Codice Progetto PMCSN

Fabiano Veglianti

January 18, 2022

Contents

1 Package Config

1.1 NetworkConfiguration.java

```
1         package config;
2         /**
3          * Manages the network configuration in terms of EC2
    instances used.
4          * It defines the EC2 instances available.
5          * */
6         public class NetworkConfiguration {
7
8           private enum EC2InstanceType {T2Nano, T2Micro, T2Small}
9
10          private class EC2Instance{
11            private static final double
    MEAN_VM1_SERVICE_TIME_T2NANO = 1.0/15.0;
12            private static final double
    MEAN_VM1_SERVICE_TIME_T2MICRO = 1.0/20.0;
13            private static final double
    MEAN_VM1_SERVICE_TIME_T2SMALL = 1.0/30.0;
14            private static final double
    MEAN_VM2CPU_SERVICE_TIME_T2NANO = 1.0/12.0;
15            private static final double
    MEAN_VM2CPU_SERVICE_TIME_T2MICRO = 1.0/15.0;
16            private static final double
    MEAN_VM2CPU_SERVICE_TIME_T2SMALL = 1.0/22.0;
17            private static final double
    MEAN_VM2BAND_SERVICE_TIME_T2NANO = 1.0/4.0;
18            private static final double
    MEAN_VM2BAND_SERVICE_TIME_T2MICRO = 1.0/7.0;
19            private static final double
    MEAN_VM2BAND_SERVICE_TIME_T2SMALL = 1.0/9.0;
20
21            public double getPricePerMinute(){
22              switch (this.type){
23                case T2Nano:
24                  return 0.01;
25                case T2Micro:
26                  return 0.02;
27                case T2Small:
28                  return 0.05;
29                default:
30                  System.err.println("ERRORE FATALE");
31                  System.exit(-1);
32                  return -1;
33              }
34            }
35
36            private EC2InstanceType type;
37
38            public EC2Instance(EC2InstanceType type){
39              this.type = type;
40            }
41
42            public double getMeanVm1ServiceTime(){
43              switch (this.type){
```

```
                case T2Nano:
                  return MEAN_VM1_SERVICE_TIME_T2NANO;
                case T2Micro:
                  return MEAN_VM1_SERVICE_TIME_T2MICRO;
                case T2Small:
                  return MEAN_VM1_SERVICE_TIME_T2SMALL;
                default:
                  System.err.println("ERRORE FATALE");
                  System.exit(-1);
                  return -1;
              }
            }

            public double getMeanVm2cpuServiceTime(){
              switch (this.type){
                case T2Nano:
                  return MEAN_VM2CPU_SERVICE_TIME_T2NANO;
                case T2Micro:
                  return MEAN_VM2CPU_SERVICE_TIME_T2MICRO;
                case T2Small:
                  return MEAN_VM2CPU_SERVICE_TIME_T2SMALL;
                default:
                  System.err.println("ERRORE FATALE");
                  System.exit(-1);
                  return -1;
              }
            }

            public double getMeanVm2bandServiceTime(){
              switch (this.type){
                case T2Nano:
                  return MEAN_VM2BAND_SERVICE_TIME_T2NANO;
                case T2Micro:
                  return MEAN_VM2BAND_SERVICE_TIME_T2MICRO;
                case T2Small:
                  return MEAN_VM2BAND_SERVICE_TIME_T2SMALL;
                default:
                  System.err.println("ERRORE FATALE");
                  System.exit(-1);
                  return -1;
              }
            }
          }

          private EC2Instance vm1Instance;
          private EC2Instance vm2Instance;

          public void setConfiguration(char configurationCode){
            switch (configurationCode){
              case 'A':
                vm1Instance = new EC2Instance(EC2InstanceType.
T2Nano);
                vm2Instance = new EC2Instance(EC2InstanceType.
T2Nano);
                break;
              case 'B':
                vm1Instance = new EC2Instance(EC2InstanceType.
```

```
T2Nano);
99                  vm2Instance = new EC2Instance(EC2InstanceType.
T2Micro);
100                 break;
101               case 'C':
102                 vm1Instance = new EC2Instance(EC2InstanceType.
T2Nano);
103                 vm2Instance = new EC2Instance(EC2InstanceType.
T2Small);
104                 break;
105               case 'D':
106                 vm1Instance = new EC2Instance(EC2InstanceType.
T2Micro);
107                 vm2Instance =  new EC2Instance(EC2InstanceType.
T2Nano);
108                 break;
109               case 'E':
110                 vm1Instance = new EC2Instance(EC2InstanceType.
T2Micro);
111                 vm2Instance = new EC2Instance(EC2InstanceType.
T2Micro);
112                 break;
113               case 'F':
114                 vm1Instance =  new EC2Instance(EC2InstanceType.
T2Micro);
115                 vm2Instance = new EC2Instance(EC2InstanceType.
T2Small);
116                 break;
117               case 'G':
118                 vm1Instance =new EC2Instance(EC2InstanceType.
T2Small);
119                 vm2Instance = new EC2Instance(EC2InstanceType.
T2Nano);
120                 break;
121               case 'H':
122                 vm1Instance = new EC2Instance(EC2InstanceType.
T2Small);
123                 vm2Instance = new EC2Instance(EC2InstanceType.
T2Micro);
124                 break;
125               case 'I':
126                 vm1Instance = new EC2Instance(EC2InstanceType.
T2Small);
127                 vm2Instance = new EC2Instance(EC2InstanceType.
T2Small);
128                 break;
129               default:
130                 System.err.println("ERRORE FATALE");
131                 System.exit(-1);
132                 break;
133             }
134
135             Params.MEAN_SERVICE_TIME_VM1 = vm1Instance.
getMeanVm1ServiceTime();
136             Params.MEAN_SERVICE_TIME_VM2CPU = vm2Instance.
getMeanVm2cpuServiceTime();
137             Params.MEAN_SERVICE_TIME_VM2BAND = vm2Instance.
```

```
        getMeanVm2bandServiceTime();
138
139             Params.VM1_PRICE_PER_MINUTE = vm1Instance.
        getPricePerMinute();
140             Params.VM2CPU_PRICE_PER_MINUTE = vm2Instance.
        getPricePerMinute();
141             Params.VM2BAND_PRICE_PER_MINUTE = vm2Instance.
        getPricePerMinute();
142
143           }
144
145           private static NetworkConfiguration instance = null;
146
147           private NetworkConfiguration(){}
148
149           public static NetworkConfiguration getInstance() {
150             if(instance == null) {
151               instance = new NetworkConfiguration();
152             }
153             return instance;
154           }
155         }
```

1.2 Params.java

```
1  package config;
2  /**
3   * It set the simulation parameters.
4   * */
5  public class Params {
6
7      //routing matrix
8      private static final double P01 = 0;
9      private static final double P02 = 1;
10     private static final double P03 = 0;
11
12     public static final double P10 = 0.0;
13     public static final double P11 = 0.2;
14     public static final double P12 = 0;
15     public static final double P13 = 0.8;
16
17     public static final double P20 = 0;
18     public static final double P21 = 0.8;
19     public static final double P22 = 0;
20     public static final double P23 = 0.2;
21
22     public static final double P30 = 0.2;
23     public static final double P31 = 0.3;
24     public static final double P32 = 0;
25     public static final double P33 = 0;
26     public static final double P34 = 0.5;
27
28     public static final double P40 = 1;
29     public static final double P41 = 0;
30     public static final double P42 = 0;
31     public static final double P43 = 0;
32
33
34     //network parameters
35     public static double MEAN_INTERARRIVAL_RATE = 7.0;
36     public static double MEAN_INTERARRIVAL_S3 = 1/(
       MEAN_INTERARRIVAL_RATE);
37     public static double MEAN_SERVICE_TIME_VM1;
38     public static double MEAN_SERVICE_TIME_S3 = 1.0;
39     public static double MEAN_SERVICE_TIME_VM2CPU;
40     public static double MEAN_SERVICE_TIME_VM2BAND;
41
42     public static double VM1_PRICE_PER_MINUTE;
43     public static double S3_PRICE_PER_REQUEST = 0.02;
44     public static double VM2CPU_PRICE_PER_MINUTE;
45     public static double VM2BAND_PRICE_PER_MINUTE;
46
47     //simulations enablers
48     public static final boolean runFiniteHorizonSimulation = true;
49     public static final boolean runBatchMeansSimulation = false;
50
51     //simulations parameters
52     public static final double NUM_REPLICAS = 64;
53
54     //finite horizon parameters
```

```
55      public static final int FH_TIME_INCREASE_STEP = 2;
56      public static final double FH_MAX_TIME_LIMIT = 100;
57
58      //batch means parameters
59      public static final int BM_NUM_BATCHES = 512;
60      public static final int BM_NUM_EVENTS = 1048576;
61
62      //debug parameters
63      public static final boolean DEBUG_MODE_ON = false;
64      public static final int DEBUG_ITERATIONS = 100000;
65  }
```

2 Package Debug

2.1 ArrivalsCounter.java

```java
package debug;
import utils.BetweenRunsMetric;
import entity.EventType;
import java.util.ArrayList;
import java.util.HashMap;
/**
 * It counts arrivals from outside.
 * In this network, arrivals from outside can only happen to the S3 center,
 * however this class is designed for a more general scenario.
 *
 * It counts the arrivals that occur in each replica and, applying the Welford algorithm,
 * it calculates the mean value between the replicas.
 *
 * */
public class ArrivalsCounter {

    HashMap<EventType, Integer> arrivalsCounterPerReplica;
    HashMap<EventType, BetweenRunsMetric> arrivalsCounter;

    public ArrivalsCounter(){
        arrivalsCounterPerReplica = new HashMap<>();
        arrivalsCounter = new HashMap<>();

        arrivalsCounterPerReplica.put(EventType.ARRIVALS3, 0);

        arrivalsCounter.put(EventType.ARRIVALS3, new BetweenRunsMetric());
    }

    public void increaseCounter(EventType type){
        arrivalsCounterPerReplica.put(type, arrivalsCounterPerReplica.get(type) +1);
    }

    public void resetCounters() {
        for(EventType key: arrivalsCounterPerReplica.keySet()){
            arrivalsCounterPerReplica.put(key, 0);
        }
    }

    public void commitCounters(double current){
        for(EventType key: arrivalsCounterPerReplica.keySet()){
            arrivalsCounter.get(key).updateMetrics(arrivalsCounterPerReplica.get(key)/current);
        }
    }

    public ArrayList<Double> getCounters(){
        ArrayList<Double> counters = new ArrayList<>();

        EventType[] keys = {EventType.ARRIVALS3};
```

```
49
50          for(EventType key: keys){
51              counters.add(arrivalsCounter.get(key).getSampleMean());
52          }
53
54          return counters;
55      }
56  }
```

2.2 RoutingMatrix.java

```java
package debug;
import utils.BetweenRunsMetric;
import entity.ServerEnum;
import java.util.ArrayList;
import java.util.HashMap;
/**
 * It calculate routing frequencies and keeps them in a list of
    RouteData.
 *
 * RouteData calculate the routing frequency between source and
    destination.
 * Source and Destination are ServerEnum data types, but
    destination can be null to indicate the outbound routing.
 *
 *
 * */
public class RoutingMatrix {

    private static class RouteData {
        private final ServerEnum source;
        private final ServerEnum destination;
        private int counter;
        private BetweenRunsMetric estimation;

        private RouteData(ServerEnum source, ServerEnum destination
    ){
            this.source = source;
            this.destination = destination;

            counter = 0;
            estimation = new BetweenRunsMetric();
        }

        private void resetCounter(){
            counter = 0;
        }


        private void increaseCounter(){
            counter += 1;
        }

        private void commitCounter(double totalSourceDepartures){
            double value = counter/totalSourceDepartures;
            estimation.updateMetrics(value);
        }


    }

    ArrayList<RouteData> routeDataList;

    public RoutingMatrix(){
        routeDataList = new ArrayList<>();
        for(ServerEnum source: ServerEnum.values()){
```

```
52              RouteData data = new RouteData(source, null);
53              routeDataList.add(data);
54              for (ServerEnum destination: ServerEnum.values()){
55                  data = new RouteData(source, destination);
56                  routeDataList.add(data);
57              }
58          }
59      }
60
61      private RouteData findRouteData(ServerEnum source, ServerEnum
        destination){
62          for(RouteData data: routeDataList){
63              if(destination == null){
64                  if (data.source.equals(source) && data.destination
        == null)
65                      return data;
66              } else {
67                  if(data.destination != null){
68                      if(data.source.equals(source) && data.
        destination.equals(destination)){
69                          return data;
70                      }
71                  }
72              }
73
74          }
75          return null;
76      }
77
78      public void increaseCounter(ServerEnum source, ServerEnum
        destination){
79          RouteData data = findRouteData(source, destination);
80          assert data != null;
81          data.increaseCounter();
82      }
83
84      public void commitCounters(){
85          HashMap<ServerEnum, Integer> mapSourceDepartures = new
        HashMap<>();
86          for(ServerEnum serverEnum : ServerEnum.values()){
87              mapSourceDepartures.put(serverEnum, 0);
88          }
89
90          for(RouteData data: routeDataList){
91              mapSourceDepartures.put(data.source,
        mapSourceDepartures.get(data.source)+data.counter);
92          }
93
94          for(RouteData data: routeDataList){
95              data.commitCounter(mapSourceDepartures.get(data.source)
        );
96          }
97      }
98
99      public void resetCounters(){
100         for(RouteData data: routeDataList){
101             data.resetCounter();
```

```
102         }
103     }
104 
105     public ArrayList<Double> getRoutingFrequencies(){
106         ArrayList<Double> frequencies = new ArrayList<>();
107 
108         ServerEnum[] serverEnums = {ServerEnum.VM1, ServerEnum.S3,
    ServerEnum.VM2CPU, ServerEnum.VM2BAND};
109 
110         for(ServerEnum source: serverEnums){
111             RouteData data = findRouteData(source, null);
112             assert data != null;
113             frequencies.add(data.estimation.getSampleMean());
114             for(ServerEnum destination: serverEnums){
115                 data = findRouteData(source, destination);
116                 assert data != null;
117                 frequencies.add(data.estimation.getSampleMean());
118 
119             }
120         }
121         return frequencies;
122     }
123 }
```

3 Package Entity

3.1 Event.java

```
1  package entity;
2  import utils.Generator;
3  import static config.Params.*;
4  import static entity.SchedulingDisciplineType.FIFO;
5  /**
6   * It models an occuring event.
7   * */
8  public class Event {
9
10     private ServerEnum nextCenter;
11
12     private final EventType type;
13
14
15     /**
16      * endTime keeps the Clock value at which the event will be
       triggered.
17      * If the event has associated a job (type is not Arrival*):
18      *      arrivalTime keeps the Clock value at which the job
       entered in the server;
19      *      serviceTime keeps the amount of time the server must
       spend on the job before the event can be triggered.
20      * */
21     private double endTime;
22     private double arrivalTime;
23     private double serviceTime;
24
25     public EventType getType() {
26         return type;
27     }
28     public double getEndTime(){
29         return endTime;
30     }
31     public double getServiceTime(){
32         return serviceTime;
33     }
34     public double getArrivalTime() { return arrivalTime; }
35     public ServerEnum getNextCenter(){
36         return nextCenter;
37     }
38
39
40     /**
41      * If the EventType is Completation* the nextCenter variable is
       set to a value according to a RNG and the
42      * routing table.
43      *
44      * If the EventType is Completation* and the associated center'
       s scheduling discipline is FIFO the endTime variable
45      * is not set immediately.
46      *
47      * If the EventType is Completation* and the associated center'
       s scheduling discipline is PS the endTime variable is
```

```
48       * set immediately, it depends of the number of jobs in the
         center and it can change if the number of jobs
49       * in the center changes
50       *
51       * If the EventType is Completation* and the associated center'
         s scheduling discipline is IS the endTime variable is
52       * set immediately, is equals to arrivalTime+serviceTime and it
         cannot change.
53      * */
54      public Event(EventType type, Generator t, double currentTime,
         double numJobsInServer, SchedulingDisciplineType
         serverSchedulingDiscipline){
55          this.type = type;
56          double routingProbability;
57          switch (type){
58
59              case ARRIVALS3:
60                  t.selectStream(3);
61                  endTime = currentTime + t.exponential(
         MEAN_INTERARRIVAL_S3);
62                  break;
63
64              case COMPLETATIONVM1:
65                  t.selectStream(57);
66                  double distributionProb = t.uniform(0,1);
67
68
69                  t.selectStream(7);
70                  arrivalTime = currentTime;
71                  if(serverSchedulingDiscipline == FIFO) {
72                      serviceTime = t.exponential(
         MEAN_SERVICE_TIME_VM1);
73                  } else {
74                      serviceTime = t.exponential(
         MEAN_SERVICE_TIME_VM1);
75                      endTime = currentTime + serviceTime * (
         numJobsInServer+1);
76                  }
77
78
79                  t.selectStream(37);
80                  routingProbability = t.uniform( 0, 1);
81
82                  if(routingProbability<P10) {
83                      this.nextCenter = null;
84                  }else if (P10 <= routingProbability &&
         routingProbability < P10+P11){
85                      this.nextCenter = ServerEnum.VM1;
86                  } else if (P10+P11 <= routingProbability &&
         routingProbability < P10+P11+P12){
87                      this.nextCenter = ServerEnum.S3;
88                  } else {
89                      this.nextCenter = ServerEnum.VM2CPU;
90                  }
91
92
93                  break;
```

```
94
95          case COMPLETATIONS3:
96              t.selectStream(9);
97              arrivalTime = currentTime;
98              serviceTime = t.exponential(MEAN_SERVICE_TIME_S3);
99              endTime = currentTime + serviceTime;
100
101
102             t.selectStream(39);
103             routingProbability = t.uniform( 0, 1);
104
105             if(routingProbability<P20){
106                 this.nextCenter = null;
107             } else if (P20 <= routingProbability &&
    routingProbability < P20+P21) {
108                 this.nextCenter = ServerEnum.VM1;
109             } else if (P20+P21 <= routingProbability &&
    routingProbability < P20+P21+P22){
110                 this.nextCenter = ServerEnum.S3;
111             } else {
112                 this.nextCenter = ServerEnum.VM2CPU;
113             }
114
115
116             break;
117
118         case COMPLETATIONVM2CPU:
119             t.selectStream(11);
120             arrivalTime = currentTime;
121             if(serverSchedulingDiscipline == FIFO) {
122                 serviceTime = t.exponential(
    MEAN_SERVICE_TIME_VM2CPU);
123             } else {
124                 serviceTime = t.exponential(
    MEAN_SERVICE_TIME_VM2CPU);
125                 endTime = currentTime + serviceTime * (
    numJobsInServer+1);
126             }
127
128
129
130             t.selectStream(41);
131             routingProbability = t.uniform( 0, 1);
132
133             if(routingProbability<P30){
134                 this.nextCenter = null;
135             } else if (P30 <= routingProbability &&
    routingProbability < P30+P31){
136                 this.nextCenter = ServerEnum.VM1;
137             } else if(P30+P31 <= routingProbability &&
    routingProbability < P30+P31+P32) {
138                 this.nextCenter = ServerEnum.S3;
139             } else if (P30+P31+P32 <= routingProbability &&
    routingProbability < P30+P31+P32+P33) {
140                 this.nextCenter = ServerEnum.VM2CPU;
141             } else {
142                 this.nextCenter = ServerEnum.VM2BAND;
```

```
143                 }
144 
145                 break;
146 
147             case COMPLETATIONVM2BAND:
148                 t.selectStream(13);
149                 arrivalTime = currentTime;
150                 if(serverSchedulingDiscipline == FIFO) {
151                     serviceTime = t.exponential(
    MEAN_SERVICE_TIME_VM2BAND);
152                 } else {
153                     serviceTime = t.exponential(
    MEAN_SERVICE_TIME_VM2BAND);
154                     endTime = currentTime + serviceTime * (
    numJobsInServer+1);
155                 }
156 
157 
158                 t.selectStream(43);
159                 routingProbability = t.uniform( 0, 1);
160 
161                 if(routingProbability<P40){
162                     this.nextCenter = null;
163                 } else if (P40 <= routingProbability &&
    routingProbability < P40+P41){
164                     this.nextCenter = ServerEnum.VM1;
165                 } else if (P40+P41 <= routingProbability &&
    routingProbability < P40+P41+P42){
166                     this.nextCenter = ServerEnum.S3;
167                 } else {
168                     this.nextCenter = ServerEnum.VM2CPU;
169                 }
170 
171                 break;
172             default:
173                 System.err.println("ERRORE FATALE");
174                 System.exit(-1);
175                 break;
176         }
177     }
178 
179     /**
180      * If the EventType is Completation* and the associated center'
    s scheduling discipline is FIFO the endTime depends
181      * on the time the job is started and the serviceTime
    associated with it.
182      * */
183     public void setEndTime(double startServiceTime){
184         endTime = startServiceTime + serviceTime;
185     }
186 
187     /**
188      * If the EventType is Completation* and the associated center'
    s scheduling discipline is PS the endTime variable
189      * value changes as the number of jobs in the center changes.
190      * Read the documentation for more details about the formulas.
191      * */
```

```
192     public void updateTime(double changeTime, double
        numJobsInServer, boolean isNextEventArrival){
193         if (isNextEventArrival) {
194             endTime = changeTime + (endTime - changeTime) * (
        numJobsInServer + 1) / numJobsInServer;
195         } else {
196             endTime = changeTime + (endTime - changeTime) * (
        numJobsInServer - 1) / numJobsInServer;
197         }
198     }
199
200
201     @Override
202     public String toString(){
203         String string = "";
204         string = string + type + " - " + endTime;
205         return string;
206     }
207 }
```

3.2 EventList.java

```java
package entity;
import entity.EventType;
import utils.Generator;
import entity.Event;
import java.util.HashMap;
/**
 * It maintains the next Event for each eventType.
 * */
public class EventList {

    HashMap<EventType, Event> eventList;

    public EventList(Generator generator, double current){
        eventList = new HashMap<>();
        for(EventType type: EventType.values()) {
            if (type == EventType.ARRIVALS3) {
                Event event = new Event(type, generator, current,
    0, null);
                eventList.put(type, event);
            } else {
                eventList.put(type, null);
            }
        }
    }

    /*
    * Used also to update the Completation* Events if the
    associated center scheduling discipline is PS and the number
    * of jobs in the center changes.
    * */
    public void putEvent(EventType type, Event event){
        eventList.put(type, event);
    }

    public Event removeEventByType(EventType type){
        return eventList.remove(type);
    }

    public Event removeNextEvent(){
        return removeEventByType(this.getNextEventType());
    }

    public EventType getNextEventType(){
        double time = Double.MAX_VALUE;
        EventType type = null;
        for(EventType eventType: EventType.values()){
            Event event = eventList.get(eventType);
            if(event != null){
                if(event.getEndTime() < time){
                    type = eventType;
                    time = event.getEndTime();
                }
            }
        }
        return type;
```

```
54         }
55 }
```

3.3 EventType.java

```
1  package entity;
2  /**
3   * Lists the EventType
4   * */
5  public enum EventType {
6
7      ARRIVALS3(),
8      COMPLETATIONVM1(),
9      COMPLETATIONS3(),
10     COMPLETATIONVM2CPU(),
11     COMPLETATIONVM2BAND();
12 }
```

3.4 Network.java

```
1  package entity;
2  import utils.BetweenRunsMetric;
3  import java.text.DecimalFormat;
4  /**
5   * It is used to handle the network's metrics.
6   * */
7  public class Network {
8
9      //between runs attributes
10     private BetweenRunsMetric throughput;
11     private BetweenRunsMetric wait;
12     private BetweenRunsMetric population;
13     private double currentBatchStartTime;
14
15
16     public  void updateBetweenRunsMetrics(double current){
17         //this operation has not effect if the simulation is not
       BatchMeans
18         double lastArrivalTimeInBatch = lastArrivalTime -
       currentBatchStartTime;
19         double currentBatchDuration = current -
       currentBatchStartTime;
20
21         if(lastArrivalTimeInBatch != 0) {
22             throughput.updateMetrics(departure /
       lastArrivalTimeInBatch);
23             population.updateMetrics(node / currentBatchDuration);
24         }
25         if(departure != 0) {
26             wait.updateMetrics(node / departure);
27         }
28     }
29
30
31     public double[] getThroughputConfidenceInterval(){
32         return throughput.getConfidenceInterval();
33     }
34
35     public double[] getWaitConfidenceInterval(){
36         return wait.getConfidenceInterval();
37     }
38
39
40     public double[] getPopulationConfidenceInterval() {return
       population.getConfidenceInterval(); }
41
42     public double[]
       getThroughputConfidenceIntervalAndAutocorrelationLagOne(){
43         return throughput.
       getConfidenceIntervalAndAutocorrelationLagOne();
44     }
45
46     public double[]
       getWaitConfidenceIntervalAndAutocorrelationLagOne(){
47         return wait.getConfidenceIntervalAndAutocorrelationLagOne()
```

```
    ;
48  }
49
50  public double[]
    getPopulationConfidenceIntervalAndAutocorrelationLagOne(){
51      return population.
    getConfidenceIntervalAndAutocorrelationLagOne();
52  }
53
54
55  public void resetBetweenRunsMetrics(){
56      wait.resetValue();
57      throughput.resetValue();
58      population.resetValue();
59  }
60
61
62  //in run attributes
63  private double node;
64  private double departure;
65  private double lastArrivalTime;
66  private int numJobsInNetwork;
67
68
69  public double getDeparture(){
70      return departure;
71  }
72
73  public void increaseDeparture(){
74      numJobsInNetwork = numJobsInNetwork -1;
75      departure +=1;
76  }
77
78  public int getNumJobsInNetwork(){
79      return numJobsInNetwork;
80  }
81
82  public void resetInRunMetrics(){
83      departure = 0.0;
84      node = 0.0;
85  }
86
87  public void removeAllEventsInNetwork(){
88      numJobsInNetwork = 0;
89  }
90
91
92  public void updateInRunMetrics(double currentTime, double
    nextTime) {
93      if (numJobsInNetwork > 0) {
94          node = node + numJobsInNetwork * (nextTime -
    currentTime);
95      }
96  }
97
98  public void setCurrentBatchStartTime(double
    currentBatchStartTime) {
```

```
99          this.currentBatchStartTime = currentBatchStartTime;
100     }
101
102     public Network(){
103         departure = 0.0;
104         numJobsInNetwork = 0;
105
106         throughput = new BetweenRunsMetric();
107         wait = new BetweenRunsMetric();
108         population = new BetweenRunsMetric();
109
110         node = 0.0;
111     }
112
113
114
115     public void insertJobInNetwork(double current){
116         numJobsInNetwork = numJobsInNetwork +1;
117         lastArrivalTime = current;
118     }
119
120
121     public void printMetrics(double current){
122         DecimalFormat f = new DecimalFormat("###0.00000000000");
123         // dovrebbe essere lastInterarrival al posto di current
124         System.out.println("   average interarrival time =   " + f.
    format(lastArrivalTime / departure));
125         System.out.println("   average wait ............ =   " + f.
    format(node/ departure));
126         System.out.println("   average # in the node ... =   " + f.
    format(node / lastArrivalTime));
127         // dovrebbe essere lastInterarrival al posto di current
128         System.out.println("   lambda .................. =   " + f.
    format(departure/lastArrivalTime));
129         System.out.println("");
130     }
131
132     public void computeAutocorrelationValues() {
133         wait.computeAutocorrelationValues();
134         throughput.computeAutocorrelationValues();
135         population.computeAutocorrelationValues();
136     }
137 }
```

3.5 SchedulingDisciplineType.java

```
1  package entity;
2  /**
3   * Lists the Scheduling Discipline types
4   * */
5  public enum SchedulingDisciplineType {
6
7      FIFO("FIFO"),
8      IS("IS"),
9      PS("PS");
10
11     private final String discipline;
12
13     SchedulingDisciplineType(String discipline) {
14         this.discipline = discipline;
15     }
16 }
```

3.6 Server.java

```
1  package entity;
2  import utils.BetweenRunsMetric;
3  import java.text.DecimalFormat;
4  import java.util.ArrayList;
5  import static entity.SchedulingDisciplineType.*;
6
7  public class Server {
8
9      //fixed attributes
10     private final ServerEnum type;
11     private final SchedulingDisciplineType discipline;
12     public SchedulingDisciplineType getDiscipline() { return
       discipline; }
13     public ServerEnum getType() {
14         return type;
15     }
16
17
18     //between runs attributes
19     private BetweenRunsMetric throughput;
20     private BetweenRunsMetric wait;
21     private BetweenRunsMetric population;
22     private BetweenRunsMetric serviceTime;
23
24     //for batch means only
25     private double currentBatchStartTime;
26
27     public void setCurrentBatchStartTime(double
       currentBatchStartTime) {
28         this.currentBatchStartTime = currentBatchStartTime;
29     }
30
31     public  void updateBetweenRunsMetrics(double current){
32         //these operations have no effect if the simulation is not
       batch means
33         double lastArrivalTimeInBatch = lastArrivalTime -
       currentBatchStartTime;
34         double currentBatchDuration = current -
       currentBatchStartTime;
35
36         if(lastArrivalTimeInBatch != 0) {
37             throughput.updateMetrics(departure /
       lastArrivalTimeInBatch);
38             population.updateMetrics(node / currentBatchDuration);
39         }
40         if(departure != 0) {
41             wait.updateMetrics(node / departure);
42             if (type == ServerEnum.S3) {
43                 serviceTime.updateMetrics(node / departure);
44             } else {
45                 serviceTime.updateMetrics(server / departure);
46             }
47         }
48
49
```

```
50      }
51
52      public double[] getServiceTimeInterval(){
53          return serviceTime.getConfidenceInterval();
54      }
55
56      public double[] getThroughputConfidenceInterval(){
57          return throughput.getConfidenceInterval();
58      }
59
60      public double[] getWaitConfidenceInterval(){
61          return wait.getConfidenceInterval();
62      }
63
64      public double[] getPopulationConfidenceInterval() {return
        population.getConfidenceInterval(); }
65
66      public double[]
        getThroughputConfidenceIntervalAndAutocorrelationLagOne(){
67          return throughput.
        getConfidenceIntervalAndAutocorrelationLagOne();
68      }
69
70      public double[]
        getWaitConfidenceIntervalAndAutocorrelationLagOne(){
71          return wait.getConfidenceIntervalAndAutocorrelationLagOne()
        ;
72      }
73
74      public double[]
        getPopulationConfidenceIntervalAndAutocorrelationLagOne(){
75          return population.
        getConfidenceIntervalAndAutocorrelationLagOne();
76      }
77
78      public void resetBetweenRunsMetrics(){
79          wait.resetValue();
80          throughput.resetValue();
81          population.resetValue();
82          serviceTime.resetValue();
83      }
84
85
86      //in run attributes
87      private double node;
88      private double server;
89      private double departure;
90      private double lastArrivalTime;
91      ArrayList<Event> jobsInCenterList;
92
93
94      public ArrayList<Event> getJobsInCenterList() { return
        jobsInCenterList;}
95
96      public double getDeparture(){
97          return departure;
98      }
```

```
99
100     public void increaseDeparture(){
101         departure +=1;
102     }
103
104
105     public int getNumJobsInCenter(){
106         return jobsInCenterList.size();
107     }
108
109     public void resetInRunMetrics(){
110         departure = 0.0;
111         node = 0.0;
112         server = 0.0;
113     }
114
115     public void removeAllEventsInCenter(){
116         jobsInCenterList = new ArrayList<>();
117     }
118
119
120     public void updateInRunMetrics(double currentTime, double
        nextTime) {
121         if (jobsInCenterList.size() > 0) {
122             node = node + jobsInCenterList.size() * (nextTime -
        currentTime);
123             server = server + (nextTime - currentTime);
124         }
125     }
126
127     public Server(ServerEnum type, SchedulingDisciplineType
        discipline){
128         this.type = type;
129         this.discipline = discipline;
130         jobsInCenterList = new ArrayList<>();
131
132         departure = 0.0;
133
134
135         throughput = new BetweenRunsMetric();
136         wait = new BetweenRunsMetric();
137         population = new BetweenRunsMetric();
138         serviceTime = new BetweenRunsMetric();
139
140         node = 0.0;
141         server = 0.0;
142
143         currentBatchStartTime = 0.0;
144     }
145
146     /**
147      * Find the position of the Event event in the queue of the
        center.
148      * The queue is kept sorted based on the endTime value of the
        jobs.
149      *
150      * If the center scheduling discipline is FIFO the current
```

```
        event must be insert at the end of the queue becuase
151      * the endTime of the current event is after the endTime of the
         (old) last job in the queue.
152      *
153      * If the center scheduling discipline is PS or IS the position
         of the current event depends on the endTime value of
154      * the job: the ordering of the queue is used to apply the
        binary search to efficiently find the position in which
155      * to insert the current event.
156      * */
157     private int findPosition(Event event){
158         if(discipline == FIFO){
159             if(jobsInCenterList.size() > 0){
160                 return jobsInCenterList.size();
161             } else {
162                 return 0;
163             }
164         } else if (discipline == PS || discipline == IS) {
165             int low = 0;
166             int high = jobsInCenterList.size() - 1;
167
168             while (low <= high) {
169                 int mid = (low + high) / 2;
170                 Double midTime = jobsInCenterList.get(mid).
        getEndTime();
171                 int cmp = midTime.compareTo(event.getEndTime());
172
173                 if (cmp < 0)
174                     low = mid + 1;
175                 else if (cmp > 0)
176                     high = mid - 1;
177                 else
178                     return mid;
179             }
180             return low;
181         }
182         else {
183             System.err.println("ERRORE NELLA DISCIPLINA DI
        SCHEDULING");
184             System.exit(-1);
185             return -1;
186         }
187     }
188
189     /**
190      * Insert the job associated to the Event event in the queue.
191      * If the center scheduling discipline is FIFO, the endTime
        value of event is set.
192      * It uses the method findPosition() to find the position in
        which to insert the job.
193      * */
194     public int insertJobInCenter(Event event, double current){
195         int position = 0;
196         if(jobsInCenterList.size() == 0){
197             if(discipline == FIFO){
198                 event.setEndTime(current);
199             }
```

```
200         } else{
201             position = findPosition(event);
202             if(discipline == FIFO){
203                 event.setEndTime(jobsInCenterList.get(position-1).
    getEndTime());
204             }
205         }
206         jobsInCenterList.add(position, event);
207 
208 
209         lastArrivalTime = current;
210 
211         return position;
212     }
213 
214     private double getLastArrivalTime(){return lastArrivalTime;}
215 
216     /**
217      * Remove the head of the queue.
218      * The queue is kept ordered, so the next event is the first of
         the queue.
219      * */
220     public void removeNextEvent(){
221         if(jobsInCenterList.size()>0){
222             jobsInCenterList.remove(0);
223         }
224     }
225 
226     public Event getNextCompletation(double current){
227         if (jobsInCenterList.size()>0) {
228             return jobsInCenterList.get(0);
229         } else{
230             return null;
231         }
232     }
233 
234     /**
235      * Used only if the scheduling discipline is PS.
236      * */
237     public void updateJobsTimeAfterCompletation(Event nextEvent){
238         for(Event job : jobsInCenterList){
239             job.updateTime(nextEvent.getEndTime(), jobsInCenterList
    .size(), false);
240         }
241     }
242 
243     /**
244      * Used only if the scheduling discipline is PS.
245      * */
246     public void updateJobsTimeAfterArrival(Event nextEvent){
247         for(Event job : jobsInCenterList){
248             job.updateTime(nextEvent.getEndTime(), jobsInCenterList
    .size(), true);
249         }
250     }
251 
252     public double getAverageWait(){
```

```
253         return node/departure;
254     }
255 
256     public double getAveragePopulation(){
257         return node/lastArrivalTime;
258     }
259 
260     public double getServiceTime() {
261         if(type != ServerEnum.S3) {
262             return server / departure;
263         } else {
264             return node / departure;
265         }
266     }
267 
268     public void printBetweenRunsMetrics(){
269         DecimalFormat f = new DecimalFormat("###0.00000000000");
270         System.out.println("\nServer " +this.type + " beetwen runs
    metrics:");
271         System.out.println("   average throughput =   " + f.format(
    throughput.getSampleMean()));
272         System.out.println("   average wait =   " + f.format(wait.
    getSampleMean()));
273     }
274 
275     public void printMetrics(double current){
276         DecimalFormat f = new DecimalFormat("###0.00000000000");
277         System.out.println("\nServer " +this.type + " for " +
    departure + " jobs");
278         // dovrebbe essere lastInterarrival al posto di current
279         System.out.println("   average interarrival time =   " + f.
    format(lastArrivalTime / departure));
280         System.out.println("   average wait ............ =   " + f.
    format(node/ departure));
281         System.out.println("   average service time .... =   " + f.
    format(server/ departure));
282         System.out.println("   average # in the node ... =   " + f.
    format(node / current));
283         System.out.println("   utilization ............. =   " + f.
    format(server / current));
284         // dovrebbe essere lastInterarrival al posto di current
285         System.out.println("   lambda .................. =   " + f.
    format(departure/lastArrivalTime));
286         System.out.println("   jobs nel centro ......... =   " + f.
    format(jobsInCenterList.size()));
287         System.out.println("");
288     }
289 
290 
291     public void computeAutocorrelationValues() {
292         wait.computeAutocorrelationValues();
293         throughput.computeAutocorrelationValues();
294         population.computeAutocorrelationValues();
295     }
296 }
```

3.7 ServerEnum.java

```java
package entity;
/**
 * Lists the Server of the network
 * */
public enum ServerEnum {

    VM1("VM1", 1),
    S3("S3", 2),
    VM2CPU("VM2CPU", 3),
    VM2BAND("VM2Band", 4);

    private final String centerName;
    private final int centerIndex;

    ServerEnum(String centerName, int centerIndex) {
        this.centerName = centerName;
        this.centerIndex = centerIndex;
    }

    public int getCenterIndex(){
        return this.centerIndex;
    }

    public String getCenterName(){
        return this.centerName;
    }
}
```

4 Package Utils

4.1 AcsModified.java

```
/*
    ------------------------------------------------------------------------
 * This program is based on a one-pass algorithm for the calculation of an
 * array of autocorrelations r[1], r[2], ... r[K].  The key feature of this
 * algorithm is the circular array 'hold' which stores the (K + 1) most
 * recent data points and the associated index 'p' which points to the
 * (rotating) head of the array.
 *
    ------------------------------------------------------------------------
 */
package utils;

/**
 * Class used to compute one lag autocorrelation using a one-pass algorithm.
 * */
public class AcsModified {

    private static final int K = 1;               /* K is the maximum lag          */
    private static final int SIZE = 2;

    private int    i;                   /* data point index        */
    private int    j;                       /* lag index               */
    private int    p;                   /* points to the head of 'hold'  */
    private double x;                       /* current x[i] data point       */
    private double sum;               /* sums x[i]                */
    long   n;                        /* number of data points    */
    private double[] hold; /* K + 1 most recent data points */
    private double[] cosum; /* cosum[j] sums x[i] * x[i+j]   */


    public AcsModified(){
        i = 0;                   /* data point index         */
        p = 0;                   /* points to the head of 'hold'  */
        sum = 0.0;               /* sums x[i]                */
        hold  = new double [SIZE]; /* K + 1 most recent data points */
```

```
34          cosum = new double [SIZE]; /* cosum[j] sums x[i] * x[i+j]
        */
35      }
36
37      public void insertValue(double value){
38          x = value;
39          sum += x;
40          hold[i] = x;
41          i++;
42      }
43
44
45      public void updateValue(double value){
46          for (j = 0; j < SIZE; j++)
47              cosum[j] += hold[p] * hold[(p + j) % SIZE];
48          x       = value;
49          sum    += x;
50          hold[p] = x;
51          p       = (p + 1) % SIZE;
52          i++;
53
54
55      }
56
57
58      public double getAutocorrelationWithLagOneComputation() {
59          return cosum[1] / cosum[0];
60      }
61
62
63      public void resetValues() {
64          i = 0;                    /* data point index
        */
65          p = 0;                    /* points to the head of 'hold'
        */
66          sum = 0.0;                /* sums x[i]
        */
67          hold  = new double [SIZE]; /* K + 1 most recent data points
         */
68          cosum = new double [SIZE]; /* cosum[j] sums x[i] * x[i+j]
         */
69      }
70
71      public void computeAutocorrelationValues() {
72          /* number of data points         */
73          long n = i;
74          while (i < n + SIZE) {        /* empty the circular array
             */
75              for (j = 0; j < SIZE; j++)
76                  cosum[j] += hold[p] * hold[(p + j) % SIZE];
77              hold[p] = 0.0;
78              p       = (p + 1) % SIZE;
79              i++;
80          }
81
82          double mean = sum / n;
83          for (j = 0; j <= K; j++)
```

```
84              cosum[j] = (cosum[j] / (n - j)) - (mean * mean);
85      }
86  }
```

4.2 BetweenRunsMetric.java

```java
package utils;
import utils.AcsModified;
import utils.Rvms;

/**
 * Class that apply Welford algorithm to compute sample mean and
    variance.
 * */
public class BetweenRunsMetric {

    private static final double CONFIDENCE = 0.95;
    private static Rvms rvms = new Rvms();

    private long i;
    private double sampleMean;
    private double vi;

    private AcsModified acs;

    public BetweenRunsMetric(){
        sampleMean = 0.0;

        vi = 0.0;
        i = 0;

        acs = new AcsModified();
    }

    public void resetValue(){
        sampleMean = 0.0;

        vi = 0.0;
        i = 0;

        acs.resetValues();
    }

    public void updateMetrics(double newValue){
        i++;
        if(i <= 2){
            double diff = newValue - sampleMean;
            sampleMean = sampleMean + diff / i;
            vi = vi + diff * diff * (i - 1) / i;
            acs.insertValue(newValue);
        } else {
            double diff = newValue - sampleMean;
            sampleMean = sampleMean + diff / i;
            vi = vi + diff * diff * (i - 1) / i;
            acs.updateValue(newValue);
        }
    }

    public double getSampleMean(){
        return sampleMean;
    }
```

```
55
56
57      public double[] getConfidenceIntervalAndAutocorrelationLagOne()
        {
58          double alpha = 1-CONFIDENCE;
59          double criticalValue = rvms.idfStudent(i-1, 1 - alpha / 2);
60          double intervalWidth = criticalValue * Math.sqrt(vi / i) /
        Math.sqrt(i-1);
61
62          return new double[]{sampleMean, intervalWidth, acs.
        getAutocorrelationWithLagOneComputation()};
63      }
64
65      public double[] getConfidenceInterval(){
66          double alpha = 1-CONFIDENCE;
67          double criticalValue = rvms.idfStudent(i-1, 1 - alpha / 2);
68          double intervalWidth = criticalValue * Math.sqrt(vi / i) /
        Math.sqrt(i-1);
69          return new double[]{sampleMean, intervalWidth};
70      }
71
72      public void computeAutocorrelationValues() {
73          acs.computeAutocorrelationValues();
74      }
75  }
```

4.3 Generator.java

```
1  package utils;
2  public class Generator extends Rngs {
3
4      public Generator(){
5          super();
6      }
7
8      public double exponential(double m) {
9          /* ---------------------------------------------------
10          * generate an Exponential random variate, use m > 0.0
11          * ---------------------------------------------------
12          */
13         return (-m * Math.log(1.0 - this.random()));
14     }
15
16     public  double uniform(double a, double b) {
17         /* ------------------------------------------------
18          * generate an Uniform random variate, use a < b
19          * ------------------------------------------------
20          */
21         return (a + (b - a) * this.random());
22     }
23
24 }
```

4.4 Generator.java

```
1  package utils;
2  import entity.Server;
3  import utils.BetweenRunsMetric;
4  import static config.Params.*;
5
6  /**
7   * Class used to compute the mean price (   /min) to keep the
       system running.
8   * */
9  public class Price {
10
11
12      private BetweenRunsMetric betweenRunsVM1Utilization;
13      private BetweenRunsMetric betweenRunsS3ArrivalRate;
14      private BetweenRunsMetric betweenRunsVM2Utilization;
15      private double inRunVM1ActiveTime;
16      private int inRunS3Arrival;
17      private double inRunVM2ActiveTime;
18
19      public Price(){
20          betweenRunsVM1Utilization = new BetweenRunsMetric();
21          betweenRunsS3ArrivalRate = new BetweenRunsMetric();
22          betweenRunsVM2Utilization = new BetweenRunsMetric();
23          inRunVM1ActiveTime = 0.0;
24          inRunS3Arrival = 0;
25          inRunVM2ActiveTime = 0.0;
26      }
27
28      public void updateBetweenRunsValues(double currentBatchDuration
        ){
29          betweenRunsVM1Utilization.updateMetrics(inRunVM1ActiveTime
        /currentBatchDuration);
30          betweenRunsS3ArrivalRate.updateMetrics(inRunS3Arrival /
        currentBatchDuration);
31          betweenRunsVM2Utilization.updateMetrics(inRunVM2ActiveTime
        /currentBatchDuration);
32      }
33
34      public void updateInRunValues(double currentTime, double
        nextTime, Server[] servers, boolean isArrivalS3){
35          if(isArrivalS3){
36              inRunS3Arrival += 1;
37          }
38          double timeInterval = nextTime - currentTime;
39
40          if(servers[0].getNumJobsInCenter() > 0){
41              inRunVM1ActiveTime += timeInterval;
42          }
43
44          //VM2 is active if VM2CPU is active or VM2Band is active
45          if(servers[2].getNumJobsInCenter() > 0 || servers[3].
        getNumJobsInCenter() > 0){
46              inRunVM2ActiveTime += timeInterval;
47          }
48
```

```
49      }
50
51      public void resetInRunValues(){
52          inRunVM1ActiveTime = 0.0;
53          inRunS3Arrival = 0;
54          inRunVM2ActiveTime = 0.0;
55      }
56
57      public void resetBetweenRunsValues() {
58          betweenRunsVM1Utilization.resetValue();
59          betweenRunsS3ArrivalRate.resetValue();
60          betweenRunsVM2Utilization.resetValue();
61      }
62
63
64      public double[] getTotalNetworkPriceInterval(){
65          double[] vm1 = betweenRunsVM1Utilization.
        getConfidenceInterval();
66          double[] s3 = betweenRunsS3ArrivalRate.
        getConfidenceInterval();
67          double[] vm2 = betweenRunsVM2Utilization.
        getConfidenceInterval();
68
69          double[] result = {0, 0};
70
71          for(int i = 0; i < 2; i++){
72              result[i] = vm1[i] * VM1_PRICE_PER_MINUTE + s3[i] *
        S3_PRICE_PER_REQUEST + vm2[i] * VM2CPU_PRICE_PER_MINUTE;
73          }
74
75          return result;
76      }
77
78
79  }
```

5 Package Writer

5.1 Writer.java

```
1  package writer;
2  import java.io.BufferedWriter;
3  import java.io.FileWriter;
4  import java.io.IOException;
5  import java.io.PrintWriter;
6
7  /**
8   * Abstract Writer.
9   * */
10 public abstract class Writer {
11
12     protected String path;
13     protected PrintWriter pw;
14
15     protected void openFile() {
16         FileWriter fw;
17         try {
18             fw = new FileWriter(path, false);
19             BufferedWriter bw = new BufferedWriter(fw);
20             pw = new PrintWriter(bw);
21
22         } catch (IOException e) {
23             e.printStackTrace();
24         }
25     }
26
27     protected abstract void writeHeader();
28
29     public void writeLine(double[] values) {
30         String line = "";
31         for(int i = 0; i < values.length; i++){
32             if(i != values.length -1){
33                 line = line + values[i]+",";
34             } else {
35                 line = line + values[i];
36             }
37         }
38         pw.println(line);
39     }
40
41     public void flush(){
42         pw.flush();
43     }
44
45     public void flushAndClose(){
46         pw.flush();
47         pw.close();
48     }
49 }
```

5.2 BatchMeansWriter.java

```
1  package writer;
2
3  /**
4   * Writer for BatchMeans simulations.
5   * */
6  public class BatchMeansWriter extends Writer {
7
8      public BatchMeansWriter(char configuration, String discipline){
9          super();
10         String prefix = "configuration_" + configuration + "_" +
       discipline + "_";
11         path = prefix + "bm_metrics.csv";
12         super.openFile();
13         writeHeader();
14     }
15
16     protected void writeHeader(){
17         pw.println("Arrival rate,VM1 Mean Wait,VM1 Wait Interval
       Width,VM1 Wait Autocorrelation Lag One,"+
18                 "VM1 Mean Throughput,VM1 Throughput Interval Width,
       VM1 Throughput Autocorrelation Lag One,"+
19                 "VM1 Mean Population,VM1 Population Interval Width,
       VM1 Population Autocorrelation Lag One,"+
20                 "S3 Mean Wait,S3 Wait Interval Width,S3 Wait
       Autocorrelation Lag One,"+
21                 "S3 Mean Throughput,S3 Throughput Interval Width,S3
        Throughput Autocorrelation Lag One,"+
22                 "S3 Mean Population,S3 Population Interval Width,S3
        Population Autocorrelation Lag One,"+
23                 "VM2CPU Mean Wait,VM2CPU Wait Interval Width,VM2CPU
        Wait Autocorrelation Lag One,"+
24                 "VM2CPU Mean Throughput,VM2CPU Throughput Interval
       Width,VM2CPU Throughput Autocorrelation Lag One,"+
25                 "VM2CPU Mean Population,VM2CPU Population Interval
       Width,VM2CPU Population Autocorrelation Lag One,"+
26                 "VM2Band Mean Wait,VM2Band Wait Interval Width,
       VM2Band Wait Autocorrelation Lag One,"+
27                 "VM2Band Mean Throughput,VM2Band Throughput
       Interval Width,VM2Band Throughput Autocorrelation Lag One,"+
28                 "VM2Band Mean Population,VM2Band Population
       Interval Width,VM2Band Population Autocorrelation Lag One,"+
29                 "System Mean Wait,System Wait Interval Width,System
        Wait Autocorrelation Lag One,"+
30                 "System Mean Throughput,System Throughput Interval
       Width,System Throughput Autocorrelation Lag One,"+
31                 "System Mean Population,System Population Interval
       Width,System Population Autocorrelation Lag One,"+
32                 "Price Per Minute,Price Per Minute Interval Width")
       ;
33     }
34 }
```

5.3 DebugWriter.java

```
1  package writer;
2
3  /**
4   * Writer for Debug simulations.
5   * */
6  public class DebugWriter extends Writer{
7
8      protected void writeHeader(){
9          pw.println("discipline,configuration,lambda,
       mean_service_time_VM1,mean_service_time_S3,
       mean_service_time_VM2CPU,mean_service_time_VM2BAND,P00,P01,P02,
       P03,P04,P10,P11,P12,P13,P14,P20,P21,P22,P23,P24,P30,P31,P32,P33
       ,P34," +
10                 "P40,P41,P42,P43,P44");
11     }
12
13     public DebugWriter(){
14         super();
15         path = "debug.csv";
16         super.openFile();
17         writeHeader();
18     }
19
20     public void writeLine(String discipline, char configuration ,
       double[] values) {
21         String line = "" + discipline + ","+configuration+",";
22         for(int i = 0; i < values.length; i++){
23             if(i != values.length -1){
24                 line = line + values[i]+",";
25             } else {
26                 line = line + values[i];
27             }
28         }
29         pw.println(line);
30     }
31 }
```

5.4 FiniteHorizonWriter.java

```
1  package writer;
2
3  /**
4   * Writer for FiniteHorizon simulations.
5   * */
6  public class FiniteHorizonWriter extends Writer{
7
8
9
10     public FiniteHorizonWriter(char configuration, String
       discipline){
11         String prefix = "configuration_" + configuration + "_" +
       discipline + "_";
12         path = prefix + "fh_metrics.csv";
13         openFile();
14         writeHeader();
15     }
16
17
18     protected void writeHeader(){
19         pw.println("ITERATIONS,VM1 Mean Wait,VM1 Wait Interval
       Width,"+
20                 "VM1 Mean Throughput,VM1 Throughput Interval Width,
       "+
21                 "VM1 Mean Population,VM1 Population Interval Width,
       "+
22                 "S3 Mean Wait,S3 Wait Interval Width,"+
23                 "S3 Mean Throughput,S3 Throughput Interval Width,"+
24                 "S3 Mean Population,S3 Population Interval Width,"+
25                 "VM2CPU Mean Wait,VM2CPU Wait Interval Width,"+
26                 "VM2CPU Mean Throughput,VM2CPU Throughput Interval
       Width,"+
27                 "VM2CPU Mean Population,VM2CPU Population Interval
       Width,"+
28                 "VM2Band Mean Wait,VM2Band Wait Interval Width,"+
29                 "VM2Band Mean Throughput,VM2Band Throughput
       Interval Width,"+
30                 "VM2Band Mean Population,VM2Band Population
       Interval Width,"+
31                 "System Mean Wait,System Wait Interval Width,"+
32                 "System Mean Throughput,System Throughput Interval
       Width,"+
33                 "System Mean Population,System Population Interval
       Width," +
34                 "Price Per Minute,Price Per Minute Interval Width,
       Simulation Time");
35
36     }
37 }
```

6 Runner

6.1 Runner.java

```
1  import config.NetworkConfiguration;
2  import config.Params;
3  import debug.ArrivalsCounter;
4  import debug.RoutingMatrix;
5  import entity.*;
6  import utils.Generator;
7  import utils.Price;
8  import writer.BatchMeansWriter;
9  import writer.FiniteHorizonWriter;
10 import writer.DebugWriter;
11 import java.util.ArrayList;
12 import static entity.SchedulingDisciplineType.*;
13
14 /**It runs the simulations*/
15 public class Runner {
16
17     public static final double START = 0.0;
18     private static final SchedulingDisciplineType[] disciplines = {
       FIFO, PS};
19     private static final char[] networkConfigurationCodes = {'A', '
       B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'};
20     private static final double[] arrivalRates = {5.0, 5.1, 5.2,
       5.3, 5.4, 5.5, 5.6, 5.7, 5.8, 5.9,
21             6.0, 6.1, 6.2, 6.3, 6.4, 6.5, 6.6, 6.7, 6.8, 6.9,
22             7.0, 7.1, 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9,
23             8.0, 8.1, 8.2, 8.3, 8.4, 8.5, 8.6, 8.7, 8.8, 8.9,
24             9.0, 9.1, 9.2, 9.3, 9.4, 9.5, 9.6, 9.7, 9.8, 9.9,
25             10.0, 10.1, 10.2, 10.3, 10.4, 10.5, 10.6, 10.7, 10.8,
       10.9,11.0};
26
27     private static double currentTime;
28     private static double currentTimeLimit = 0;
29     private static Network network;
30     private static Price price;
31     public static SchedulingDisciplineType actualDiscipline;
32     private static char actualConfiguration;
33
34
35     /**
36      * Initialize the servers according to the scheduling
       discipline for the current simulation
37      * (actualDiscipline value)
38      * */
39     private static void initServers(Server[] servers){
40         Server serverVM1 = null;
41         Server serverS3 = null;
42         Server serverVM2CPU = null;
43         Server serverVM2Band = null;
44         if(actualDiscipline.equals(FIFO)) {
45             serverVM1 = new Server(ServerEnum.VM1, FIFO);
46             serverS3 = new Server(ServerEnum.S3, IS);
47             serverVM2CPU = new Server(ServerEnum.VM2CPU, FIFO);
48             serverVM2Band = new Server(ServerEnum.VM2BAND, FIFO);
```

```
49          } else if(actualDiscipline.equals(PS)){
50              serverVM1 = new Server(ServerEnum.VM1, PS);
51              serverS3 = new Server(ServerEnum.S3, IS);
52              serverVM2CPU = new Server(ServerEnum.VM2CPU, PS);
53              serverVM2Band = new Server(ServerEnum.VM2BAND, PS);
54          } else {
55              System.err.println("Errore nella gestione delle
    discipline di scheduling.");
56              System.exit(-1);
57          }
58          servers[0] = serverVM1;
59          servers[1] = serverS3;
60          servers[2] = serverVM2CPU;
61          servers[3] = serverVM2Band;
62      }
63
64
65      public static void main(String[] args) {
66
67          DebugWriter debugWriter = null;
68
69          for(SchedulingDisciplineType discipline : disciplines) {
70              actualDiscipline = discipline;
71              System.out.println("Disciplina: " + discipline);
72
73              Generator generator = new Generator();
74              NetworkConfiguration networkConfiguration =
    NetworkConfiguration.getInstance();
75
76
77              BatchMeansWriter batchMeansWriter;
78
79
80              for (char configuration : networkConfigurationCodes) {
81                  actualConfiguration = configuration;
82
83                  if (Params.DEBUG_MODE_ON) {
84                      //it opens the output file when the first
    simulation runs
85                      if(configuration == networkConfigurationCodes
    [0] && discipline == disciplines[0])
86                          debugWriter = new DebugWriter();
87
88                      network = new Network();
89                      price = new Price();
90                      networkConfiguration.setConfiguration(
    configuration);
91                      Server[] servers = new Server[4];
92                      initServers(servers);
93
94                      runDebugMode(generator, debugWriter, servers);
95                      //it close the output file when the last
    simulation runs
96                      if(configuration == networkConfigurationCodes[
    networkConfigurationCodes.length-1] && discipline.equals(
    disciplines[disciplines.length -1])) {
97                          assert debugWriter != null;
```

```
98                      debugWriter.flushAndClose();
99                  }
100             }
101
102             if (Params.runFiniteHorizonSimulation) {
103                 //for each (discipline, configuration) pair it
    opens a new file
104                 FiniteHorizonWriter writer = new
    FiniteHorizonWriter(actualConfiguration, actualDiscipline.name
    ());
105                 System.out.println("Configurazione: " +
    actualConfiguration);
106
107                 network = new Network();
108                 price = new Price();
109                 networkConfiguration.setConfiguration(
    configuration);
110                 Server[] servers = new Server[4];
111                 initServers(servers);
112
113                 runFiniteHorizonSimulation(generator, writer,
    servers);
114             }
115
116             if (Params.runBatchMeansSimulation) {
117                 //for each (discipline, configuration) pair it
    opens a new file
118                 batchMeansWriter = new BatchMeansWriter(
    actualConfiguration, actualDiscipline.name());
119                 System.out.println("Configurazione: " +
    actualConfiguration);
120
121                 network = new Network();
122                 price = new Price();
123                 networkConfiguration.setConfiguration(
    configuration);
124                 Server[] servers = new Server[4];
125                 initServers(servers);
126
127                 runBatchMeansSimulation(generator,
    batchMeansWriter, servers);
128             }
129         }
130     }
131 }
132
133
134 /**
135  * It runs a debug mode simulation.
136  * Debug mode simulation is a long finite-horizon simulation
    that collects different data.
137  * */
138 public static void runDebugMode(Generator generator,
    DebugWriter writer, Server[] servers){
139     generator.plantSeeds(123456789);
140     RoutingMatrix routingMatrix = new RoutingMatrix();
141     ArrivalsCounter arrivalsCounter = new ArrivalsCounter();
```

```
142         currentTimeLimit = Double.MAX_VALUE;
143 
144         for(int i = 0; i < Params.NUM_REPLICAS; i++) {
145             System.out.println("Configurazione: " +
    actualConfiguration + ". Replica: " + i + " ...");
146             currentTime = START;
147 
148             //currentLedger is a ledger that memorize all the
    values of currentTime
149             ArrayList<Double> currentLedger = new ArrayList<>();
150             currentLedger.add(0, currentTime);
151 
152 
153             EventList eventList = new EventList(generator,
    currentTime);
154 
155             int iterations = 0;
156             while (iterations < Params.DEBUG_ITERATIONS) {
157                 Event nextEvent = eventList.removeNextEvent();
158                 EventType nextEventType = nextEvent.getType();
159 
160                 //update arrivalsCounter or routingMatrix
161                 if (nextEventType == EventType.ARRIVALS3) {
162                     arrivalsCounter.increaseCounter(nextEventType);
163                 } else {
164                     ServerEnum nextCenter = nextEvent.getNextCenter
    ();
165                     switch (nextEventType) {
166                         case COMPLETATIONVM1:
167                             routingMatrix.increaseCounter(
    ServerEnum.VM1, nextCenter);
168                             break;
169                         case COMPLETATIONS3:
170                             routingMatrix.increaseCounter(
    ServerEnum.S3, nextCenter);
171                             break;
172                         case COMPLETATIONVM2CPU:
173                             routingMatrix.increaseCounter(
    ServerEnum.VM2CPU, nextCenter);
174                             break;
175                         case COMPLETATIONVM2BAND:
176                             routingMatrix.increaseCounter(
    ServerEnum.VM2BAND, nextCenter);
177                             break;
178                         default:
179                             System.err.println("Errore Fatale");
180                             System.exit(-1);
181                     }
182                 }
183 
184 
185                 handleEvent(nextEvent, servers, generator,
    eventList);
186                 iterations++;
187 
188                 currentLedger.add(0, currentTime);
189             }
```

```
190
191             //it checks that, for the current replica, the
    currentTime values are monotonically strictly increasing
192             for (int j = 1; j < currentLedger.size(); j++) {
193                 if (currentLedger.get(j - 1) < currentLedger.get(j)
    ) {
194                     System.err.println("Errore nell'avanzamento del
     clock.");
195                     System.err.println(j + " - " + currentLedger.
    get(j - 1) + " < " + currentLedger.get(j));
196                     System.exit(-1);
197                 }
198             }
199
200             routingMatrix.commitCounters();
201             routingMatrix.resetCounters();
202
203             arrivalsCounter.commitCounters(currentTime);
204             arrivalsCounter.resetCounters();
205
206             for (Server server : servers) {
207                 server.updateBetweenRunsMetrics(currentTime);
208                 server.removeAllEventsInCenter();
209                 server.resetInRunMetrics();
210             }
211             network.updateBetweenRunsMetrics(currentTime);
212             network.removeAllEventsInNetwork();
213             network.resetInRunMetrics();
214         }
215
216         ArrayList<Double> frequencies = routingMatrix.
    getRoutingFrequencies();
217         ArrayList<Double> counters = arrivalsCounter.getCounters();
218         double[] values = new double[30];
219         values[0] = counters.get(0);    //lambda_S3
220
221         values[1] = servers[0].getServiceTimeInterval()[0];   //
    mean_service_time_VM1
222         values[2] = servers[1].getServiceTimeInterval()[0];   //
    mean_service_time_S3
223         values[3] = servers[2].getServiceTimeInterval()[0];   //
    mean_service_time_VM2CPU
224         values[4] = servers[3].getServiceTimeInterval()[0];   //
    mean_service_time_VM2BAND
225
226         //These values are statically set because we can only have
    arrivals to S3
227         values[5] = 0;  //P00
228         values[6] = 0;  //P01
229         values[7] = 1;  //P02
230         values[8] = 0;  //P03
231         values[9] = 0;  //P04
232
233
234         for(int k = 0; k < frequencies.size(); k++){
235             values[10+k] = frequencies.get(k);
236         }
```

```
237
238         writer.writeLine(actualDiscipline.name(),
    actualConfiguration, values);
239         if(actualConfiguration == networkConfigurationCodes[
    networkConfigurationCodes.length-1]){
240             writer.flush();
241         }
242     }
243
244
245
246
247     /**
248      * It runs a finite horizon mode simulation.
249      * */
250     private static void runFiniteHorizonSimulation(Generator
    generator, FiniteHorizonWriter writer, Server[] servers){
251         currentTimeLimit = 0;
252         //the runs are done until the currentTimeLimit value reach
    Params.FH_MAX_TIME_LIMIT (=100)
253         //each run the currentTimeLimit value is increased by
    Params.FH_TIME_INCREASE_STEP (=2)
254         while(currentTimeLimit < Params.FH_MAX_TIME_LIMIT) {
255             generator.plantSeeds(123456789+(long) currentTimeLimit)
    ;
256             currentTimeLimit += Params.FH_TIME_INCREASE_STEP;
257
258
259             int iterations = 0;
260             for(int i = 0; i < Params.NUM_REPLICAS; i++) {
261
262                 currentTime = START;
263
264
265                 EventList eventList = new EventList(generator,
    currentTime);
266                 iterations = 0;
267
268                 while (currentTime < currentTimeLimit) {
269                     Event nextEvent = eventList.removeNextEvent();
270                     handleEvent(nextEvent, servers, generator,
    eventList);
271                     iterations++;
272
273                 }
274
275
276                 network.updateBetweenRunsMetrics(currentTime);
277                 network.removeAllEventsInNetwork();
278                 network.resetInRunMetrics();
279
280                 for (Server server : servers) {
281
282                     server.updateBetweenRunsMetrics(currentTime);
283                     server.removeAllEventsInCenter();
284                     server.resetInRunMetrics();
285                 }
```

```
286
287                 price.updateBetweenRunsValues(currentTime);
288                 price.resetInRunValues();
289
290             }
291
292             double[] values = new double[34];
293             values[0] = iterations;
294             int j = 0;
295             for (Server server : servers) {
296                 values[1+6*j] = server.getWaitConfidenceInterval()
    [0];
297                 values[2+6*j] = server.getWaitConfidenceInterval()
    [1];
298                 values[3+6*j] = server.
    getThroughputConfidenceInterval()[0];
299                 values[4+6*j] = server.
    getThroughputConfidenceInterval()[1];
300                 values[5+6*j] = server.
    getPopulationConfidenceInterval()[0];
301                 values[6+6*j] = server.
    getPopulationConfidenceInterval()[1];
302                 j++;
303             }
304             values[1+6*j] = network.getWaitConfidenceInterval()[0];
305             values[2+6*j] = network.getWaitConfidenceInterval()[1];
306             values[3+6*j] = network.getThroughputConfidenceInterval
    ()[0];
307             values[4+6*j] = network.getThroughputConfidenceInterval
    ()[1];
308             values[5+6*j] = network.getPopulationConfidenceInterval
    ()[0];
309             values[6+6*j] = network.getPopulationConfidenceInterval
    ()[1];
310             values[31] = price.getTotalNetworkPriceInterval()[0];
311             values[32] = price.getTotalNetworkPriceInterval()[1];
312             values[33] = currentTimeLimit;
313             writer.writeLine(values);
314             writer.flush();
315             for(Server server : servers){
316                 server.resetBetweenRunsMetrics();
317             }
318             network.resetBetweenRunsMetrics();
319             price.resetBetweenRunsValues();
320
321         }
322         writer.flushAndClose();
323     }
324
325
326     /**
327      * It runs a batch means mode simulation.
328      * */
329     private static void runBatchMeansSimulation(Generator generator
    , BatchMeansWriter writer, Server[] servers) {
330         int batchSize = Params.BM_NUM_EVENTS / Params.
    BM_NUM_BATCHES;
```

```
331         double initialArrivalRate = Params.MEAN_INTERARRIVAL_RATE;
332         /*
333          * To reduce the duration of the simulation, the used values for the arrivalRate used are differentiated
334          * according to the configuration of the network.
335          * This is done because when the network is congested and the scheduling discipline is PS, it is extremely
336          * expensive to run the simulation.
337          * */
338         for(double arrivalRate: arrivalRates) {
339             if(arrivalRate > 7.0 && (actualConfiguration == 'A' || actualConfiguration == 'D' || actualConfiguration == 'G')){
340                 continue;
341             } else if(arrivalRate <7.0 && (actualConfiguration != 'A' && actualConfiguration != 'D' && actualConfiguration != 'G')){
342                 continue;
343             }
344             generator.plantSeeds(123456789);
345             setArrivalRate(arrivalRate);
346             System.out.println("Arrival Rate = " + arrivalRate);
347 
348 
349             currentTime = START;
350             EventList eventList = new EventList(generator, currentTime);
351 
352             int numBatch = 0;
353             while (numBatch < Params.BM_NUM_BATCHES) {
354 
355                 //to correctly collect measures we need to know when the current batch is started
356                 double currentBatchStartTime = currentTime;
357                 for (Server server : servers) {
358                     server.setCurrentBatchStartTime(currentBatchStartTime);
359                 }
360                 network.setCurrentBatchStartTime(currentBatchStartTime);
361 
362                 for (int executionInBatch = 0; executionInBatch < batchSize; executionInBatch++) {
363                     Event nextEvent = eventList.removeNextEvent();
364                     handleEvent(nextEvent, servers, generator, eventList);
365                 }
366 
367 
368                 for (Server server : servers) {
369                     server.updateBetweenRunsMetrics(currentTime);
370                     server.resetInRunMetrics();
371                 }
372                 network.updateBetweenRunsMetrics(currentTime);
373                 price.updateBetweenRunsValues(currentTime - currentBatchStartTime);
374                 network.resetInRunMetrics();
375                 price.resetInRunValues();
```

```
376                 numBatch++;
377
378             }
379
380             for(Server server:servers){
381                 server.computeAutocorrelationValues();
382             }
383             network.computeAutocorrelationValues();
384
385             double[] values = new double[48];
386             values[0] = arrivalRate;
387             int j = 0;
388             for (Server server : servers) {
389                 values[1 + 9 * j] = server.
    getWaitConfidenceIntervalAndAutocorrelationLagOne()[0];
390                 values[2 + 9 * j] = server.
    getWaitConfidenceIntervalAndAutocorrelationLagOne()[1];
391                 values[3 + 9 * j] = server.
    getWaitConfidenceIntervalAndAutocorrelationLagOne()[2];
392                 values[4 + 9 * j] = server.
    getThroughputConfidenceIntervalAndAutocorrelationLagOne()[0];
393                 values[5 + 9 * j] = server.
    getThroughputConfidenceIntervalAndAutocorrelationLagOne()[1];
394                 values[6 + 9 * j] = server.
    getThroughputConfidenceIntervalAndAutocorrelationLagOne()[2];
395                 values[7 + 9 * j] = server.
    getPopulationConfidenceIntervalAndAutocorrelationLagOne()[0];
396                 values[8 + 9 * j] = server.
    getPopulationConfidenceIntervalAndAutocorrelationLagOne()[1];
397                 values[9 + 9 * j] = server.
    getPopulationConfidenceIntervalAndAutocorrelationLagOne()[2];
398                 j++;
399             }
400             values[1 + 9 * j] = network.
    getWaitConfidenceIntervalAndAutocorrelationLagOne()[0];
401             values[2 + 9 * j] = network.
    getWaitConfidenceIntervalAndAutocorrelationLagOne()[1];
402             values[3 + 9 * j] = network.
    getWaitConfidenceIntervalAndAutocorrelationLagOne()[2];
403             values[4 + 9 * j] = network.
    getThroughputConfidenceIntervalAndAutocorrelationLagOne()[0];
404             values[5 + 9 * j] = network.
    getThroughputConfidenceIntervalAndAutocorrelationLagOne()[1];
405             values[6 + 9 * j] = network.
    getThroughputConfidenceIntervalAndAutocorrelationLagOne()[2];
406             values[7 + 9 * j] = network.
    getPopulationConfidenceIntervalAndAutocorrelationLagOne()[0];
407             values[8 + 9 * j] = network.
    getPopulationConfidenceIntervalAndAutocorrelationLagOne()[1];
408             values[9 + 9 * j] = network.
    getPopulationConfidenceIntervalAndAutocorrelationLagOne()[2];
409             values[46] = price.getTotalNetworkPriceInterval()[0];
410             values[47] = price.getTotalNetworkPriceInterval()[1];
411             writer.writeLine(values);
412             writer.flush();
413
414
```

```
415             for (Server server : servers) {
416                 server.removeAllEventsInCenter();
417                 server.resetBetweenRunsMetrics();
418             }
419             network.resetBetweenRunsMetrics();
420             network.removeAllEventsInNetwork();
421             price.resetBetweenRunsValues();
422 
423         }
424         writer.flushAndClose();
425         setArrivalRate(initialArrivalRate);
426     }
427 
428     /**
429      * For batch means simulation only.
430      * */
431     private static void setArrivalRate(double arrivalRate) {
432         Params.MEAN_INTERARRIVAL_RATE = arrivalRate;
433         Params.MEAN_INTERARRIVAL_S3 = 1/arrivalRate;
434     }
435 
436     /**
437      * Handle an Event with EventType Arrival*
438      * It puts the Event event in the Server server; if the
        position of the event in the server's queue is 0, we need to
439      * insert the event in the EventList.
440      * A new Event of the same Arrival* type is created and
        inserted in the EventList.
441      * NOTE: This model has only 1 EventType Arrival* (ArrivalS3),
        but this method is more general than that.
442      * */
443     private static void handleArrival(Server server, Event event,
        EventList eventList, Generator generator, EventType
        newEventType){
444         Event newEvent = new Event(newEventType, generator,
        currentTime, server.getNumJobsInCenter(), server.getDiscipline
        ());
445 
446         int position = server.insertJobInCenter(newEvent,
        currentTime);
447         network.insertJobInNetwork(currentTime);
448 
449         if(position == 0){
450             eventList.putEvent(newEvent.getType(), newEvent);
451         }
452 
453         Event newArrival = new Event(event.getType(), generator,
        currentTime, server.getNumJobsInCenter(), server.getDiscipline
        ());
454         eventList.putEvent(newArrival.getType(), newArrival);
455     }
456 
457     /**
458      * Insert the Event event in the Server destination, if the
        position of the event in the destination queue is 0
459      * we need to insert the event in the EventList.
460      * */
```

```
461     private static void routeTo(Event event, Server destination,
        EventList eventList){
462         int position = destination.insertJobInCenter(event,
        currentTime);
463         if(position == 0){
464             eventList.putEvent(event.getType(), event);
465         }
466     }
467
468
469     /**
470      * Handle an Event with EventType Completation*
471      * */
472     private static void handleCompletation(Server server, Event
        event, Generator generator, EventList eventList, Server[]
        servers){
473         //It removes the event from the server queue and, if the
        queue is not empty, it update the EventList.
474         server.removeNextEvent();
475         Event newEvent = server.getNextCompletation(currentTime);
476         if(newEvent != null) {
477             eventList.putEvent(newEvent.getType(), newEvent);
478         }
479         server.increaseDeparture();
480
481         //In handles the routing.
482         ServerEnum nextCenter = event.getNextCenter();
483         if(nextCenter == null){
484             server.increaseExitCounter();
485             network.increaseDeparture();
486         } else {
487             switch (nextCenter) {
488                 case VM1:
489                     Server vm1 = servers[ServerEnum.VM1.
        getCenterIndex()-1];
490                     newEvent = new Event(EventType.COMPLETATIONVM1,
         generator, currentTime, vm1.getNumJobsInCenter(), vm1.
        getDiscipline());
491                     if(vm1.getDiscipline() == PS)
492                         vm1.updateJobsTimeAfterArrival(event);
493                     routeTo(newEvent, vm1, eventList);
494                     break;
495                 case S3:
496                     Server s3 = servers[ServerEnum.S3.
        getCenterIndex()-1];
497                     newEvent = new Event(EventType.COMPLETATIONS3,
        generator, currentTime, s3.getNumJobsInCenter(), s3.
        getDiscipline());
498                     routeTo(newEvent, s3, eventList);
499                     break;
500                 case VM2CPU:
501                     Server vm2cpu = servers[ServerEnum.VM2CPU.
        getCenterIndex()-1];
502                     newEvent = new Event(EventType.
        COMPLETATIONVM2CPU, generator, currentTime, vm2cpu.
        getNumJobsInCenter(), vm2cpu.getDiscipline());
503                     if(vm2cpu.getDiscipline() == PS)
```

```
                    vm2cpu.updateJobsTimeAfterArrival(event);
                routeTo(newEvent, vm2cpu, eventList);
                break;
            case VM2BAND:
                Server vm2band = servers[ServerEnum.VM2BAND.
getCenterIndex()-1];
                newEvent = new Event(EventType.
COMPLETATIONVM2BAND, generator, currentTime, vm2band.
getNumJobsInCenter(), vm2band.getDiscipline());
                if(vm2band.getDiscipline() == PS)
                    vm2band.updateJobsTimeAfterArrival(event);
                routeTo(newEvent, vm2band, eventList);
                break;
            default:
                System.err.println("ERRORE FATALE");
                System.exit(-1);
                break;
        }
    }
}

/**
 * It handles an event: it update the system state and the
clock and it calls handleArrival or handleCompletation.
 * */
private static void handleEvent(Event event, Server[] servers,
Generator generator, EventList eventList){
    double nextEventTime = event.getEndTime();

    for(Server server: servers) {
        server.updateInRunMetrics(currentTime, nextEventTime);
    }
    network.updateInRunMetrics(currentTime, nextEventTime);
    price.updateInRunValues(currentTime,nextEventTime, servers,
 event.getType()==EventType.ARRIVALS3);

    currentTime = nextEventTime;

    Server vm1 = servers[ServerEnum.VM1.getCenterIndex()-1];
    Server s3 = servers[ServerEnum.S3.getCenterIndex()-1];
    Server vm2cpu = servers[ServerEnum.VM2CPU.getCenterIndex()
-1];
    Server vm2band = servers[ServerEnum.VM2BAND.getCenterIndex
()-1];

    switch(event.getType()){
        case ARRIVALS3:
            handleArrival(s3, event, eventList, generator,
EventType.COMPLETATIONS3);
            break;
        case COMPLETATIONVM1:
            if(vm1.getDiscipline() == PS)
                vm1.updateJobsTimeAfterCompletation(event);
            handleCompletation(vm1, event, generator, eventList
, servers);
            break;
        case COMPLETATIONS3:
```

```
551                 handleCompletation(s3, event, generator, eventList,
        servers);
552                 break;
553             case COMPLETATIONVM2CPU:
554                 if(vm2cpu.getDiscipline() == PS)
555                    vm2cpu.updateJobsTimeAfterCompletation(event);
556                 handleCompletation(vm2cpu, event, generator,
      eventList, servers);
557                 break;
558             case COMPLETATIONVM2BAND:
559                 if(vm2band.getDiscipline() == PS)
560                     vm2band.updateJobsTimeAfterCompletation(event);
561                 handleCompletation(vm2band, event, generator,
      eventList, servers);
562                 break;
563             default:
564                 System.err.println("ERRORE FATALE");
565                 System.exit(-1);
566                 break;
567         }
568     }
569 }
```