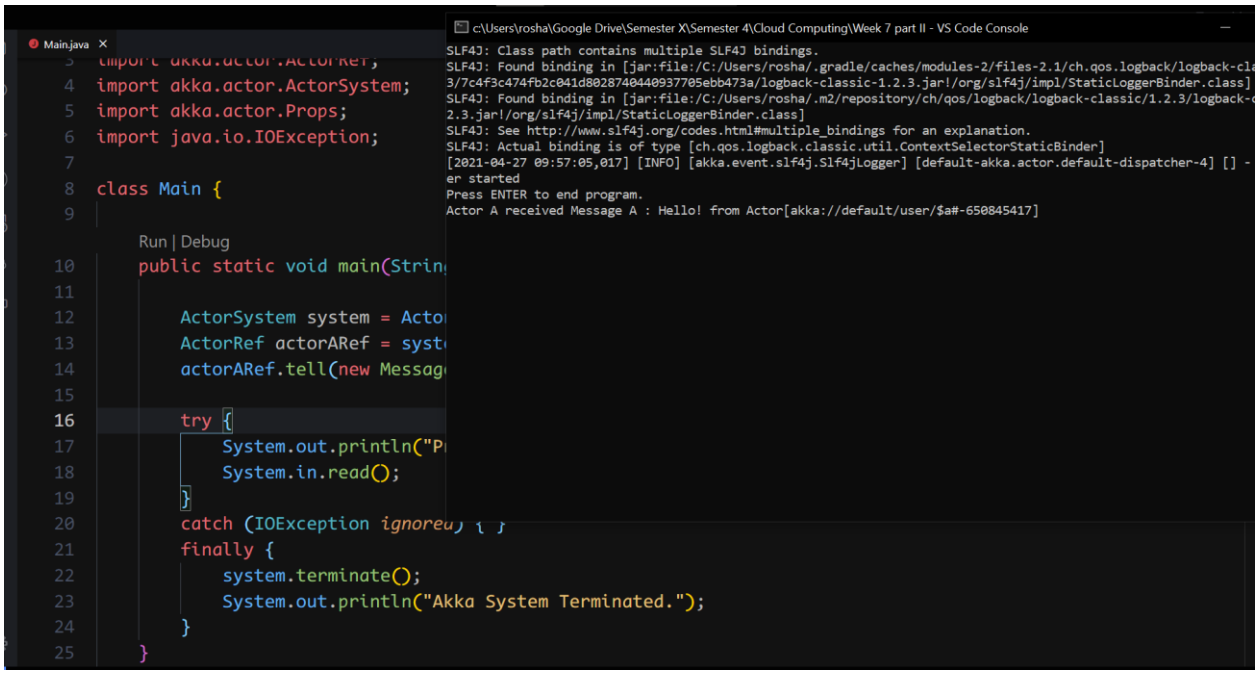
Module	Assessment Type
Distributed and Cloud Systems Programming	Individual Report

Workshop 7 ii

Student Id : 2049867
Student Name : Roshan Parajuli
Section : L5CG3
Module Leader : Rupak Koirala
Lecturer /Tutor : Saroj Sharma
Submitted on : 2021-05-14

Part 1

The sample project was downloaded and imported into the IDE. It was then built and ran. The output received was the output expected.



Message A is received by Actor A and the provided message was printed into the console.

Part 2

Q. No.1

```
LoggingAdapter log = Logging.getLogger(getContext().system(), this);

@Override
public Receive createReceive() {
    return receiveBuilder().match(MessageA.class, this::onMessageA)
            .matchAny(msg -> log.info("Actor A has received an unknown message {}", msg)).build();
}
```

Since the Actor class Actor A received responds to only one message object i.e. MessageA due to its matching with it, a new method is added in the receive builder to match any of the messages and print the specified message.

```
class NewMsg {
    boolean value;

    NewMsg(boolean value) {
        this.value = value;
    }
}
```

Now, a new message object is created namely NewMsg for Boolean messages. Its constructor requires a Boolean value. Now, we are able to send Boolean messages to actor A with the help of its reference.

```
Run | Debug
public static void main(String[] args) {

    ActorSystem system = ActorSystem.create();
    ActorRef actorARef = system.actorOf(Props.create(ActorA.class));
    actorARef.tell(new MessageA("Hello!"), actorARef);
    actorARef.tell(new NewMsg(true), actorARef);
```

The output is shown in the screenshot below. The unknown message output is printed on the console.

```
c:\Users\rosha\Google Drive\Semester X\Semester 4\Cloud Computing\Week 7 part II - VS Code Console
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/C:/Users/rosha/.gradle/caches/modules-2/files-2.1/ch.qos.logback/logback-classic/1.2.
3/7c4f3c474fb2c041d8028740440937705ebb473a/logback-classic-1.2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/C:/Users/rosha/.m2/repository/ch/qos/logback/logback-classic/1.2.3/logback-classic-1.
2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]
[2021-04-27 10:38:16,270] [INFO] [akka.event.slf4j.Slf4jLogger] [default-akka.actor.default-dispatcher-4] [] - Slf4jLogg
er started
Press ENTER to end program.
Actor A received Message A : Hello! from Actor[akka://default/user/$a#-1334624308]
[2021-04-27 10:38:16,398] [INFO] [com.example.ActorA] [default-akka.actor.default-dispatcher-4] [akka://default/user/$a]
 - Actor A has received an unknown message com.example.NewMsg@3115f912
```

Q. No. 2.

Since, all of the messages sent until now were a custom java object, it is not possible to create new java classes for every type of messages. The program is modified such that it can respond to any other messages. Simply sending the message with String("…") or Integer(int) would work just fine if there are the methods to listen for it.

```
ActorSystem system = ActorSystem.create();
ActorRef actorARef = system.actorOf(Props.create(ActorA.class));
actorARef.tell(new MessageA("Hello!"), actorARef);
actorARef.tell(new String("Hello from wrapper class!"), actorARef);
// actorARef.tell(new NewMsg(true), actorARef);
```

```java
public class ActorA extends AbstractActor {
    LoggingAdapter log = Logging.getLogger(getContext().system(), this);

    @Override
    public Receive createReceive() {
        return receiveBuilder().match(MessageA.class, this::onMessageA)
                .match(String.class, this::onString)
                .build();
    }

    private void onMessageA(MessageA msg) {
        System.out.println("Actor A received Message A : " + msg.text + " from " + getSender());
    }

    private void onString(String msg) {
        System.out.println("Actor A received Message A : " + msg + " from " + getSender());
    }
}
```

On sending a string message, the onString method will be called and the specified message will be printed on the screen.

The output can be seen below:

```
c:\Users\rosha\Google Drive\Semester X\Semester 4\Cloud Computing\Week 7 part II - VS Code Console
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/C:/Users/rosha/.gradle/caches/modules-2/files-2.1/ch.qos.logback/logback-classic/1.2.
3/7c4f3c474fb2c041d8028740440937705ebb473a/logback-classic-1.2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/C:/Users/rosha/.m2/repository/ch/qos/logback/logback-classic/1.2.3/logback-classic-1.
2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]
[2021-04-27 10:55:29,936] [INFO] [akka.event.slf4j.Slf4jLogger] [default-akka.actor.default-dispatcher-4] [] - Slf4jLogg
er started
Press ENTER to end program.
Actor A received Message A : Hello! from Actor[akka://default/user/$a#79662704]
Actor A received Message A : Hello from wrapper class! from Actor[akka://default/user/$a#79662704]
```

Q. No. 3

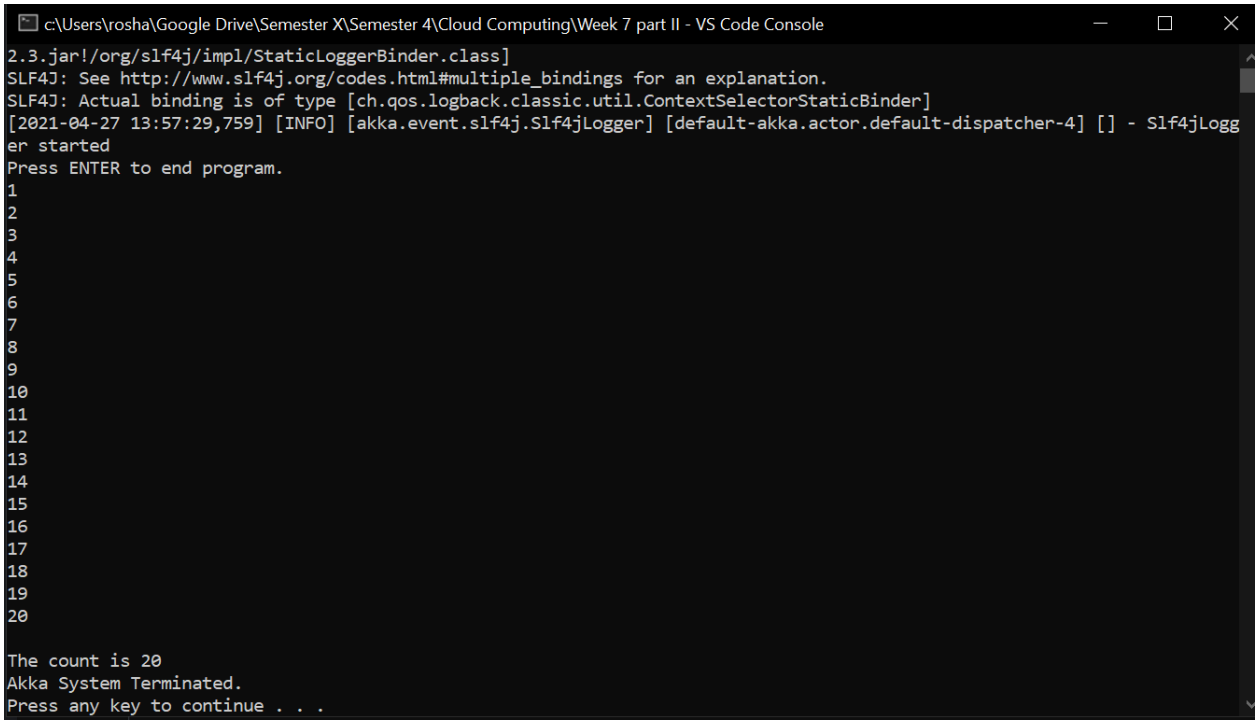
A new Counter class is created to show that the critical section is not an issue with the Akka actors.

```java
public class Counter extends AbstractActor {

    private int count;

    Counter() {
        count = 0;
    }

    @Override
    public Receive createReceive() {
        return receiveBuilder().match(MessageA.class, this::onMessageA).match(String.class, this::onSt
    }

    private void onMessageA(MessageA msg) {
        count++;
        System.out.println(count);
    }

    private void onString(String msg) {
        System.out.println("The count is " + count);
    }

}
```

The constructor is initialized with a value of zero and upon receiving the message A by the counter class, its value is increased by one.

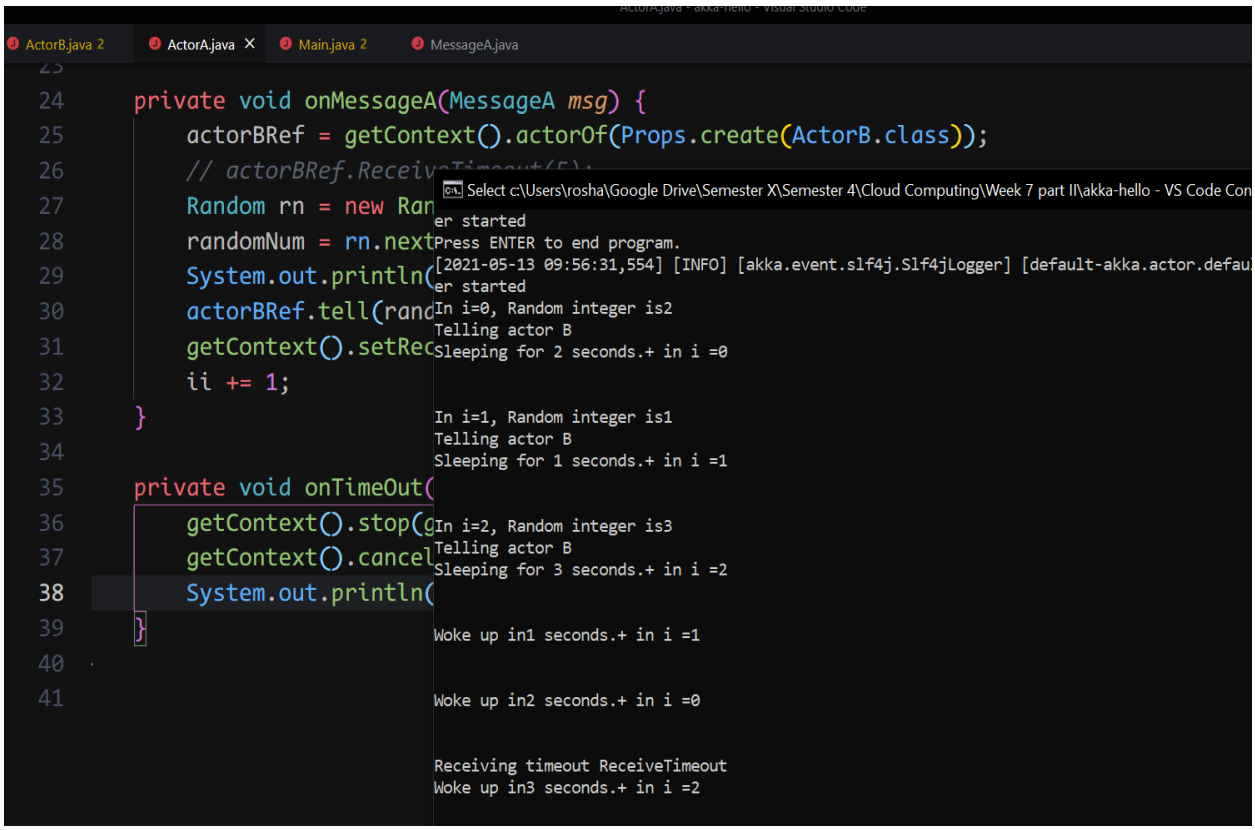
```
ArrayList<ActorRef> actorARef = new ArrayList<ActorRef>();

for (int i = 0; i < 20; i++) {
    actorARef.add(system.actorOf(Props.create(ActorA.class)));
    actorARef.get(i).tell(new MessageA("Hello!"), counterRef);
}
```

20 instances of actor A is created and the value of the counter variable id changed by each instances of the actor A.



It can be observed that the value of counter is exactly 20 which proves that even if the resource is shared, the random change in the value of global counter is not an issue in the akka actor. Akka actor eliminates the problem of mutexes and critical sections.

Q. No. 4.



In the task, two actors are created Actor A and Actor B. A random number is generated in a loop for a 100 times which is between 1 and 5 in the main method of the program. Everytime the message is sent to actor B by actor A.

```
public static void main(String[] args) {

    ActorSystem system = ActorSystem.create();
    ActorRef actorARef = system.actorOf(Props.create(ActorA.class));

    for (int i = 0; i < 100; i++) {
        actorARef.tell(new MessageA(1), actorARef);
    }

    try {
        System.out.println("Press ENTER to end program.");
        System.in.read();
    } catch (IOException ignored) {
    } finally {
        system.terminate();
        System.out.println("Akka System Terminated.");
    }
}
```

The main method sends a number “1” to the actor A to initiate the program.

On receiving the initiation,

```
private void onMessageA(MessageA msg) {
    actorBRef = getContext().actorOf(Props.create(ActorB.class));
    // actorBRef.ReceiveTimeout(5);
    Random rn = new Random();
    randomNum = rn.nextInt(5) + 1;
    System.out.println("In i=" + ii + ", Random integer is" + randomNum + "\nTelling actor B");
    actorBRef.tell(randomNum + " " + ii, getSelf());
    getContext().setReceiveTimeout(Duration.create("2 seconds"));
    ii += 1;
}
```

Actor A immediately starts the set receive timeout for 2 seconds.

After a specific interval of 2 seconds is passed,

```
private void onTimeOut(ReceiveTimeout rt) {
    getContext().stop(getSelf());
    getContext().cancelReceiveTimeout();
    System.out.println("Receiving timeout " + rt);
}
```

ActorB is sent a stop message and the timeout is cancelled.

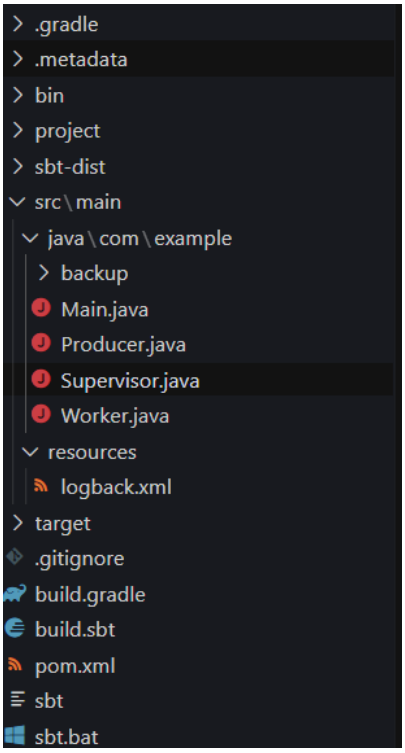
Meanwhile in actor B,

```
@Override
public Receive createReceive() {
    return receiveBuilder().match(String.class, this::onMessageA).build();
}

private void onMessageA(String msg) throws InterruptedException {
    System.out.println("Sleeping for " + msg.split(" ")[0] + " seconds.+ in i =" + msg.split(" ")[
    Thread.sleep(Integer.parseInt(msg.split(" ")[0]) * 1000); // converting to milliseconds
    System.out.println("Woke up in" + msg.split(" ")[0] + " seconds.+ in i =" + msg.split(" ")[1]
}
```

Upon receiving the message A, the string is splitted and the thread goes to sleep for the particular time frame (if the random number generated was 5, thread goes to sleep for 5 seconds).

Assessed Task:

1. Three actors are created i.e. Producer, Supervisor and Worker along with the main class to invoke the first actor.



2. Main method in the main class calls the producer by sending a string message.

```
class Main {

    public static void main(String[] args) {

        ActorSystem system = ActorSystem.create();
        ActorRef ProdRef = system.actorOf(Props.create(Producer.class));
        ProdRef.tell(new String("Start"), ProdRef);
    }
}
```

3. After the producer gets the string message, it is mapped to the onString function in the createReceive function.

```
@Override
public Receive createReceive() {
    return receiveBuilder().match(String.class, this::onString)
                           .match(Integer.class, this::getPrimes).build();
}
```

4. The onString method generates a 1000 random numbers by calling the numGenerate() method and sends everyone of them to the supervisor using its reference.

```
private void onString(String s) {
    System.out.println("\n\nProducer: Starting...");
    System.out.println("Producer: Generating random integers...");
    System.out.println(
            "Producer: Message is being conveyed to Supervisor... \n\t  Supervisor will now do its

    for (int i = 0; i < 1000; i++) {
        numGenerate();
    }
}

private void numGenerate() {
    Random rn = new Random();
    int randomNum = rn.nextInt(100000) + 10000;

    supRef.tell(randomNum, getSelf());
}
```

5. At supervisor class, 10 worker classes are being created in loop and added to the list of routes. After adding all 10 worker classes a new Router class in initialized which distributed the work in a round robin routing logic.

```
List<Routee> routees = new ArrayList<Routee>();
for (int i = 0; i < 10; i++) {
ActorRef workRef = getContext().actorOf(Props.create(Worker.class));
getContext().watch(workRef);
routees.add(new ActorRefRoutee(workRef));
}
router = new Router(new RoundRobinRoutingLogic(), routees);
}
```

6. Upon receiving the integer sent by the producer, it uses API forward() to forward the message to all the worker actors.

```
@Override
public Receive createReceive() {
    return receiveBuilder().match(Integer.class, this::onProducerInt).build();
}

private void onProducerInt(Integer i) {
    System.out.println("Producer sent a number" + i);
    workRef.forward(i, getContext());
}
```

7. When supervisor sends a number (integer) to the worker class, the onSuperVisorInt function calls isPrime function to check if the number is indeed a prime number.

```java
@Override
public Receive createReceive() {
    return receiveBuilder().match(Integer.class, this::onSupervisorInt).build();
}

private void onSupervisorInt(Integer num) {
    if (isPrime(num)) {
        prodRef.tell(num, getSelf());
    }
}
```

```java
  private boolean isPrime(Integer num) {

      int i, m = 0, flag = 0;
      m = num / 2;

      for (i = 2; i <= m; i++) {
          if (num % i == 0) {
              flag = 1;
          }
      }

      if (flag == 0) {
          return true; // is prime
      } else {
          return false;
      }
  }
```

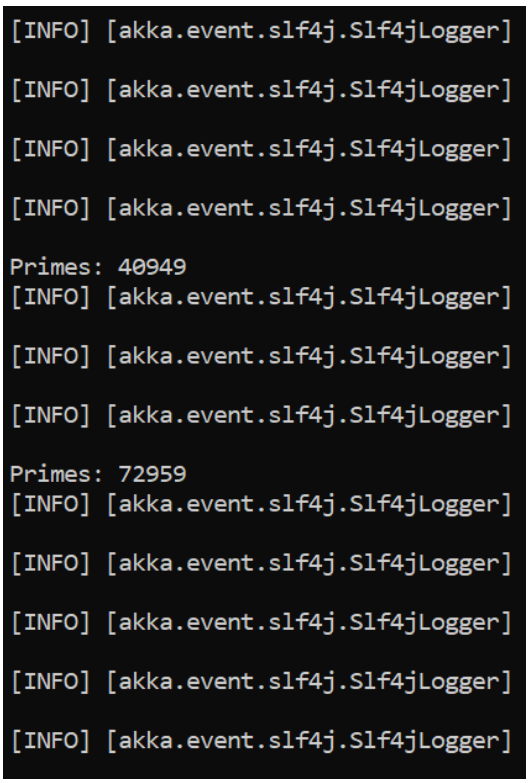
8. If the number is prime it is sent to the producer class. The getPrimes method is then called to print out the number.

```java
private void getPrimes(Integer prime_num) {
    System.out.println("Primes: " + prime_num);
}
```

9. After all that is done, the producer class terminates the whole actor system.

```java
getContext().getSystem().terminate();
```

10. The prime number is printed between the info messages like such:



As it can be seen that the numbers printed are indeed the prime numbers and it was producers duty to print out on the screen.