priority threads to wait on resources from low priority threads which could led to delays in meeting deadline (Welc, et al., 2004). In addition to that, One of Javas features which is Java Garbage Collection which is triggered by JVM, the more live objects found in the system, the suspension in performance will be longer causing response time and throughput of the system to suffer (Dynatrace, 2020).

Other than that, as mentioned a real time system is consisted of numerous threads that performance concurrent task. The threads all have their own life cycle, the life cycle of threads generally includes new, runnable, running, blocked/waiting, dead/terminated (GeeksforGeeks, 2019). The details of each state are stated as follows:

New

During this state, the thread object is created but not yet started or alive as it is not started using the start() method in Java yet (javaTpoint, 2020).

Runnable

The start() function is called upon the beginning of the thread object and the state of the thread is now runnable. Control over the thread is given to the Thread scheduler upon its execution depending on the OS implementation of thread scheduler (Pankaj, 2020).

Running

In this state, the CPU executes the thread and thread scheduler picks the threads from a thread pool to execute and It is highly dependent on OS and CPU scheduling (Pankaj, 2020).

Blocked/ Waiting

Blocked/Waiting state is the state whereby the thread is still alive but it is restricted from execution based on defined conditions like thread.wait(). The thread can be reactivated by notifying it like thread.notify() (GeeksforGeeks, 2019).

Dead/ Terminated

Once the thread has finished performing its task, the thread will then be terminated and will not consume any cycles of the CPU, it is important to note that threads can also be terminated if there is error because of unhandled exceptions (GeeksforGeeks, 2019).

Overall, implementation of Real Time concepts using Java is applicable and some aspects of common elements that will affect the performance is stated in the study and should be taken into consideration when developing real time system using java.

2.3 Memory Management & Garbage Collection in Java

Garbage Collection (GC) is often characterised as Automatic Memory Management and is an essential component of the Java runtime system, the use of garbage collection has simplified many tasks of optimization on software. However, garbage collection is mostly neglected or unwanted in real time systems due to the unpredictable behaviour that could potentially cause delay in satisfying deadlines (Schoeberl, 2006). Research conducted by (Agesen, et al., 1998) further adds that garbage collections on Java is often times too conservative meaning that root set is often overestimated causing large amount of unexpected garbage, and further emphasizes on the need of optimizing objects and utilization of Java heap to avoid such unexpected results.

Additional research has been conducted by (Huang, et al., 2004) and (Stichnoth, et al., 1999) strengthens the above statements and added different propositions on methods of improving Garbage collection performances by reducing the GB maps or utilizing specific tools and architectures like MMTk which is a memory management tool kit. Furthermore, Real Time Specification for Java has defined scoped and immortal memory to assist Java Heap enabling an interruption free execution even if Garbage collection is triggered. Though useful, scope memory management could also be tailored to manage the heaps and memory needed for the system (Gosling, 2000).

Further approaches and optimization methods for improving Garbage Collection has been researched by (Li, et al., 2018) which takes a deep dive upon two approaches in optimization including dynamic allocation of shadow regions to eliminate dependencies and ignorance of dense regions to reduce Garbage collection workload, which further concludes to a 18.2% improvement in performances. Other studies conducted by (Suo, et al., 2018) mentioned a different approach in optimization which involves in using

HotSpot JVM along with a enforcement on Garbage Collection thread for load balancing and purposing a more efficient work stealing algorithm.

All in all, upon all the studies that have conducted, it has clearly outlined the impact of performances that garbage collection could effect on the performance of a real time system, thus, those studies has implied on the optimization of designs to avoid such impact on real time system performances.

3.0 Methodology

Upon research on how performance is evaluated and optimized, the developer will be gone through a series of iterations, conducting different implementations and methods on constructing the Smart Watch Simulation. Each iteration will then be profiled, monitored, and analysed to seek out the best concurrent concepts that could be implemented for other development of real time systems. Attributes like garbage collection, CPU Usage, Memory Usage, Benchmarking Score and Thread Classes/ Objects will be taken into consideration upon the analysis of performances.

3.1 Overview of the Smart Watch Simulation

```
Sensor 1 : Checking Heart Rate
Sensor 2 : Checking Blood Oxygen
Sensor 1 : Normal Heart Rate Detected
Sensor 2 : Normal Blood Oxygen Detected
Sensor 1 : Heart Rate Check Completed
Sensor 2 : Blood Oxygen Check Completed
Sensor 1 : writing details to system
Sensor 1 : recorded Heart Rate = 59 to system
Sensor 2 : writing details to system
Notification : notifying details to user
```

Figure 3.1.1 Periodic Task of the Simulation

The Periodic Task of the simulation is for sensors to check the heart rate and blood oxygen levels, 2 sensors will be generated to do checking, each checking will take 7 seconds, upon the completion of the checking, each sensor will then write the details in the system in a text file. Afterwards, the notification thread will then notify the user.

```
Sensor 1 : Emergency Detected, Saving Details and Stop All Work
Notification : Danger Detected Notified User and Calling Ambulance
Notification : Average Heart Rate is 65.4
Notification : Average Blood Oxygen is 72.3
Notification : Total collected data is 19
```

Figure 3.1.2 Asynchronous Task of the Simulation

The Asynchronous Task of the simulation is when any sensor detects irregular heart rate or blood oxygen, the sensors will then stop its work, notification will prompt a call to the ambulance and performs calculation. Measurements and benchmarking will be performed on how quick the notification can provide the calculation during irregular event.

3.2 Design of the Simulation

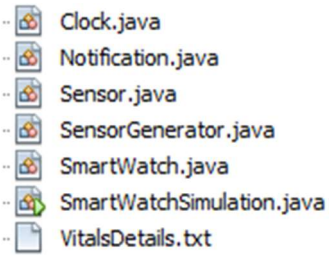

Figure 3.2.1 Object/ Classes of the Simulation

Iteration	Implementations
1	MultiThread + Semaphore
2	MultiThread + Lock
3	ExecutorService + Semaphore
4	ExecutorService + Lock
5	ExecutorService + Semaphore +Lock
6	TimerTask + Semaphore
7	TimerTask + Lock
8	TimerTask + ExecutorService + Lock
9	TimerTask + ExecutorService + Semaphore
10	TimerTask + ExecutorService + Lock + Semaphore
11	Structure Change + MultiThread + Lock
12	Structure Change + TimerTask + Lock
13	Structure Change + ExecutorService + Lock
14	Structure Change + TimerTask + ExecutorService + Lock
15	Structure Change + MultiThread + Semaphore
16	Structure Change + TimerTask (create clock and notification thread) + ExecutorService Semaphore

Table 3.2.2 Overview of Different Designs Developed by the Developer

The Table above shows the different approaches that the developer has developed to seek the best performing simulation. These implementations will be detailly discussed in the sections below.

1st Iteration

```
public void checkVitals(Sensor sensor) throws InterruptedException, IOException {
    while (!danger) {
        if (!HeartRateChecker) {
            HeartRateChecker = true;
            checkHeartRate(sensor);
            addDetails("Heart Rate", HeartRate);
            writeDetails(sensor, "Heart Rate", HeartRate);
            countRecord();

        } else if (!BloodOxygenChecker) {
            BloodOxygenChecker = true;
            checkBloodOxygen(sensor);
            addDetails("Blood Oxygen", BloodOxygen);
            writeDetails(sensor, "Blood Oxygen", BloodOxygen);
            countRecord();
        }

        if (HeartCheckFin || BloodCheckFin) {
            synchronized (notiWait) {
                notiWait.notify();
            }
        }

    }
    synchronized(notiWait){
        notiWait.notify();
    }
    System.out.println(sensor.sensorName + " : Emergency Detected, Saving Details

}
```

Figure 3.2.3 Sensors check vitals function.

Sensors will be generated using the sensor generator by implementing Runnable, clock thread is for forcing the system to trigger asynchronous task based on defined time in order to perform testing, notification will be a separate class that will be generated. Smart Watch class is a shared class that contains most of the functions performed by the threads. Details will be written in a text file named as VitalsDetails.txt.

Sensor Generator Class

```
public class SensorGenerator implements Runnable{

    SmartWatch display;
    int numSensor = 1;
    boolean dangerTime;

    public SensorGenerator(SmartWatch display){
        this.display = display;
    }

    @Override
    public void run() {
        while(numSensor != 3){
            Sensor sensor = new Sensor(display, numSensor);
            Thread thSensor = new Thread(sensor);
            thSensor.start();
            numSensor++;
        }
    }
```

Figure 3.2.4 Sensor Generator Code

The generator is consisted of a while loop for the creation of Sensor 1 and Sensor 2.

Smart Watch Class

```
public SmartWatch(){
    SensorGenerator senseGen = new SensorGenerator(this);
    Thread thClock = new Thread(clock);
    thClock.start();

    Thread thSenseGen = new Thread(senseGen);
    thSenseGen.start();

    Thread thNotification = new Thread(notification);
    thNotification.start();
}
```

Figure 3.2.5 Smart Watch Class

In the smart watch class, sensors get access over the check vitals function if there is no danger detected and will check whether the heart rate checker or blood oxygen checker is being accessed to prevent overlapped checking, checking results will then be counted and accumulated until the danger is detected. Upon completion of the task, the sensor will notify the notification thread.

```
Semaphore sem = new Semaphore(1);
Semaphore sem1 = new Semaphore(1);
Semaphore sem2 = new Semaphore(1);
```

Figure 3.2.6 Mutex Semaphore on 1st iteration

In the first iteration, synchronization or mutual exclusion is enabled using binary semaphore.

```
public void check(Notification notification) throws InterruptedException {
    while (!HeartRateChecker && !BloodOxygenChecker) {
        synchronized (notiWait) {
            notiWait.wait();
        }
    }

    if (HeartCheckFin && BloodCheckFin) {
        System.out.println(notification.notiName + " : notifying details to user");
        HeartRateChecker = false;
        BloodOxygenChecker = false;
        HeartCheckFin = false;
        BloodCheckFin = false;

    } else if (danger) {
        notification.dangerTime = true;
        notification.heartRateAvg = calculateAvg(totalHeart,countHeart);
        notification.bloodOxyAvg = calculateAvg(totalBlood,countBlood);
    } else {
        synchronized (notiWait) {
            notiWait.wait();
        }
    }
}
```

Figure 3.2.7 Notification check for error function

Similarly, the function is accessible in the smart watch class and is utilized by the notification thread. If no irregular details are checked, the notification will then notify the user about the details and then release the access of the checkers to the threads to continue its work. Furthermore, if danger is detected, the notification will then perform final calculations using the accumulated total data divided by total count to get the average and display it for the user.

```
public void  writeDetails(Sensor sensor, String dataType, int data) throws IOException, InterruptedException {
    lock.lock();
    System.out.println(sensor.sensorName + " : writing details to system");
    File file = new File("src/rts/version/pkg2/VitalsDetails.txt");
    FileWriter writer = null;
    writer = new FileWriter(file, true);
    PrintWriter printer = new PrintWriter(writer);
    printer.append(sensor.sensorName + " : " + dataType + " = " + data + "\n");
    printer.close();
    System.out.println(sensor.sensorName + " : recorded " + dataType + " = " + data + " to system");
    lock.unlock();
}
```

Figure 3.2.8 Writing details inside text file

The write details function is the synchronized function that will be mutually exclusive for the two threads, implementation of lock or semaphore is used for different iterations.

2nd iteration

```
Lock lock = new ReentrantLock();
Lock lock1 = new ReentrantLock();
Lock lock2 = new ReentrantLock();
```

Figure 3.2.9 Utilization of Lock

Second iteration utilizes lock for mutual exclusion as compared to first iteration.

3rd iteration

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

Figure 1 Cached Thread Pool

```
public SmartWatch() {
    SensorGenerator senseGen = new SensorGenerator(this);
    execute.execute(clock);
    execute.execute(senseGen);
    execute.execute(notification);

}
```

Figure 3.2.10 Executor Service

3rd iteration of the simulation uses semaphores like the first iteration, but

executor service is utilized in this version for executing threads.

4th iteration

```
Lock lock = new ReentrantLock();
Lock lock1 = new ReentrantLock();
Lock lock2 = new ReentrantLock();
```

Figure 3.2.11 Lock for Mutual Exclusion

```
public SmartWatch() {
    SensorGenerator senseGen = new SensorGenerator(this);
    execute.execute(clock);
    execute.execute(senseGen);
    execute.execute(notification);

}
```

Figure 3.2.12 Executor Service

4th iteration of the simulation changes the utilization of semaphore to lock.

5th iteration

```
Lock lock = new ReentrantLock();
Semaphore sem = new Semaphore(1);
Lock lock1 = new ReentrantLock();
```

Figure 3.2.13 Lock for Mutual Exclusion

5th iteration is like the previous iteration but mutex is enabled using both binary semaphore and lock.

6th iteration

```
class createThread extends TimerTask{
    int numSensorThread;
    createThread(int numSensor){
        this.numSensorThread = numSensor;
    }

    @Override
    public void run() {
        Sensor sensor = new Sensor(display, numSensorThread);
        sensor.run();
        t.cancel();
        t1.cancel();

    }

}

@Override
public void run() {

    t.schedule(new createThread(1), 0);
    t1.schedule(new createThread(2), 1000);

}
```

Figure 3.2.14 Timer Task utilization of Sensor Generator

In the 6th iteration, sensor generator is designed in a different approach by creating nested timer task for executing the 2 spawned threads.

```
public SmartWatch() {
    SensorGenerator senseGen = new SensorGenerator(this);
    t.schedule(clock,0);
    t1.schedule(senseGen,0);
    t2.schedule(notification,0);

}
```

Figure 3.2.15 Timer Task on Thread Generation

Other threads are also created and ran by using Timer Task, in this iteration, the simulation utilizes semaphore for mutex.

7th iteration

```
Lock lock = new ReentrantLock();
Lock lock1 = new ReentrantLock();
Lock lock2 = new ReentrantLock();
```

Figure 3.2.16 Lock for Mutex

7th iteration is like the 6th iteration, the only difference is the use of lock instead of semaphore.

8th iteration

```
public SmartWatch() {
    SensorGenerator senseGen = new SensorGenerator(this);
    execute.execute(clock);
    t.schedule(senseGen,0);
    execute.execute(notification);

}
```

Figure 3.2.17 Mixture of execution service and timer task for Thread generation

The 8th iteration uses a mix of executor service and timer task for generating the threads and uses lock for mutex.

9th iteration

```
Semaphore sem = new Semaphore(1);
Semaphore sem1 = new Semaphore(1);
Semaphore sem2 = new Semaphore(1);
```

Figure 3.2.18 Mutex using Semaphore

The design for the 9th simulation is like the 8th, only difference is mutex is enabled using binary semaphore.

10th iteration

```
Lock lock = new ReentrantLock();
Semaphore sem = new Semaphore(1);
Lock lock1 = new ReentrantLock();
```

Figure 3.2.19 Mutex using mixture of lock and semaphore

In the 10th iteration, the design is the same as 8th and 9th but the use of mutex will be consisted of binary semaphore and lock.

11th iteration

```
public Sensor(SmartWatch display, int sensorNum){
    this.sensorID = sensorNum;
    sensorName = "Sensor " + sensorNum;
    this.display = display;
}

public void checkVitals(Sensor sensor) throws IOException, InterruptedException{
    while(!danger){
        if (!display.HeartRateChecker) {
            display.HeartRateChecker = true;
            checkHeartRate(sensor);
            addDetails("Heart Rate", HeartRate);
            writeDetails(sensor, "Heart Rate", HeartRate);

        } else if (!display.BloodOxygenChecker) {
            display.BloodOxygenChecker = true;
            checkBloodOxygen(sensor);
            addDetails("Blood Oxygen", BloodOxygen);
            writeDetails(sensor, "Blood Oxygen", BloodOxygen);
        }

        if (display.HeartCheckFin || display.BloodCheckFin) {
            synchronized (display.notiWait) {
                display.notiWait.notify();
            }
        }
    }

    System.out.println(sensor.sensorName + " : Emergency Detected, Saving Details and
    synchronized (display.notiWait) {
        display.notiWait.notify();
    }

}
```

Figure 3.2.20 Amendment of design for each class

During the 11th iteration, the developer has changed the simulation with a different approach whereby all the functionality will be removed from Smart Watch class and allocate it to the respective classes that uses the functionality.

```
LinkedBlockingQueue<Integer> bloodOxygen = new LinkedBlockingQueue();
LinkedBlockingQueue<Integer> heartRate = new LinkedBlockingQueue();
```

Figure 3.2.21 Linked Blocking Queue for storing details

```
Iterator<Integer> listOfHeartRec = smartwatch.heartRate.iterator();
Iterator<Integer> listOfBloodRec = smartwatch.bloodOxygen.iterator();

while (listOfHeartRec.hasNext()){
    totalHeart = totalHeart + listOfHeartRec.next();
    countHeart+=1;
}
while (listOfBloodRec.hasNext()){
    totalBlood = totalBlood + listOfBloodRec.next();
    countBlood+=1;
}
bloodOxyAvg = calculateAvg(totalBlood,countBlood);
heartRateAvg = calculateAvg(totalHeart, countHeart);
totalRecord = countHeart + countBlood;
```

Figure 3.2.22 Iterator for performing calculation

Unlike the previous designs, the calculation will utilize the Linked Blocking Queue, each information will be pulled out and calculation of the average blood oxygen and heart rate will be performed.

```
public SmartWatch(){
    SensorGenerator senseGen = new SensorGenerator(this);
    Thread thClock = new Thread(clock);
    thClock.start();

    Thread thSenseGen = new Thread(senseGen);
    thSenseGen.start();

    Thread thNotification = new Thread(notification);
    thNotification.start();
}
```

Figure 3.2.23 Implementing Runnable for Thread generation

Normal thread creation by implementing runnable is used on this iteration along with mutexing using Lock.

12th iteration

```
public SmartWatch(){
    SensorGenerator senseGen = new SensorGenerator(this);
    t.scheduleAtFixedRate(clock,0,6*1000);
    t1.schedule(senseGen,0);
    t2.schedule(notification,0);
}
```

Figure 3.2.24 Timer Task

This iteration is like the previous iteration except the thread creation is utilizing Timer Task.

13th iteration

```
public SmartWatch(){
    SensorGenerator senseGen = new SensorGenerator(this);
    execute.execute(clock);
    execute.execute(senseGen);
    execute.execute(notification);
}
```

Figure 3.2.25 Executor Service

This iteration uses executor service for creation of threads.

14th iteration

```
public SmartWatch(){
    SensorGenerator senseGen = new SensorGenerator(this);
    execute.execute(clock);
    t.schedule(senseGen,0);
    execute.execute(notification);
}
```

Figure 3.2.26 Mixture of Executor Service and Timer Task for Thread Creation

15th iteration

```
Semaphore sem = new Semaphore(1);
Semaphore seml = new Semaphore(1);
```

Figure 3.2.27 Semaphore for mutex

The 15th iteration uses semaphore for mutex as compared to using lock on the 14th iteration.

16th iteration

```
public SmartWatch(){
    SensorGenerator senseGen = new SensorGenerator(this);
    t.schedule(clock,0);
    execute.execute(senseGen);
    tl.schedule(notification,0);
}
```

Figure 3.2.28 Mixture of Timer Task and Executor Service

```
Lock lock = new ReentrantLock();
Semaphore sem = new Semaphore(1);
```

Figure 3.2.29 Mutex using Lock and Semaphore

This iteration uses a mix of Timer Task and Executor Service for Thread creation. However, a slight shift in thread creation is used, there are no longer nested timer task for the sensor generator. Furthermore, lock and semaphore are both used for mutex.

3.3 Testing

Performance Evaluation and testing will be conducted to evaluate the performance of the simulation. Upon inspection of researches, 2 types of testing will be performed. Micro Benchmarking using Maven is also applicable, however, as the developer is using NetBeans as IDE, performance profiling will be the first test that will be conducted to monitor CPU, etc. Furthermore, as mentioned above, the speed of calculation for average vitals will be analyzed.

Performance profiling is an embedded feature that is located inside NetBeans, upon profiling, the developer will consider the overall usage of heap/memory, CPU usage and garbage collection as mentioned in the researches overutilization of heap will cause garbage collection which will result to a halt in the real-time system. Each iteration will be executed for 2 minutes for fairness. Furthermore, the calculation time of the average vitals will be performed in the end when notification detects irregular vitals and call the ambulance, this will test the response time of the simulation, it is an important aspect of a real-time system as mentioned by researches conducted in the section above. A summary will be stated as follows:

Test 1: Performance profiling evaluating overall CPU Usage, heap/ memory usage and garbage collection in 2 minutes of execution on the simulation. Multiple execution will be conducted to calculate the average usage.

Test 2: Overall calculation speed when notification thread detects irregular vitals upon 1 minutes of execution on the simulation. 3 executions will be conducted, and an average will be calculated.

4.0 Results and Discussion

Following Section will be conduction of test along with the discussion of the results.

4.1 Conduction of Test 1

Iteration	Implementations
1	MultiThread + Semaphore
2	MultiThread + Lock
3	ExecutorService + Semaphore
4	ExecutorService + Lock
5	ExecutorService + Semaphore +Lock
6	TimerTask + Semaphore
7	TimerTask + Lock
8	TimerTask + ExecutorService + Lock
9	TimerTask + ExecutorService + Semaphore
10	TimerTask + ExecutorService + Lock + Semaphore
11	Structure Change + MultiThread + Lock
12	Structure Change + TimerTask + Lock
13	Structure Change + ExecutorService + Lock
14	Structure Change + TimerTask + ExecutorService + Lock
15	Structure Change + MultiThread + Semaphore
16	Structure Change + TimerTask (create clock and notification thread) + ExecutorService Semaphore

Table 4.1.1 Different implementation in iterations

Overall CPU Usage

Iteration	CPU (%)
1	0.2
2	0.2
3	0.4
4	0.2
5	0.2
6	9.2
7	8.9
8	7.6
9	8.2
10	6.3
11	0.4
12	10.9
13	0.1
14	10.8
15	0.3
16	0.2

Table 4.1.2 Average CPU Usage

Average Memory Usage (Byte)

Iteration	Memory
1	22,461,912
2	22,462,024
3	22,463,040
4	22,462,656
5	22,461,960
6	32,178,172
7	33,859,424
8	33,837,824
9	30,035,136
10	30,480,504
11	22,462,160
12	32,525,984
13	22,462,152
14	25,144,088
15	22,462,192
16	22,462,184

Table 4.1.3 Average Memory Usage

Garbage Collection

Iteration	Garbage Collection
All	**None**

Table 4.1.4 Garbage Collection on Simulation

Iteration 1:

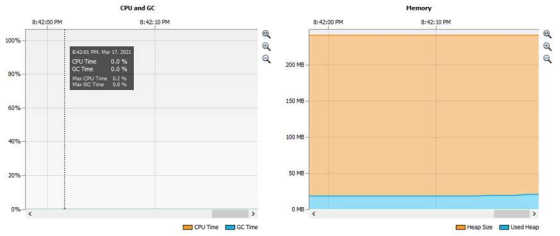

Figure 4.1.5 CPU Usage and Memory Usage on Iteration

Iteration 2:

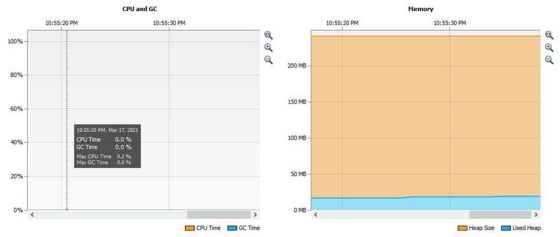

Figure 4.1.6 CPU Usage and Memory Usage on Iteration

Iteration 3:

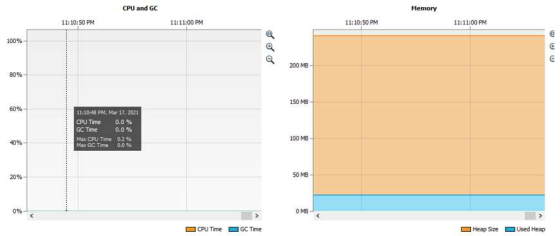

Figure 4.1.7 CPU Usage and Memory Usage on Iteration

Iteration 4:

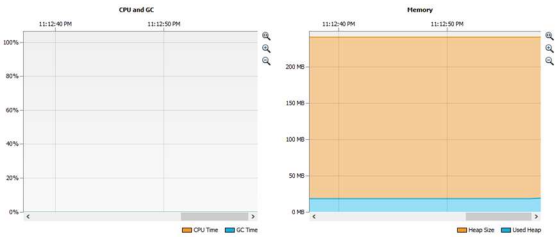

Figure 4.1.8 CPU Usage and Memory Usage on Iteration

Iteration 5:

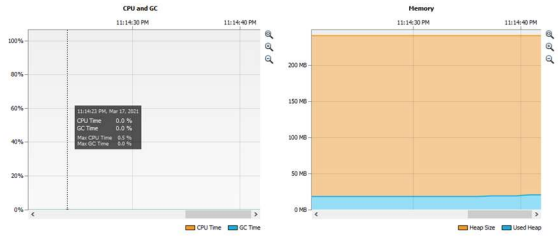

Figure 4.1.9 CPU Usage and Memory Usage on Iteration

Iteration 6:

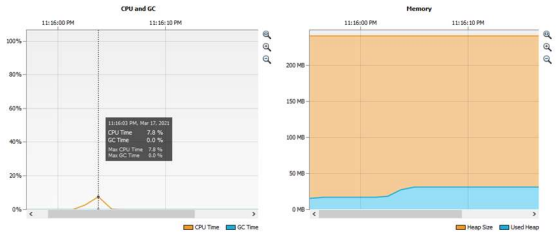

Figure 4.1.10 CPU Usage and Memory Usage on Iteration

Iteration 7:

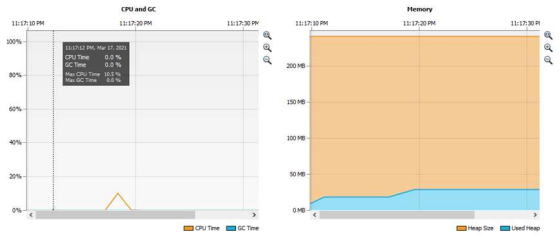

Figure 4.1.11 CPU Usage and Memory Usage on Iteration

Iteration 8:

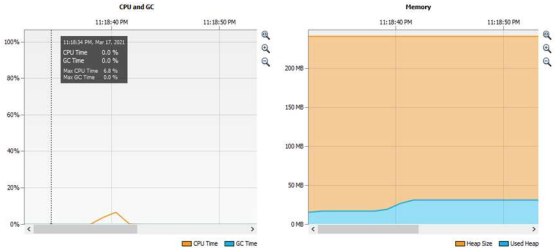

Figure 4.1.12 CPU Usage and Memory Usage on Iteration

Iteration 9:

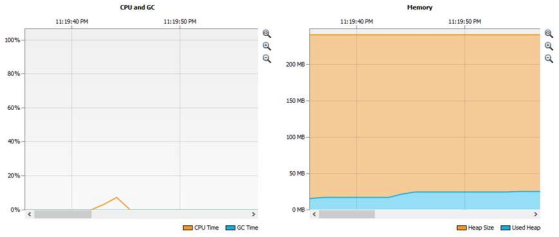

Figure 4.1.13 CPU Usage and Memory Usage on Iteration

Iteration 10:

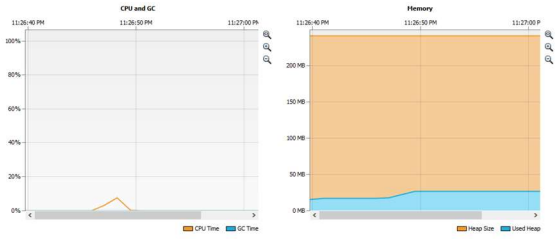

Figure 4.1.14 CPU Usage and Memory Usage on Iteration

Iteration 11:

Figure 4.1.15 CPU Usage and Memory Usage on Iteration

Iteration 12:

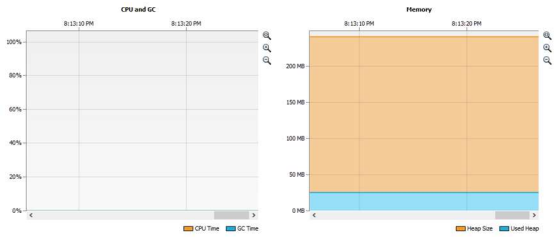

Figure 4.1.16 CPU Usage and Memory Usage on Iteration

Iteration 13:

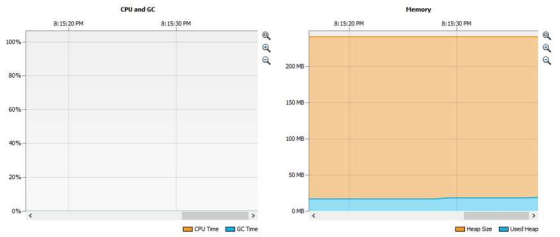

Figure 4.1.17 CPU Usage and Memory Usage on Iteration

Iteration 14:

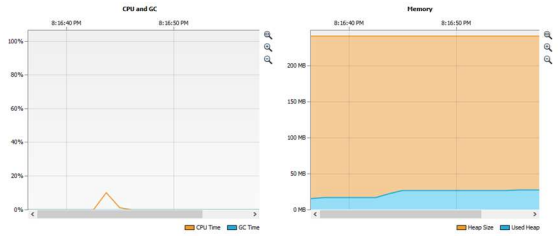

Figure 4.1.18 CPU Usage and Memory Usage on Iteration

Iteration 15:

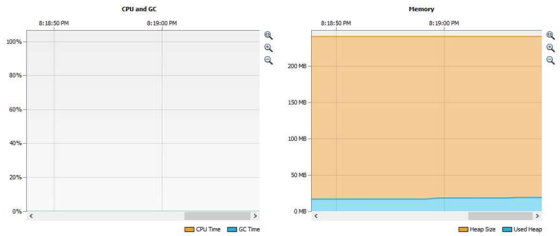

Figure 4.1.19 CPU Usage and Memory Usage on Iteration

Iteration 16:

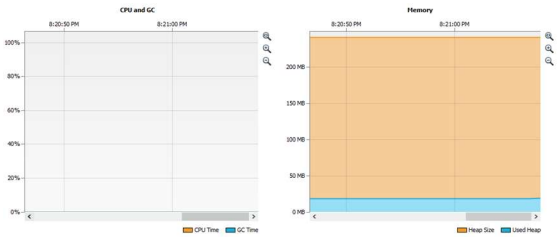

Figure 4.1.20 CPU Usage and Memory Usage on Iteration

4.2 Analysis of Result for Test 1

During each iteration, the developer has designed the simulation well enough to avoid garbage collection. Thus, it is found that normal creation of threads had similar performance with thread creation using executor service. However, upon further inspection, the creation of thread uses lesser resources during the start and slowly accumulates over time while executor service provided a stable result in terms of CPU Usage and Memory consumption consuming around 0.2% of CPU and around 22,000,000 byte of memory even if the structure of the program has been altered slightly.

On the other hand, Any implementation of Timer Task consumes a large amount of CPU Usage and Memory Usage which resulted over 9 percent of CPU usage and over 30,000,000 in memory usage, this might be due to the developers design of constructing a nested Timer Task, this assumption is made as one of the iterations has no implementation of nested Timer Task which resulted in lesser consumption in memory and CPU. Hence, nested Timer

Task is not suggested to be implemented for design as it will consume more CPU and memory. Other than that, it is found that Timer Task works better with semaphore as compared to lock, providing a slightly lesser result in memory consumption.

In addition to that, it is found that implementation of binary semaphore or lock hasn't affected much in terms of performance or memory consumption. Upon further analysis of performance in each iteration, it is identified that **iteration 4** is the best performer, because the average CPU usage is low, and performance is the most stable with no fluctuation in memory and CPU usage. Iteration 13 on the other hand though performed slightly better in CPU performance but it performs poorly on response time which will be presented in Test 2.

4.3 Conduction of Test 2

```
long start = System.nanoTime();
notification.dangerTime = true;
notification.heartRateAvg = calculateAvg(totalHeart,countHeart);
notification.bloodOxyAvg = calculateAvg(totalBlood,countBlood);
long end = System.nanoTime();
notification.performance = end - start;
```

Figure 4.3.1 Response Time (Calculation Time) of Asynchronous Task

```
start = System.nanoTime();
while (listOfHeartRec.hasNext()){
    totalHeart = totalHeart + listOfHeartRec.next();
    countHeart+=1;
}
while (listOfBloodRec.hasNext()){
    totalBlood = totalBlood + listOfBloodRec.next();
    countBlood+=1;
}
bloodOxyAvg = calculateAvg(totalBlood,countBlood);
heartRateAvg = calculateAvg(totalHeart, countHeart);
totalRecord = countHeart + countBlood;
end = System.nanoTime();
performance = end - start;
System.out.println(notiName + " : Danger Detected Notified User and Calling Ambulance");
System.out.println(notiName + " : Average Heart Rate is " + heartRateAvg);
System.out.println(notiName + " : Average Blood Oxygen is " + bloodOxyAvg);
System.out.println(notiName +" : Total collected data is "+ totalRecord);
System.out.println(notiName +" : Response Time is " + performance);
```

Figure 4.3.2 Response Time (Calculation Time) of Asynchronous Task for 2nd structure

```
Notification : Response Time is 30800
```

Figure 2 Sample Result of Response Time

Each Iteration will be executed 3 times, the results will be tabulated, and an average result will be presented.

Iteration 1:

Execution	Results (nano sec)
1	30,800
2	28,500
3	9,200
Average:	22,833

Table 4.3.3 Result of Iteration 1

Iteration 2:

Execution	Results (nano sec)
1	32,100
2	10,200
3	39,200
Average:	27,166

Table 4.3.4 Result of Iteration 2

Iteration 3:

Execution	Results (nano sec)
1	27,700
2	31,500
3	41,700
Average:	33,633

Table 4.3.5 Result of Iteration 3

Iteration 4:

Execution	Results (nano sec)
1	14,500
2	8,200
3	8,200
Average:	10,300

Table 4.3.6 Result of Iteration 4

Iteration 5:

Execution	Results (nano sec)
1	9,900
2	29,900
3	10,200
Average:	16,667

Table 4.3.7 Result of Iteration 5

Iteration 6:

Execution	Results (nano sec)
1	28,400
2	9,600
3	29,600
Average:	22,533

Table 4.3.8 Result of Iteration 6

Iteration 7:

Execution	Results (nano sec)
1	16,700
2	18,800
3	21,500
Average:	19,000

Table 4.3.9 Result of Iteration 7

Iteration 8:

Execution	Results (nano sec)
1	26,800
2	9,900
3	63,500
Average:	33,400

Table 4.3.10 Result of Iteration 8

Iteration 9:

Execution	Results (nano sec)
1	46,000
2	32,500
3	44,700
Average:	41,067

Table 4.3.11 Result of Iteration 9

Iteration 10:

Execution	Results (nano sec)
1	25,200
2	36,100
3	29,700
Average:	30,333

Table 4.3.12 Result of Iteration 10

Iteration 11:

Execution	Results (nano sec)
1	300,100
2	261,500
3	89,800
Average:	217,133

Table 4.3.13 Result of Iteration 11

Iteration 12:

Execution	Results (nano sec)
1	128,900
2	308,400
3	249,300
Average:	228,867

Table 4.3.14 Result of Iteration 12

Iteration 13:

Execution	Results (nano sec)
1	295,800
2	203,600
3	158,400
Average:	219,266

Table 4.3.15 Result of Iteration 13

Iteration 14:

Execution	Results (nano sec)
1	250,400
2	380,200
3	151,900
Average:	260,833

Table 4.3.16 Result of Iteration 14

Iteration 15:

Execution	Results (nano sec)
1	120,100
2	223,400
3	215,000
Average:	181,167

Table 4.3.17 Result of Iteration 15

Iteration 16:

Execution	Results (nano sec)
1	370,500
2	414,100
3	180,300
Average:	321,633

Table 4.3.18 Result of Iteration 16

Comparison of Results

Iteration	Implementations
1	22,833
2	27,166
3	33,633
4	10,300
5	16,667
6	22,533
7	19,000
8	33,400
9	41,067
10	30,333
11	217,133
12	228,867
13	219,266
14	260,833
15	181,167
16	321,633

Table 4.3.19 Comparison of Results for Response Time

4.4 Analysis of Results for Test 2

Upon Analysis of the result, again iteration 4 performed the best amongst other iterations. Thus, it is found that the structure change has significantly affected the performance of the design in terms of response time. The cause of this is due to the concentrated efforts on performing traversal and calculation using while loop along with Linked Block Queue. Hence, it is suggested that the calculation should be performed incrementally to increase the performance and reliability of response time for the system.

Furthermore, the performance of each iteration varies more differently as compared to CPU and memory usage. However, as mentioned, response time is one of the most crucial criteria for a real time system. Some executions in certain iterations might perform better in certain scenarios, however, the selection of response time should take into consideration the consistency of the performance. Hence, iteration 4 is selected as the best design amongst the iterations.

5.0 Conclusion

Overall, upon completion of studies and test conducted by the developer, it has been identified that reliability, safety, response time, garbage collection, memory management and CPU usage test should be conducted to examine the impact of design on the performance of the simulation. Furthermore, upon testing it is identified that executor service with lock is the best combination for design and to add on, response time on calculation will be significantly affected if the calculation is performed at the asynchronous event as compared to accumulative calculation performed by each threads. There are other factors that could be taken into consideration for performance. However, the performance goal should be identified and planned by the developer but overall, the crucial elements does not hinder much from the attributes that has been selected. Nevertheless, the research conducted has proven that performance will be impacted by the design of the system so, planning for design should be examined before development.

6.0 References

Agesen, O., Detlefs, D. & Moss, E., 1998. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. *ACM SIGPLAN Notices,* 33(5), pp. 1-19.

Bench Partner, 2019. *Explain all Types of Task Classes in Real Time System.* [Online]
Available at: https://benchpartner.com/explain-all-types-of-task-classes-in-real-time-system/
[Accessed 9 March 2021].

Chen, J., Du, C., Han, P. & Zhang, Y., 2019. Sensitivity Analysis of Strictly Periodic Tasks in Multi-Core Real-Time Systems. *IEEE Access,* Volume 7, pp. 135005-135022.

Claypool, M., 2005. The effect of latency on user performance in Real-Time Strategy games. *Computer Networks,* 49(1), pp. 52-70.

Doug, A., 2018. Linux and real-time. *Linux for Embedded and Real-Time Applications (Fourth Edition),* pp. 257-270.

Dynatrace, 2020. *The impact of Garbage Collection on Application Performance.* [Online]
Available at: https://www.dynatrace.com/resources/ebooks/javabook/impact-of-garbage-collection-on-performance/#:~:text=The%20impact%20of%20Garbage%20Collection%20on%20

Application%20Performance,-Chapter%3A%20Memory%20Management&text=An%20application%20that%20spends%201,
[Accessed 8 March 2021].

Electronics Hub, 2015. *Real Time Operating System (RTOS).* [Online]
Available at:
https://www.electronicshub.org/real-time-operating-system-rtos/#:~:text=In%20soft%20real%20time%20system,or%20can%20miss%20the%20deadline.
[Accessed 18 March 2021].

GeeksforGeeks, 2019. *Lifecycle and States of a Thread in java.* [Online]
Available at:
https://www.geeksforgeeks.org/lifecycle-and-states-of-a-thread-in-java/
[Accessed 8 March 2021].

GeeksforGeeks, 2019. *Tasks in Real Time Systems.* [Online]
Available at:
https://www.geeksforgeeks.org/tasks-in-real-time-systems/
[Accessed 8 March 2021].

Gosling, J., 2000. The Real-Time Specification for Java. *IEEE Xplore,* 33(6), pp. 47-54.

Gupta, A., Tucker, A. & Urushibara, S., 1991. The impact of operating system scheduling policies and synchronization methods of performance of parallel application. *SIGMETRICS 91,* pp. 120-132.

Halang, W. A., Gumzej, R. & Druzovec, M., 2000. Measuring the Performance of Real-Time Systems. *The International Journal of Time-Critical Computing Systems,* Volume 18, pp. 59-68.

Heide, J. A. & Halang, W. A., 1991. Performance Metrics for Real-Time Systems. *Informatik Fachberichte Book Series,* Volume 295, pp. 121-122.

Huang, X. et al., 2004. The garbage collection advantage: improving program locally. *ACM SIGPLAN Notices,* 39(10), pp. 1-12.

javaTpoint, 2020. *Life cycle of a Thread (Thread States).* [Online]
Available at:
https://www.javatpoint.com/life-cycle-of-a-thread
[Accessed 8 March 2021].

Kay, J., 2020. *Introduction to Real-time Systems.* [Online]
Available at:
https://design.ros2.org/articles/realtime_background.html
[Accessed 15 March 2021].

Keycdn, 2018. *What is Latency and how to Reduce it.* [Online]
Available at:
https://www.keycdn.com/support/what-is-latency
[Accessed 7 March 2021].

Li, H., Wu, M. & Chen, H., 2018. *Analysis and Optimization of Java Full Garbage Collection.* New York, Association for Computing Machinery.

Mc Graw Hill, 2017. *Real Time Systems.* [Online]
Available at:
http://www.mhhe.com/engcs/compsci/pressman/information/olc/RTdesign.html
[Accessed 16 March 2021].

Oracle, 2008. *An Introduction to Real-Time Java Technology: Part 1, The Real-Time Specification for Java.* [Online]
Available at:
https://www.oracle.com/technical-resources/articles/javase/jsr-1.html
[Accessed 7 March 2021].

Pankaj, 2020. *Thread Life Cycle in Java - Thread States in Java.* [Online]
Available at: https://www.journaldev.com/1044/thread-life-cycle-in-java-thread-states-in-java
[Accessed 8 March 2021].

Schoeberl, M., 2006. *Real-time garbage collection for Java.* Gyrongju, IEEE Explore.

Sharma, M., Emiligi, H. & Gebali, F., 2015. Performance Evaluation of Real-Time System. *International Journal of Computing and Digital Systems,* 4(1), pp. 43-53.

Stichnoth, J. M., Lueh, G.-Y. & Cierniak, M., 1999. Support for garbage collection at every instruction in a Java compiler. *ACM SIGPLAN Notices,* 34(5), pp. 20-24.

Suo, K., Rao, J., Jiang, H. & Srisa-an, W., 2018. Characterizing and optimizing hotspot parallel garbage collection on multicore systems. *Euro Sys'18,* Volume 35, pp. 1-15.

VIVADIFFERENCES, 2020. *10 Difference Between Hard Real Time System And Soft Real Time System.* [Online]
Available at: https://vivadifferences.com/10-difference-between-hard-real-time-system-and-soft-real-time-system/
[Accessed 7 March 2021].

Welc, A., Hosking, A. L. & Jagannathan, S., 2004. *Preemption-Based Avoidance of Prioirty Inversion for Java.* Quebec, IEEE Computer Society.

Wolf, M., 2014. Multiprocessor Architectures. *High-Performance Embedded Computing,* pp. 243-299.

Zambrano, O. M. et al., 2017. Community Early Warning System. *Wireless Public Safety Networks,* Volume 3, pp. 39-66.