Home assignment 1, Simulation

Joel Bångdal
Viktor Claesson

Kurs
EITN95
Lund Universitet

April 29, 2020

1 Task 1

With Q1 having a constant arrival time of the value in the first column in Table 1 and a serving time of Exp(1/2.1) and Q2 arrivals from Q1 and a constant serving time of 2 and with measurements Exp(1/5) apart, we got the Table 1 as results by letting the simulation run for 10,000 measurements per Arrival time for Q1.

Q1 arrival time	Mean Q2 size	Chance to be rejected
0.50	10.30	76.13%
1.00	13.06	52.44%
1.50	8.40	28.86%
2.00	4.40	7.42%
2.50	1.68	0.49%
3.00	0.96	0.01%
3.50	0.73	0.00%
4.00	0.58	0.00%
4.50	0.49	0.00%
5.00	0.43	0.00%

Table 1: Results from simulating Task 1, 10000 measurements were taken and averaged.

2 Task 2

Given the parameters from the task the following three results were found:

Assignment	Mean number of jobs in the buffer
1.	135.742
2.	5.584
3.	4.146

Table 2: Results from simulating Task 2, 1000 measurements were taken and averaged.

2.1 Task 2.4

As we can see from table 2 there is a big difference between prioritizing B and having a constant d time, compared to either prioritizing A or having a exponential d time. This is due to the jobs of type A bulking up. During the first second of the simulation only jobs of task A are entering the buffer. Thus during this period there will be a build up of jobs of type B that will start appearing after that first second. Then the system only prioritizes jobs of type B until they are all done, which makes the buffer build up A. Thus it will alternate between doing only jobs of type A and then only jobs of type B.

Compare this to when the d time is exponential. Instead of all the type B jobs coming after exactly 1s, in a wave, they will be randomly distributed. Thus the in stream of type B jobs and type A jobs will be uniform during the whole simulation. And thus it more quickly alters between doing some type B jobs and some type A jobs.

Lastly when prioritizing jobs of type A there will never be a build up of jobs of type A, because the in stream of jobs will always be taken care of. In between A jobs there is time for the jobs of type B, which means it never builds up. However if the stream of type A jobs would be too big, or too fast, we would see a build up of jobs of type B as the simulation goes on.

Figure 1: Graph showing how number of jobs of type A and type B exist at each measurement. It highlights the wave behaviour between jobs of type A and B.

3 Task 3

The simulated values are close to the theoretical values given the very limited amount of measurements made.

Theory N	Simulated N	Theory T	Simulated T
20.00	17.68	22.00	19.50
4.00	4.19	6.00	6.27
2.00	1.98	4.00	3.96

Table 3: Results from simulating Task 3 compared to theoretical values after 1000 measurements

4 Task 4

In table 4 you can see our estimated of the transient phase given the graph in figure 2

Sub-task	Transient phase
1	100
2	20
3	200

Table 4: Approximated values of the transient phase

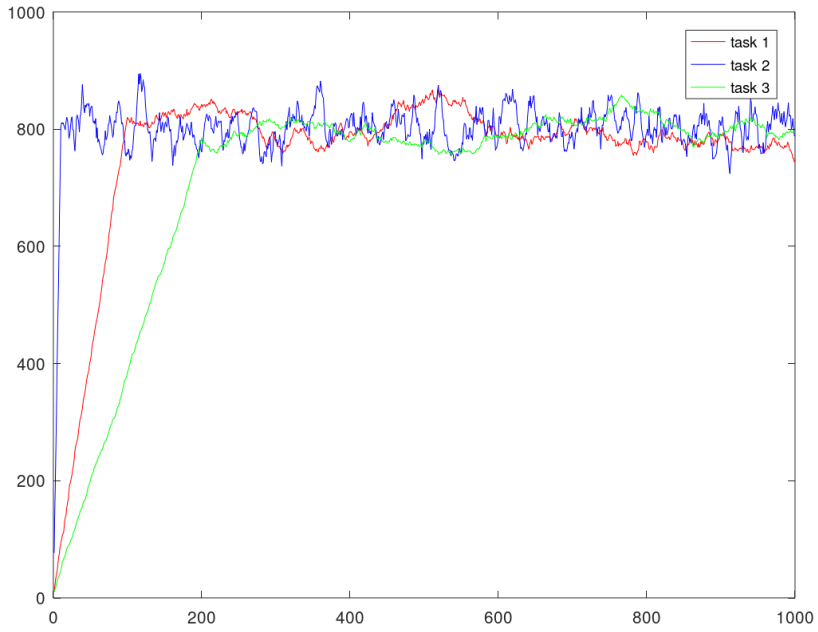

Figure 2: A graph representation of the three first sub-tasks

From table 5 we can draw the conclusion that sub-task 5 is the most stable of runs. That would in turn imply that measuring often and long will give a more accurate result since sub-task 6 was not as accurate while having the same amount of measure points but longer between them.

Sub-task	Mean	Standard Deviation	CI Length
4	39,700	1,434	56
5	39,983	1,382	156
6	40,082	1,423	87

Table 5: A graph of mean, standard deviation and confidential interval length of the last three sub-tasks

5 Task 5

5.1 Task 5.2

In table 6 we can see that Little's Law differs on the three different algorithms. This is to be expected due to there being less waiting for someone else being serviced when jobs are sent to empty queues more often.

5.2 Task 5.3

Picking the Queue with the least number of jobs in the buffer is always the best strategy. However when the queues are outperforming the arrivals by a lot the difference is minimal.

Arrival Mean	Algorithm	Mean L	Mean W	Lambda	Little's Law ($\lambda W/L = 1$)
0.11	Random	46.92	4.69	9.10	0.91
0.11	Round Robin	26.60	2.45	9.09	0.84
0.11	Least in Q	10.45	0.71	9.09	0.62
0.12	Random	23.57	2.35	8.34	0.83
0.12	Round Robin	13.83	1.21	8.33	0.73
0.12	Least in Q	6.72	0.40	8.33	0.49
0.15	Random	9.31	0.95	6.66	0.68
0.15	Round Robin	5.99	0.49	6.66	0.54
0.15	Least in Q	3.93	0.22	6.67	0.38
2.00	Random	0.26	0.17	0.50	0.32
2.00	Round Robin	0.25	0.16	0.50	0.31
2.00	Least in Q	0.25	0.16	0.50	0.32

Table 6: Results from running a distributor with different arrival means and algorithms for choosing an queue of five to send the arrival to. With simulations running for 100,000s

6 Task 6

The average time he finishes work each day is 18:00
The average serve time is about 50 minutes

7 Task 7

Mean time until all parts have broken down after 1000 runs: 3.704

8 Code

8.1 Event, EventListClass

For Tasks 1, 2, 3, 4, 6, and 7 the classes Event and EventListClass are identical to the ones received except for package.

8.2 Proc, Signal, SignalList

For Task 5 the classes Proc, Signal, and SignalList are identical to the ones received except for package.

8.3 Task 1

8.3.1 MainSimulation.java

```
package task1;

import java.io.*;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws
     IOException {
    System.out.println(String.format("%16s|%16s|%16s",
        "Q1 arrival time", "Mean Q2 size", "Rejected
       chance"));

    double start = 0.5;
    double end = 5.0;
    double step = 0.5;

    double amp = end - start;
    int itr = (int) Math.ceil(amp / step);
    for (int i = 0; i <= itr; i++) {
      double arrTime = start + amp * i / itr;

      Event actEvent;
      State actState = new State(arrTime);
      eventList = new EventListClass();
      insertEvent(ARRIVALQ1, 0);
      insertEvent(MEASURE, actState.expRandom(5));

      while (actState.noMeasurements < 10000) {
        actEvent = eventList.fetchEvent();
        time = actEvent.eventTime;
        actState.treatEvent(actEvent);
      }

      System.out.println(String.format("%16.2f|%16.2f
          |%15.2f%%", arrTime,
```

```
            1.0 * actState.accumulated / actState.
               noMeasurements,
            100.0 * actState.rejected / actState.arrived
               ));
    }

  }
}
```

8.3.2 State.java

```
package task1;

import java.util.*;

class State extends GlobalSimulation {

  public final double arrTime;

  public State(double arrTime) {
    this.arrTime = arrTime;
  }

  public int numberInQueue1 = 0, numberInQueue2,
     accumulated = 0, arrived = 0, rejected = 0,
     noMeasurements = 0;

  Random slump = new Random();

  public double expRandom(double mean) {
    return -mean * Math.log(1 - slump.nextDouble());
  }

  public void treatEvent(Event x) {
    switch (x.eventType) {
      case ARRIVALQ1:
        arrivalq1();
        break;
      case DEPARTQ1:
        departq1();
```

```
        break;
      case DEPARTQ2:
        departq2();
        break;
      case MEASURE:
        measure();
        break;
    }
  }

  private void arrivalq1() {
    arrived++;
    if (numberInQueue1 < 10) { // ACCEPT
      numberInQueue1++;
      if (numberInQueue1 == 1)
        insertEvent(DEPARTQ1, time + expRandom(2.1));
    } else { // REJECT
      rejected++;
    }
    insertEvent(ARRIVALQ1, time + arrTime);
  }

  private void departq1() {
    numberInQueue1--;
    if (numberInQueue1 > 0)
      insertEvent(DEPARTQ1, time + expRandom(2.1));

    numberInQueue2++;
    if (numberInQueue2 == 1)
      insertEvent(DEPARTQ2, time + 2);
  }

  private void departq2() {
    numberInQueue2--;
    if (numberInQueue2 > 0)
      insertEvent(DEPARTQ2, time + 2);
  }

  private void measure() {
    accumulated = accumulated + numberInQueue2;
```

```
    noMeasurements++;
    insertEvent(MEASURE, time + expRandom(5));
  }
}
```

8.3.3 GlobalSimulation.java

```
package task1;

public class GlobalSimulation {

  public static final int ARRIVALQ1 = 1, DEPARTQ1 = 2,
      DEPARTQ2 = 3, MEASURE = 4;
  public static double time = 0;
  public static EventListClass eventList = new
     EventListClass();

  public static void insertEvent(int type, double
     TimeOfEvent) {
    eventList.InsertEvent(type, TimeOfEvent);
  }
}
```

8.4 Task 2

8.4.1 MainSimulation.java

```
package task2;

import java.io.*;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws
     IOException {
    Event actEvent;
    State actState = new State(); // The state that
       shoud be used
    // Some events must be put in the event list at
       the beginning
    insertEvent(ARRIVAL_A, 0);
    insertEvent(MEASURE, 0.1);

    // The main simulation loop
    while (actState.noMeasurements < 1000) {
      actEvent = eventList.fetchEvent();
      time = actEvent.eventTime;
      actState.treatEvent(actEvent);
    }

    // Printing the result of the simulation, in this
       case a mean value
    System.out.println(1.0 * actState.accumulated /
       actState.noMeasurements);

    File file = new File(String.format("results/task2
       /%s.txt", System.currentTimeMillis()));
    FileWriter fw = new FileWriter(file);
    StringBuffer sb = new StringBuffer();
    actState.data.forEach(dp -> sb.append(String.
       format("%f, %d, %d%n", dp.time, dp.aInBuffer,
       dp.bInBuffer)));
    fw.write(sb.toString());
```

```
    fw.close();
  }
}
```

8.4.2 State.java

```
package task2;

import java.util.*;

class State extends GlobalSimulation {

  // Here follows the state variables and other
     variables that might be needed
  // e.g. for measurements
  public int accumulated, noMeasurements;
  public int aInBuffer, bInBuffer;

  class DataPoint {
    double time;
    int aInBuffer, bInBuffer;

    DataPoint(double time, int aInBuffer, int
       bInBuffer) {
      this.time = time;
      this.aInBuffer = aInBuffer;
      this.bInBuffer = bInBuffer;
    }
  }

  public ArrayList<DataPoint> data = new ArrayList<>()
     ;

  private final double lambda = 150, x_a = 0.002, x_b
     = 0.004, d = 1, measure_time = 0.1;

  Random slump = new Random(); // This is just a
     random number generator

  public double expRandom(double mean) {
```

```
  return -mean * Math.log(1 - slump.nextDouble());
}

public void treatEvent(Event x) {
  switch (x.eventType) {
    case ARRIVAL_A:
      arrivalA();
      break;
    case ARRIVAL_B:
      arrivalB();
      break;
    case DEPART_A:
      departA();
      break;
    case DEPART_B:
      departB();
      break;
    case MEASURE:
      measure();
      break;
  }
}

private void arrivalA() {
  aInBuffer++;
  if (aInBuffer == 1 && bInBuffer == 0)
    insertEvent(DEPART_A, time + x_a);
  insertEvent(ARRIVAL_A, time + expRandom(1 / lambda
     ));
}

private void arrivalB() {
  bInBuffer++;
  if (aInBuffer == 0 && bInBuffer == 1)
    insertEvent(DEPART_B, time + x_b);
}

private void departNext() {
  if (bInBuffer > 0)
    insertEvent(DEPART_B, time + x_b);
```

```
    else if (aInBuffer > 0)
      insertEvent(DEPART_A, time + x_a);
  }

  private void departA() {
    aInBuffer--;
    insertEvent(ARRIVAL_B, time + d);
    departNext();
  }

  private void departB() {
    bInBuffer--;
    departNext();
  }

  private void measure() {
    // measure
    accumulated += aInBuffer + bInBuffer;
    noMeasurements += 1;
    data.add(new DataPoint(time, aInBuffer, bInBuffer)
        );
    insertEvent(MEASURE, time + measure_time);
  }
}
```

8.4.3 GlobalSimulation.java

```
package task2;

public class GlobalSimulation {
  public static final int ARRIVAL_A = 1, ARRIVAL_B =
      2, DEPART_A = 3, DEPART_B = 4, MEASURE = 5;
  public static double time = 0; // The global time
      variable
  public static EventListClass eventList = new
      EventListClass();

  public static void insertEvent(int type, double
      TimeOfEvent) {
    eventList.InsertEvent(type, TimeOfEvent);
```

```
    }
}
```

8.5 Task 3

8.5.1 MainSimulation.java

```
package task3;

import java.io.*;
import java.util.ArrayList;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws
     IOException {
    System.out.println(String.format("%16s|%16s|%16s
       |%16s", "Theory␣N", "Simulated␣N", "Theory␣T",
       "Simulated␣T"));

    ArrayList<Double> meantime = new ArrayList<>();
    meantime.add(1.1);
    meantime.add(1.5);
    meantime.add(2.0);
    for (int i = 0; i < 3; i++) {
      double arrTime = (Double)meantime.get(i);

      Event actEvent;
      State actState = new State(arrTime);
      eventList = new EventListClass();
      insertEvent(ARRIVALQ1, 0);
      insertEvent(MEASURE, actState.expRandom(5));

      while (actState.noMeasurements < 1000) {
        actEvent = eventList.fetchEvent();
        time = actEvent.eventTime;
        actState.treatEvent(actEvent);
      }

      System.out.println(String.format("%16.2f|%16.2f
         |%16.2f|%16.2f", (2)/(arrTime-1),
      1.0 * actState.accumulated / actState.
```

```
            noMeasurements, (2*arrTime)/(arrTime-1),
          1.0 * actState.totalTime / actState.arrived));
      }
    }
}
```

8.5.2 State.java

```
package task3;

import java.util.*;

class State extends GlobalSimulation {

  public final double arrTime;
  public LinkedList<Double> startQueue = new
     LinkedList<>();

  public State(double arrTime) {
    this.arrTime = arrTime;
  }

  public int numberInQueue1 = 0, numberInQueue2,
     accumulated = 0, arrived = 0, noMeasurements = 0;
  public double totalTime;

  Random slump = new Random(); // This is just a
     random number generator

  public double expRandom(double mean) {
    return -mean * Math.log(1 - slump.nextDouble());
  }

  public void treatEvent(Event x) {
    switch (x.eventType) {
      case ARRIVALQ1:
        arrivalq1();
        break;
      case DEPARTQ1:
```

```
        departq1();
        break;
      case DEPARTQ2:
        departq2();
        break;
      case MEASURE:
        measure();
        break;
    }
  }

  private void arrivalq1() {
    arrived++;
    numberInQueue1++;
    if (numberInQueue1 == 1)
      insertEvent(DEPARTQ1, time + expRandom(1));
    insertEvent(ARRIVALQ1, time + expRandom(arrTime));
    startQueue.add(time);
  }

  private void departq1() {
    numberInQueue1--;
    if (numberInQueue1 > 0)
      insertEvent(DEPARTQ1, time + expRandom(1));

    numberInQueue2++;
    if (numberInQueue2 == 1)
      insertEvent(DEPARTQ2, time + expRandom(1));
  }

  private void departq2() {
    numberInQueue2--;
    if (numberInQueue2 > 0)
      insertEvent(DEPARTQ2, time + expRandom(1));
    totalTime += time - startQueue.poll();
  }

  private void measure() {
    accumulated = accumulated + numberInQueue2 +
       numberInQueue1;
```

```
    noMeasurements++;
    insertEvent(MEASURE, time + expRandom(5));
  }
}
```

8.5.3 GlobalSimulation.java

```
package task3;

public class GlobalSimulation {
  public static final int ARRIVALQ1 = 1, DEPARTQ1 = 2,
      DEPARTQ2 = 3, MEASURE = 4;
  public static double time = 0; // The global time
     variable
  public static EventListClass eventList = new
     EventListClass(); // The event list used in the
     program

  public static void insertEvent(int type, double
     TimeOfEvent) { // Just to be able to skip dot
     notation
    eventList.InsertEvent(type, TimeOfEvent);
  }
}
```

8.6 Task 4

8.6.1 MainSimulation.java

```
package task4;

import java.io.*;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws
     IOException {
    int[] M = {1000, 1000, 1000, 1000, 4000, 4000};
    int[] N = {1000, 1000, 1000, 100, 100, 100};
    int[] x = {100, 10, 200, 10, 10, 10};
    int[] lambda = {8, 80, 4, 4, 4, 4};
    int[] T = {1, 1, 1, 4, 1, 4};
    double average;
    double[] datapoints;
    double standardDeviation;
    int longestCI;
    int currentCI;

    Event actEvent;
    State actState;
    System.out.println(String.format("%20s|%20s|%15s",
        "Mean", "Standard Deviation", "CI Length"));
    for (int i = 0; i < T.length; i++) {
      longestCI = 0;
      currentCI = 0;
      average = 0.0;
      standardDeviation = 0.0;
      datapoints = new double[M[i]];

      actState = new State(N[i], x[i], lambda[i], T[i
         ]); // The state that shoud be used
      // Some events must be put in the event list at
         the beginning
      eventList = new EventListClass();
```

```
insertEvent(ARRIVAL, 0);
insertEvent(MEASURE, 1);

File file = new File(String.format("results/
   task4/%d.txt", i));
FileWriter fw = new FileWriter(file);
StringBuffer sb = new StringBuffer();
// The main simulation loop
while (actState.noMeasurements < M[i]) {
  actEvent = eventList.fetchEvent();
  time = actEvent.eventTime;
  actState.treatEvent(actEvent);
  if (actEvent.eventType == MEASURE) {
    average += actState.currentNbr;
    sb.append(actState.currentNbr + "\n");
    datapoints[actState.noMeasurements-1] =
        actState.currentNbr;
  }
}
fw.write(sb.toString());
fw.close();
average  = average / M[i];
for (int j = 0; j < datapoints.length; j++) {
  datapoints[j] = (Math.pow((datapoints[j]-(
     average)), 2));
  standardDeviation += datapoints[j];
}
standardDeviation = 1.96 * (standardDeviation/M[
   i]);
for (int j = 0; j < datapoints.length; j++) {
  if (average+standardDeviation >= datapoints[j]
      && average-standardDeviation <= datapoints
      [j] ){
    if (currentCI == longestCI) {
      longestCI++;
    }
    currentCI++;
  } else {
    currentCI = 0;
  }
```

```
        }
        System.out.println(String.format("%20f|%20f|%15d
            ", (average), Math.sqrt(standardDeviation/(
            average)), longestCI));
      }
    }
}
```

8.6.2 State.java

```
package task4;

import java.util.*;

class State extends GlobalSimulation {
  int servers, serviceTime, lambda, measureTime;

  public State(int servers, int serviceTime, int
     lambda, int measureTime) {
    this.servers = servers;
    this.serviceTime = serviceTime;
    this.lambda = lambda;
    this.measureTime = measureTime;
  }

  public int currentNbr = 0, noMeasurements = 0;

  Random slump = new Random();

  public double expRandom(double mean) {
    return -mean * Math.log(1 - slump.nextDouble());
  }

  public String eventTypeToString(Event x) {
    switch (x.eventType) {
      case ARRIVAL:
        return "ARRIVALQ1";
      case DEPART:
        return "DEPARTQ1";
```

```
      case MEASURE:
        return "MEASURE";
      default:
        return "Uknown";
    }
  }

  public void treatEvent(Event x) {
    switch (x.eventType) {
      case ARRIVAL:
        arrival();
        break;
      case DEPART:
        depart();
        break;
      case MEASURE:
        measure();
        break;
    }
    // printInfo(x);
  }

  private void arrival() {
    if(currentNbr <= servers){
      currentNbr++;
      insertEvent(DEPART, time + serviceTime);
    }
    insertEvent(ARRIVAL, time + expRandom(1.0/lambda))
        ;
  }

  private void depart() {
    currentNbr--;
  }

  private void measure() {
    noMeasurements++;
    insertEvent(MEASURE, time + measureTime);
  }
}
```

8.6.3 GlobalSimulation.java

```
package task4;

public class GlobalSimulation {

  public static final int ARRIVAL = 1, DEPART = 2,
     MEASURE = 4;
  public static double time = 0;
  public static EventListClass eventList = new
     EventListClass();

  public static void insertEvent(int type, double
     TimeOfEvent) {
    eventList.InsertEvent(type, TimeOfEvent);
  }
}
```

8.7 Task 5

8.7.1 MainSimulation.java

```
package task5;

import java.io.IOException;
import java.util.List;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.function.Function;
import java.util.stream.Collectors;

public class MainSimulation extends Global {

    public static Proc pickLeast(QS[] input) {
        List<QS> candidates = Arrays.asList(input);
        int min = candidates.stream().mapToInt(qs ->
           qs.numberInQueue).min().getAsInt();
        List<QS> minList = candidates.stream().filter(
           a -> a.numberInQueue <= min).collect(
           Collectors.toList());
        Proc temp = minList.get(random.nextInt(minList
           .size()));
        return temp;
    }

    public static void resetQueues(QS[] queueList) {
        for (int i = 0; i < queueList.length; i++) {
            queueList[i] = new QS(0.5, mean ->
               expRandom(mean));
        }
    }

    public static void main(String[] args) throws
       IOException {
        Signal actSignal;

        QS[] queueList = new QS[5];
```

```
List<Double> arrivalTimes = Arrays.asList(new
    Double[] { 0.11, 0.12, 0.15, 2.0 });

System.out.println(String.format("%12s|%12s|%7
    s|%7s|%7s|%12s", "Arrival Mean", "Algorithm
    ", "Mean L", "Mean W",
         "Lambda", "Little's Law"));

for (double meanA : arrivalTimes) {
     List<Gen> generators = new ArrayList<Gen
         >();

     Gen RandomGenerator = new Gen("Random",
        meanA, queueList, ql -> ql[random.
        nextInt(ql.length)]);
     Gen RoundRobinGenerator = new Gen("Round
        Robin", meanA, queueList, new Function<
        Proc[], Proc>() {
         private int counter = 0;

         @Override
         public Proc apply(Proc[] ql) {
             Proc temp = ql[counter];
             counter = (counter + 1) % ql.
                length;
             return temp;
         }
     });
     Gen LeastGenerator = new Gen("Least in Q",
         meanA, queueList, ql -> pickLeast((QS
        []) ql));

     generators.add(RandomGenerator);
     generators.add(RoundRobinGenerator);
     generators.add(LeastGenerator);

     for (Gen gen : generators) {
         resetGlobal();
```

```
                resetQueues(queueList);

                new SignalList();

                SignalList.SendSignal(READY, gen, time
                    );
                for (QS q : queueList) {
                    SignalList.SendSignal(MEASURE, q,
                        time);
                }

                while (time < 100000) {
                    actSignal = SignalList.FetchSignal
                        ();
                    time = actSignal.arrivalTime;
                    actSignal.destination.TreatSignal(
                        actSignal);
                }

                double meanL = Arrays.asList(queueList
                    ).stream()
                        .mapToDouble(q -> 1.0 * q.
                            accumulated / q.
                            noMeasurements).sum();
                double meanW = Arrays.asList(queueList
                    ).stream().mapToDouble(q -> 1.0 * q
                    .serviceTime / q.arrived).sum()
                        / queueList.length;
                double lambda = Arrays.asList(
                    queueList).stream().mapToDouble(q
                    -> q.arrived).sum() / time;

                System.out.println(String.format("
                    %12.2f|%12s|%7.2f|%7.2f|%7.2f|%12.2
                    f", meanA, gen.name, meanL, meanW,
                        lambda, lambda * meanW / meanL
                            ));
            }
        }
    }
```

```
}
```

8.7.2 Gen.java

```
package task5;

import java.util.function.Function;
import java.util.function.DoubleFunction;

class Gen extends Proc {

  public Proc[] sendTo;
  public double mean;
  public Function<Proc[], Proc> procPicker;

  public String name;

  public DoubleFunction<Double> timePicker;

  public Gen(String name, double mean, Proc[] sendTo,
     Function<Proc[], Proc> procPicker) {
    this.name = name;
    this.mean = mean;
    this.sendTo = sendTo;
    this.procPicker = procPicker;
  }

  public void TreatSignal(Signal x) {
    switch (x.signalType) {
      case READY:
        SignalList.SendSignal(ARRIVAL, procPicker.
            apply(sendTo), time);
        SignalList.SendSignal(READY, this, time + 2 *
            random.nextDouble() * mean);
        break;
    }
  }
}
```

8.7.3 QS.java

```
package task5;

import java.util.function.DoubleFunction;
import java.util.LinkedList;

class QS extends Proc {
  public int numberInQueue, arrived, serviceTime,
     accumulated, noMeasurements;
  private LinkedList<Double> arriveTimes = new
     LinkedList<>();
  public Proc sendTo;

  DoubleFunction<Double> timeFunc;
  double mean;

  public QS(double mean, DoubleFunction<Double>
     timeFunc) {
    this.timeFunc = timeFunc;
    this.mean = mean;
  }

  public void TreatSignal(Signal x) {
    switch (x.signalType) {
      case ARRIVAL:
        arrived++;
        arriveTimes.add(time);
        numberInQueue++;
        if (numberInQueue == 1) {
          SignalList.SendSignal(READY, this, time +
             timeFunc.apply(mean));
        }
        break;
      case READY:
        numberInQueue--;
        serviceTime += (time - arriveTimes.pop());
        if (sendTo != null) {
          SignalList.SendSignal(ARRIVAL, sendTo, time)
```

```
                ;
            }
            if (numberInQueue > 0) {
              SignalList.SendSignal(READY, this, time +
                 timeFunc.apply(mean));
            }
            break;
          case MEASURE:
            noMeasurements++;
            accumulated = accumulated + numberInQueue;
            SignalList.SendSignal(MEASURE, this, time +
               expRandom(1));
            break;
        }
      }
}
```

8.7.4 Global.java

```
package task5;

import java.util.Random;

public class Global {
    public static final int ARRIVAL = 1, READY = 2,
       MEASURE = 3;
    public static double time = 0;

    public static Random random = new Random();

    public static void resetGlobal() {
        time = 0;
        random = new Random();
    }

    public static double expRandom(double mean) {
        return -mean * Math.log(1 - random.nextDouble
           ());
    }
```

```
}
```

8.8 Task 6

8.8.1 MainSimulation.java

```
package task6;

import java.io.*;
import java.util.LinkedList;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws
     IOException {
    int minutesInDay = (17 * 60) - (9 * 60);
    int overTime = 0;
    System.out.println(String.format("%16s|%16s", "
       Average overtime", "Average Serv-time"));
    Event actEvent;
    State actState;
    LinkedList<Double> startTime;
    double averageServTime = 0;
    double totalAverageServTime = 0;

    for (int i = 0; i < 1000; i++) {
      handling = 0;
      time = 0;
      averageServTime = 0;
      startTime = new LinkedList<>();
      actState = new State(minutesInDay);
      eventList = new EventListClass();
      insertEvent(ARRIVAL, actState.expRandom(15));

      while (actState.waiting > 0 || actState.
         currentTime < minutesInDay) {
        actEvent = eventList.fetchEvent();
        time = actEvent.eventTime;
        actState.treatEvent(actEvent);
        if (actEvent.eventType == ARRIVAL) {
          startTime.add(actState.currentTime);
```

```
        } else if (actEvent.eventType == DEPART){
          averageServTime += actState.currentTime -
              startTime.poll();
        }
      }
      totalAverageServTime += averageServTime/actState
          .arrivals;
      overTime += (int)(((time/60)*60) + time%60) -
          minutesInDay;
    }
    System.out.println(String.format("%16d|%16d",
        overTime/1000, (int)totalAverageServTime/1000))
        ;
  }
}
```

8.8.2 State.java

```
package task6;

import java.util.*;

class State extends GlobalSimulation {
  public int minutesInDay;
  public double currentTime = 0;
  public int waiting = 0;
  public int arrivals = 0;

  public State(int minutesInDay) {
    this.minutesInDay = minutesInDay;
  }

  Random slump = new Random();

  public double expRandom(double mean) {
    return -mean * Math.log(1 - slump.nextDouble());
  }

  public void treatEvent(Event x) {
```

```
    switch (x.eventType) {
      case ARRIVAL:
        arrival();
        break;
      case DEPART:
        depart();
        break;
      case MEASURE:
        measure();
        break;
    }
  }

  private void arrival() {
    currentTime = time;
    arrivals++;
    double temp1 = time + expRandom(15);
    if(temp1 <= minutesInDay){
      waiting ++;
      insertEvent(ARRIVAL, temp1);
    }
    if(handling == 0 && waiting > 0){
      insertEvent(DEPART, time + ((slump.nextDouble()
          * 10) + 10));
      handling ++;
      waiting--;
    }
  }

  private void depart() {
    handling--;
    currentTime = time;
    if (handling == 0 && waiting > 0){
      insertEvent(DEPART, time + ((slump.nextDouble()
          * 10) + 10));
      handling++;
      waiting--;
    } else {
      insertEvent(MEASURE, time);
    }
```

```
  }

  private void measure() {
    currentTime = time;
    insertEvent(MEASURE, time + 1);
  }
}
```

8.8.3 GlobalSimulation.java

```
package task6;

public class GlobalSimulation {

  public static final int ARRIVAL = 1, DEPART = 2,
     MEASURE = 4;
  public static double time = 0, handling = 0; // The
     global time variable
  public static EventListClass eventList = new
     EventListClass();

  public static void insertEvent(int type, double
     TimeOfEvent) {
    eventList.InsertEvent(type, TimeOfEvent);
  }
}
```

8.9 Task 7

8.9.1 MainSimulation.java

```
package task7;

import java.io.*;

public class MainSimulation extends GlobalSimulation {

  public static void main(String[] args) throws
     IOException {
    double mean = 0;
    for (int i = 0; i <= 1000; i++) {
      Event actEvent;
      State actState = new State();
      eventList = new EventListClass();
      insertEvent(COMP_1, (actState.slump.nextDouble()
         *4) + 1);
      insertEvent(COMP_2, (actState.slump.nextDouble()
         *4) + 1);
      insertEvent(COMP_3, (actState.slump.nextDouble()
         *4) + 1);
      insertEvent(COMP_4, (actState.slump.nextDouble()
         *4) + 1);
      insertEvent(COMP_5, (actState.slump.nextDouble()
         *4) + 1);

      // The main simulation loop
      while (actState.isAlive) {
        actEvent = eventList.fetchEvent();
        time = actEvent.eventTime;
        actState.treatEvent(actEvent);
      }
      mean += time;
    }
    System.out.println(mean/1000);
  }
}
```

8.9.2 State.java

```
package task7;

import java.util.*;

class State extends GlobalSimulation {

  public boolean isAlive = true,
  comp1_alive = true,
  comp2_alive = true,
  comp3_alive = true,
  comp4_alive = true,
  comp5_alive = true;
  Random slump = new Random();

  public void treatEvent(Event x) {
    switch (x.eventType) {
      case COMP_1:
        comp_1();
        break;
      case COMP_2:
        comp_2();
        break;
      case COMP_3:
        comp_3();
        break;
      case COMP_4:
        comp_4();
        break;
      case COMP_5:
        comp_5();
        break;
    }
  }

  private void comp_1() {
    comp1_alive = false;
    comp2_alive = false;
```

```
    comp5_alive = false;
    alive();
  }

  private void comp_2() {
    comp2_alive = false;
    alive();
  }

  private void comp_3() {
    comp3_alive = false;
    comp4_alive = false;
    alive();
  }

  private void comp_4() {
    comp4_alive = false;
    alive();
  }

  private void comp_5() {
    comp5_alive = false;
    alive();
  }

  private void alive() {
    if(!comp1_alive && !comp2_alive && !comp3_alive &&
        !comp4_alive && !comp5_alive)
      isAlive = false;
  }
}
```

8.9.3 GlobalSimulation.java

```
package task7;

public class GlobalSimulation {
  public static final int COMP_1 = 1, COMP_2 = 2,
    COMP_3 = 3, COMP_4 = 4, COMP_5 = 5;
```

```
  public static double time = 0; // The global time
     variable
  public static EventListClass eventList = new
     EventListClass();

  public static void insertEvent(int type, double
     TimeOfEvent) {
    eventList.InsertEvent(type, TimeOfEvent);
  }
}
```