COMP9334
Capacity Planning of Computer Systems and Networks
Project Report

z3447294
Li LI

Contents

1 Simulation Program

This project simulates a fork-join multi-server system with a pre-processor by Java. This part will be divided into four steps to demonstrate how I work out this program and its correctness.

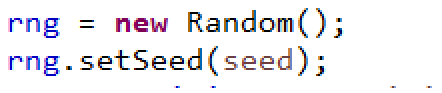
(1) Input

The input is mainly based on Exponentially distribution, Uniform Distribution and Pareto Distribution with parameters required in specification. Through these distributions, random interval time and service time can be obtained for arrival event and service event at preprocessor and multi-servers. Specific requirements of distributions and parameters settings can be obtained in project specification and will not be repeated here.

In terms of input, what should be clear includes three points here.

a. Random Number

Java provides a class called Random with a customized seed to generate random number for reproducibility.



```
rng = new Random();
rng.setSeed(seed);
```

Figure1 Random

b. Inverse Transform Method

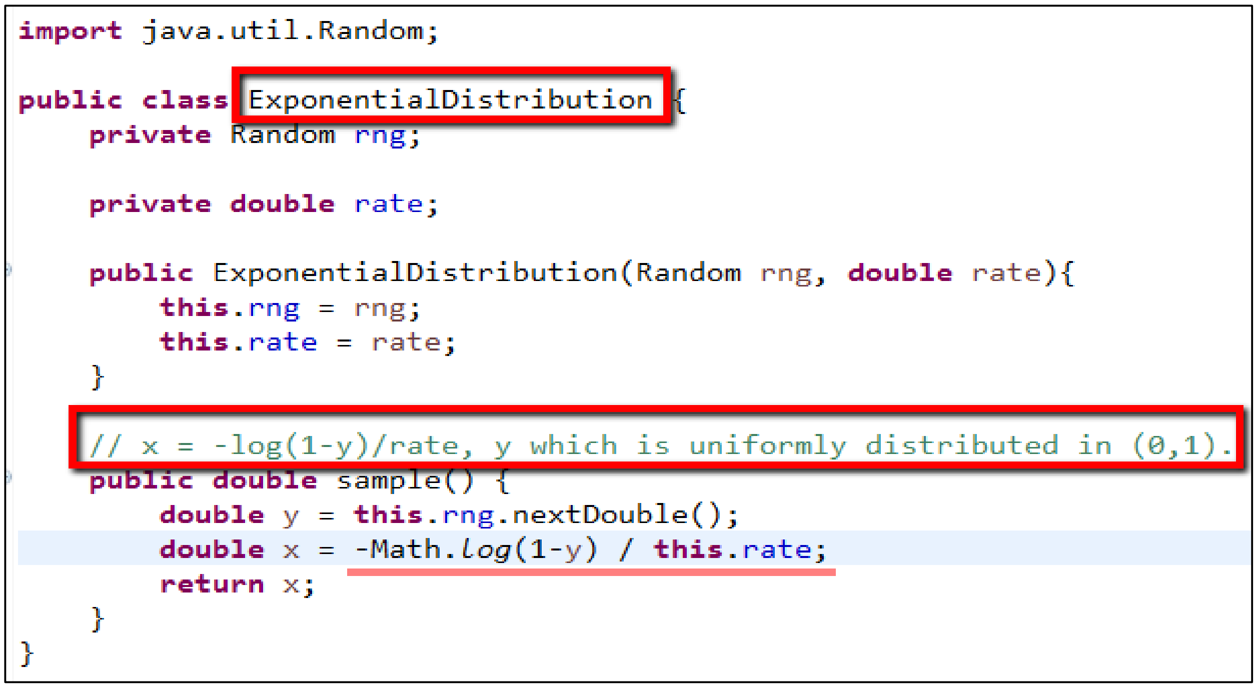
Inverse transform method can be utilized to generate random event time from a particular distribution. Assume a cumulative density function F(x) = P[X <= x],

Two steps:

1) generate a number u which is uniformly distributed in (0,1)

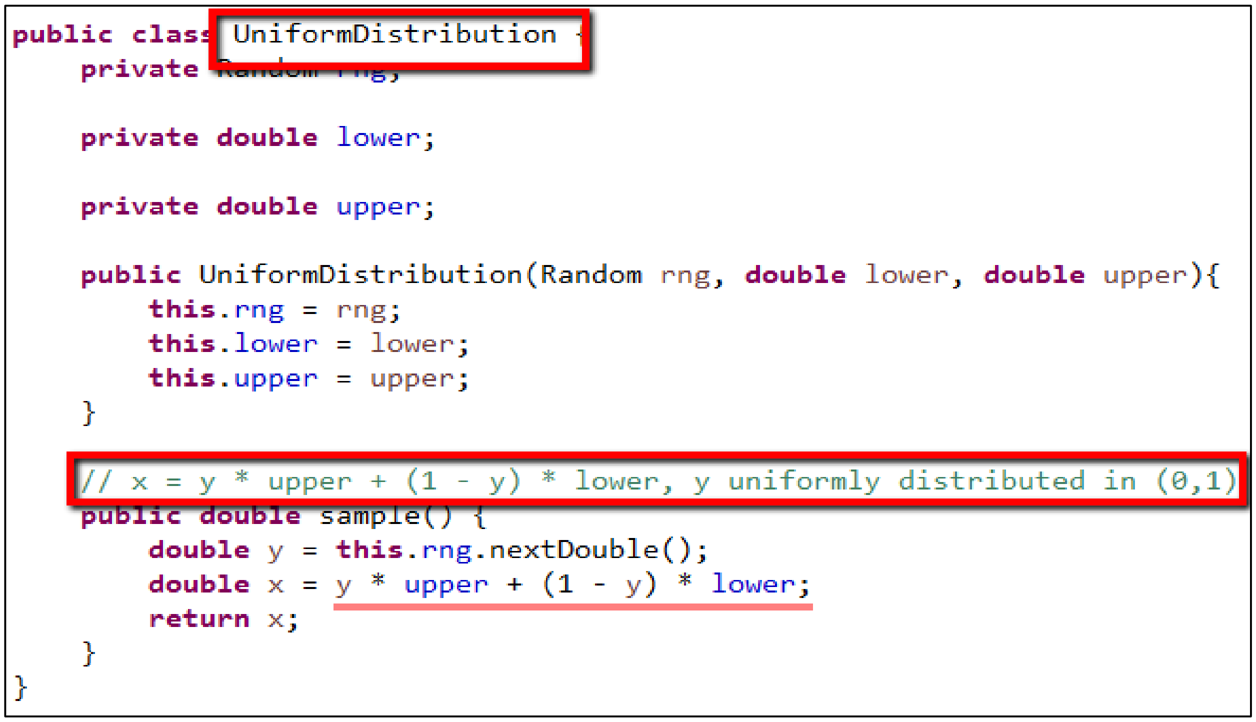
2) compute the number $F^{-1}(u)$

This can be done on paper at first and then expressed using Java.



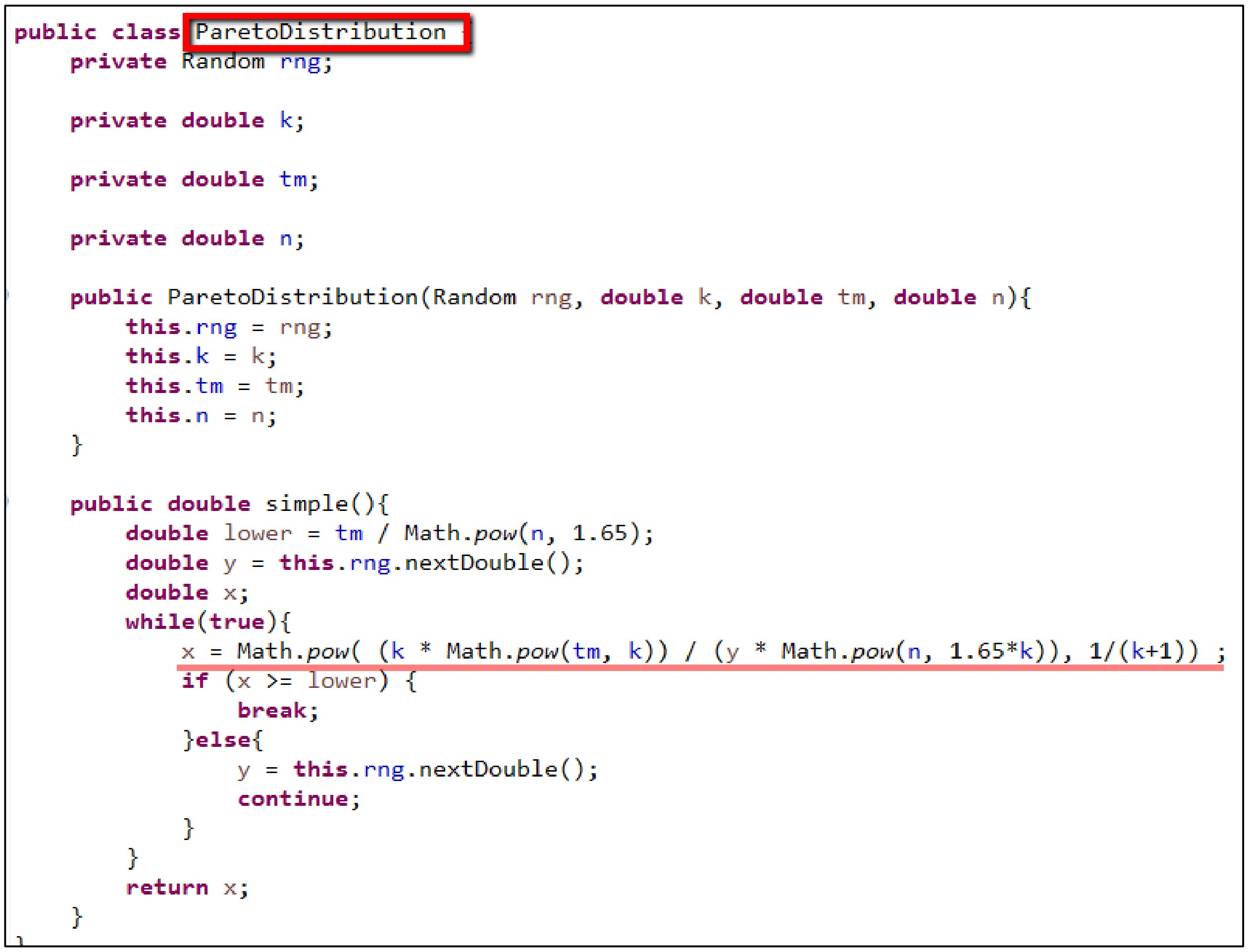
```java
import java.util.Random;

public class ExponentialDistribution {
    private Random rng;

    private double rate;

    public ExponentialDistribution(Random rng, double rate){
        this.rng = rng;
        this.rate = rate;
    }

    // x = -log(1-y)/rate, y which is uniformly distributed in (0,1).
    public double sample() {
        double y = this.rng.nextDouble();
        double x = -Math.log(1-y) / this.rate;
        return x;
    }
}
```

Figure2 Exponential Distribution



```java
public class UniformDistribution {
    private Random rng;

    private double lower;

    private double upper;

    public UniformDistribution(Random rng, double lower, double upper){
        this.rng = rng;
        this.lower = lower;
        this.upper = upper;
    }

    // x = y * upper + (1 - y) * lower, y uniformly distributed in (0,1)
    public double sample() {
        double y = this.rng.nextDouble();
        double x = y * upper + (1 - y) * lower;
        return x;
    }
}
```

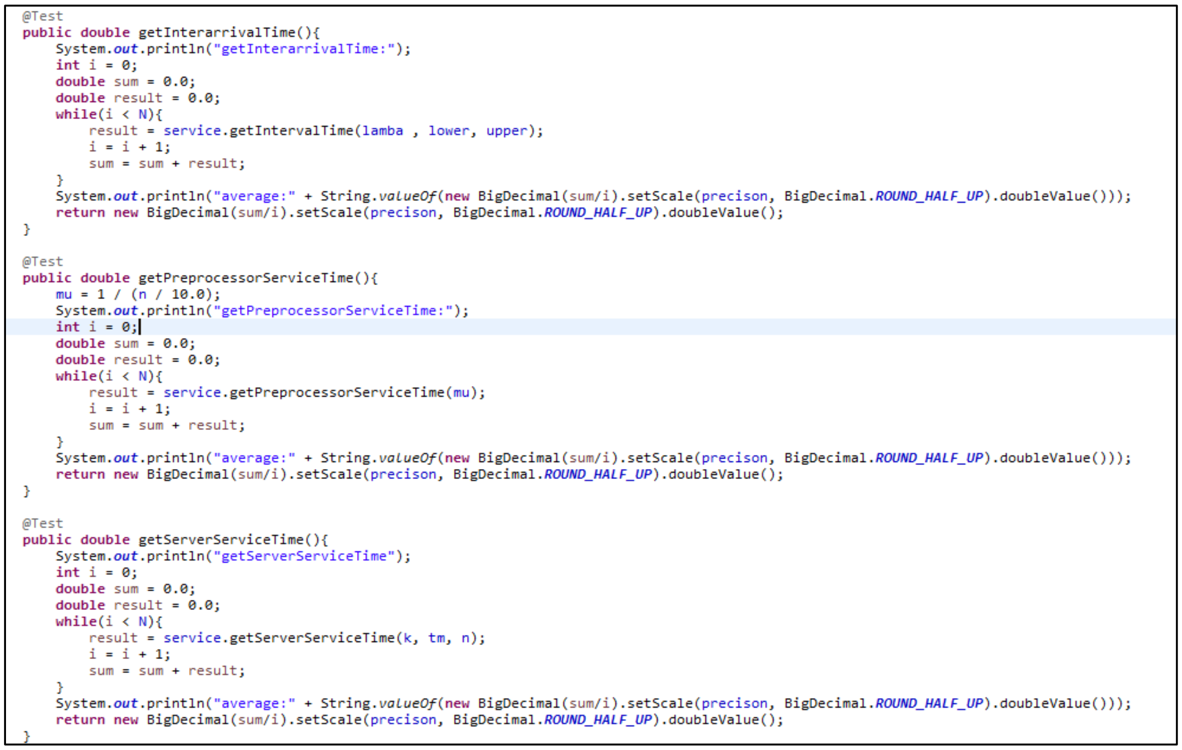
Figure3 Uniform Distribution



```
public class ParetoDistribution {
    private Random rng;

    private double k;

    private double tm;

    private double n;

    public ParetoDistribution(Random rng, double k, double tm, double n){
        this.rng = rng;
        this.k = k;
        this.tm = tm;
        this.n = n;
    }

    public double simple(){
        double lower = tm / Math.pow(n, 1.65);
        double y = this.rng.nextDouble();
        double x;
        while(true){
            x = Math.pow( (k * Math.pow(tm, k)) / (y * Math.pow(n, 1.65*k)), 1/(k+1)) ;
            if (x >= lower) {
                break;
            }else{
                y = this.rng.nextDouble();
                continue;
            }
        }
        return x;
    }
}
```

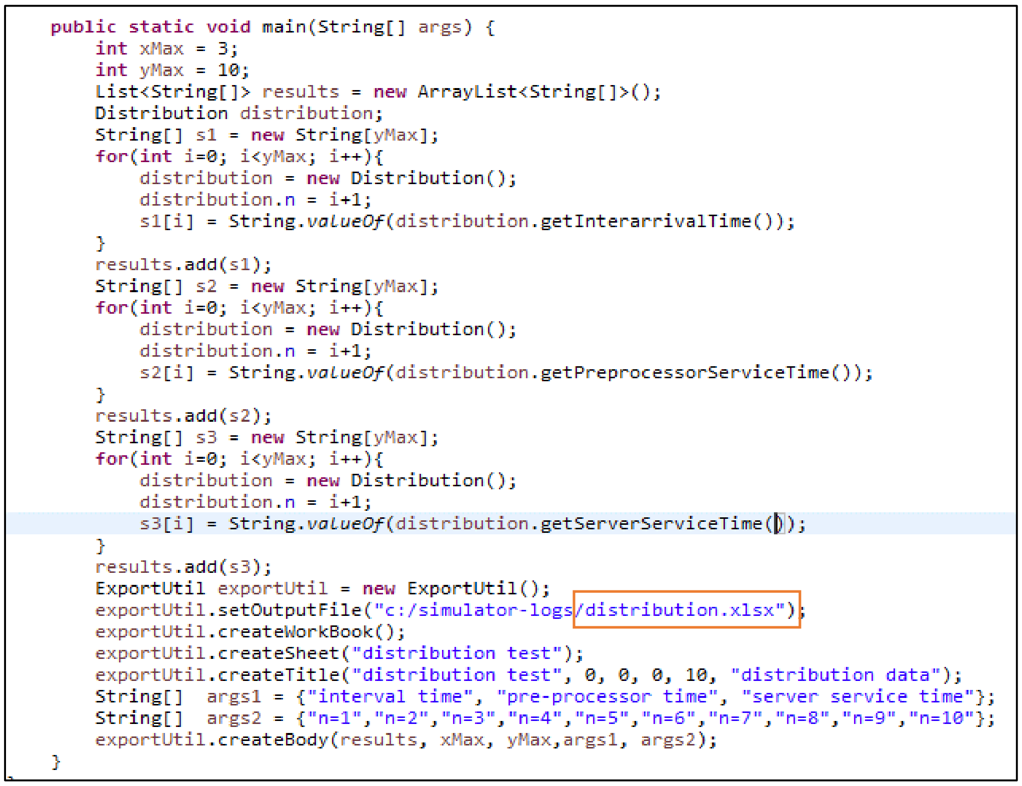
Figure4 Pareto Distribution

c. Correctness



```
@Test
public double getInterarrivalTime(){
    System.out.println("getInterarrivalTime:");
    int i = 0;
    double sum = 0.0;
    double result = 0.0;
    while(i < N){
        result = service.getIntervalTime(lamba , lower, upper);
        i = i + 1;
        sum = sum + result;
    }
    System.out.println("average:" + String.valueOf(new BigDecimal(sum/i).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue()));
    return new BigDecimal(sum/i).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue();
}

@Test
public double getPreprocessorServiceTime(){
    mu = 1 / (n / 10.0);
    System.out.println("getPreprocessorServiceTime:");
    int i = 0;
    double sum = 0.0;
    double result = 0.0;
    while(i < N){
        result = service.getPreprocessorServiceTime(mu);
        i = i + 1;
        sum = sum + result;
    }
    System.out.println("average:" + String.valueOf(new BigDecimal(sum/i).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue()));
    return new BigDecimal(sum/i).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue();
}

@Test
public double getServerServiceTime(){
    System.out.println("getServerServiceTime");
    int i = 0;
    double sum = 0.0;
    double result = 0.0;
    while(i < N){
        result = service.getServerServiceTime(k, tm, n);
        i = i + 1;
        sum = sum + result;
    }
    System.out.println("average:" + String.valueOf(new BigDecimal(sum/i).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue()));
    return new BigDecimal(sum/i).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue();
}
```

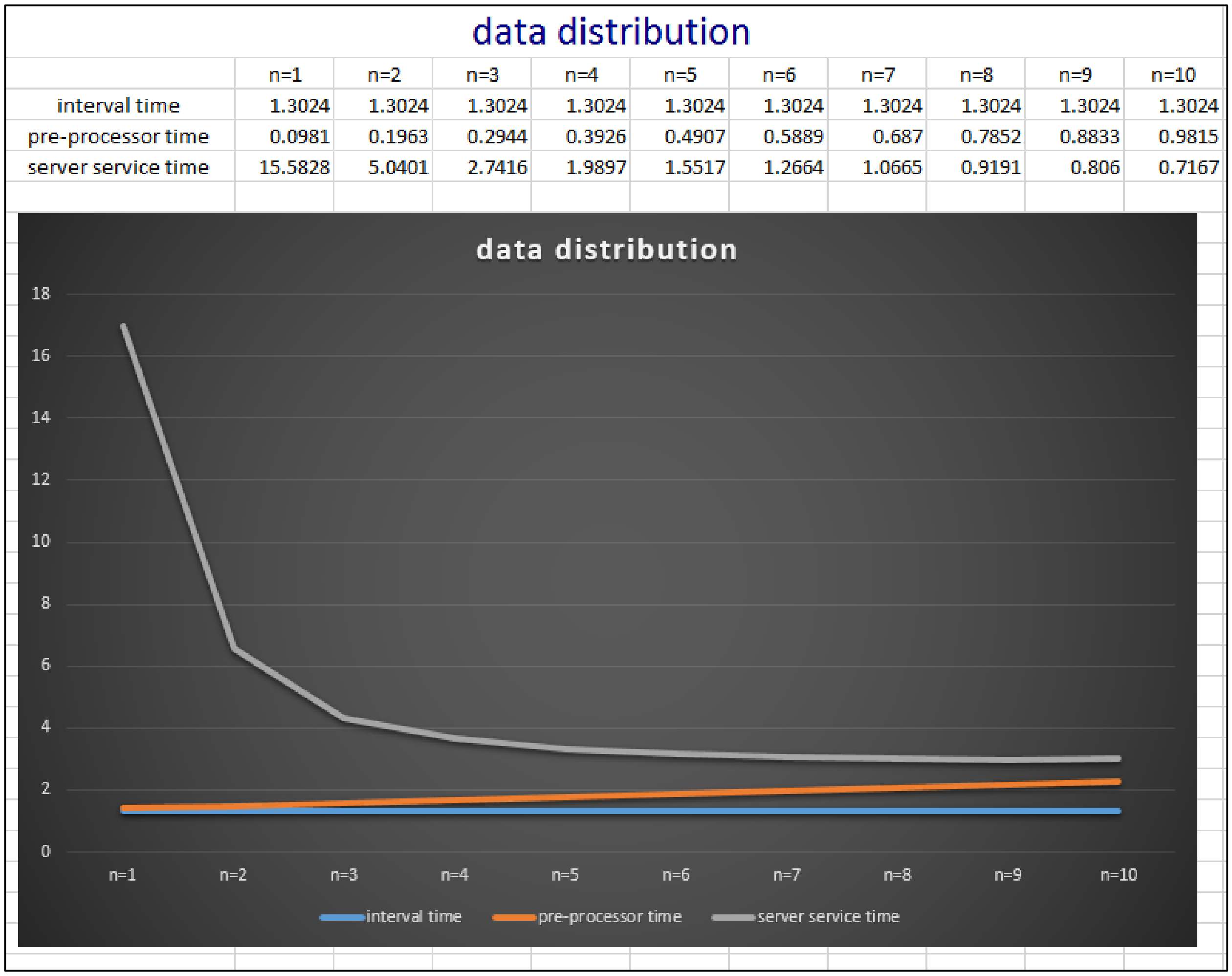
Figure5 code1, Test cases, N = 1000



```
public static void main(String[] args) {
    int xMax = 3;
    int yMax = 10;
    List<String[]> results = new ArrayList<String[]>();
    Distribution distribution;
    String[] s1 = new String[yMax];
    for(int i=0; i<yMax; i++){
        distribution = new Distribution();
        distribution.n = i+1;
        s1[i] = String.valueOf(distribution.getInterarrivalTime());
    }
    results.add(s1);
    String[] s2 = new String[yMax];
    for(int i=0; i<yMax; i++){
        distribution = new Distribution();
        distribution.n = i+1;
        s2[i] = String.valueOf(distribution.getPreprocessorServiceTime());
    }
    results.add(s2);
    String[] s3 = new String[yMax];
    for(int i=0; i<yMax; i++){
        distribution = new Distribution();
        distribution.n = i+1;
        s3[i] = String.valueOf(distribution.getServerServiceTime());
    }
    results.add(s3);
    ExportUtil exportUtil = new ExportUtil();
    exportUtil.setOutputFile("c:/simulator-logs/distribution.xlsx");
    exportUtil.createWorkBook();
    exportUtil.createSheet("distribution test");
    exportUtil.createTitle("distribution test", 0, 0, 0, 10, "distribution data");
    String[]  args1 = {"interval time", "pre-processor time", "server service time"};
    String[]  args2 = {"n=1","n=2","n=3","n=4","n=5","n=6","n=7","n=8","n=9","n=10"};
    exportUtil.createBody(results, xMax, yMax,args1, args2);
}
```

Figure6 code2

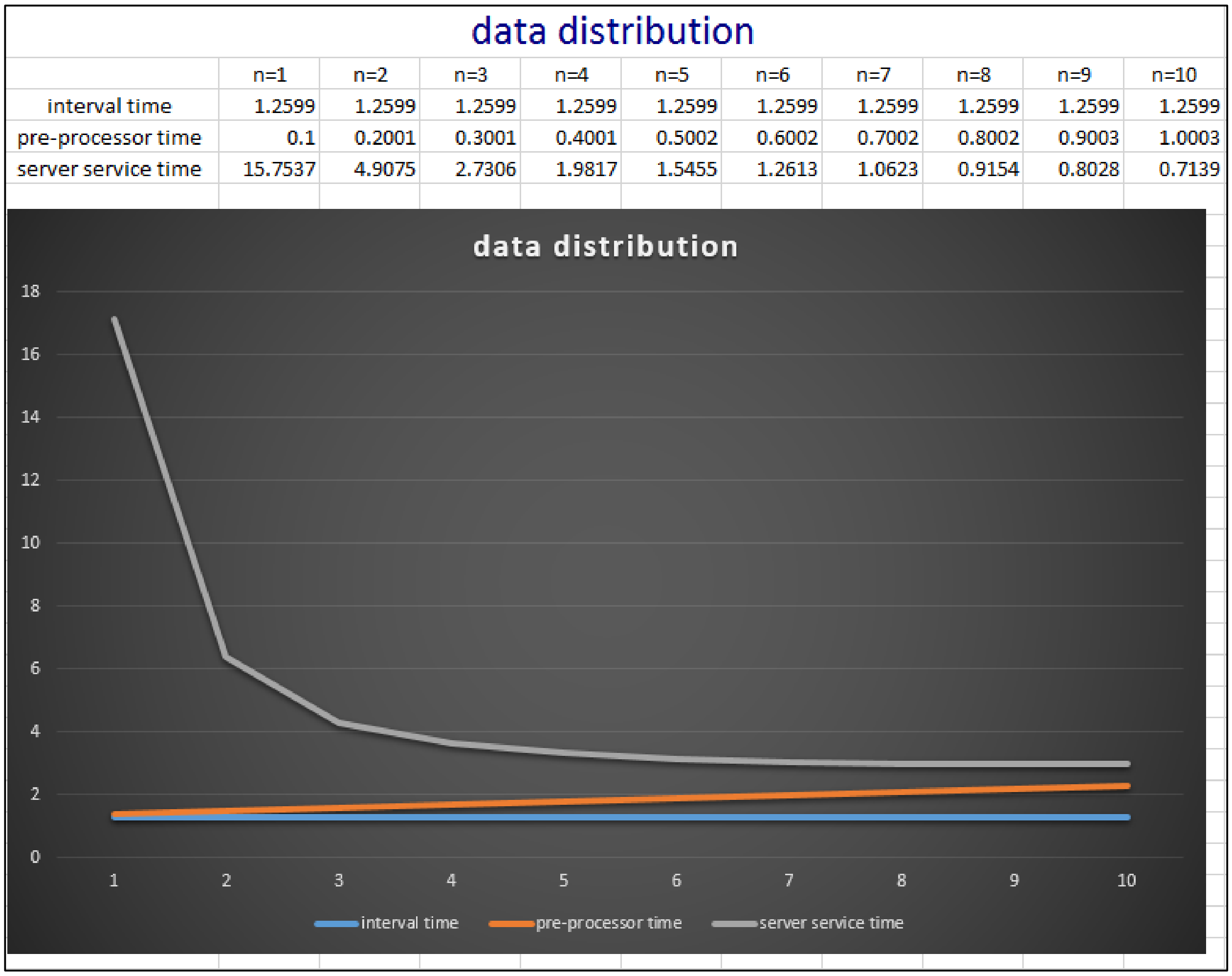
By the above code in test class **Distribution**, the trend of generated random numbers for different events can be indicated in two following example with seed 0 and 1. Assume max number of servers is 10, changes of pre-processor time and server service time could happen as n becomes larger and other parameters keep the same.



data distribution

	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
interval time	1.3024	1.3024	1.3024	1.3024	1.3024	1.3024	1.3024	1.3024	1.3024	1.3024
pre-processor time	0.0981	0.1963	0.2944	0.3926	0.4907	0.5889	0.687	0.7852	0.8833	0.9815
server service time	15.5828	5.0401	2.7416	1.9897	1.5517	1.2664	1.0665	0.9191	0.806	0.7167

Figure7 trend of random time of three events in “**distribution(seed0).xlsx**”


data distribution

	n=1	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9	n=10
interval time	1.2599	1.2599	1.2599	1.2599	1.2599	1.2599	1.2599	1.2599	1.2599	1.2599
pre-processor time	0.1	0.2001	0.3001	0.4001	0.5002	0.6002	0.7002	0.8002	0.9003	1.0003
server service time	15.7537	4.9075	2.7306	1.9817	1.5455	1.2613	1.0623	0.9154	0.8028	0.7139

Figure8 trend of random time of three events in “**distribution(seed1).xlsx”**

In the specification, it is clear that if requests are broken into many sub-tasks it requires more processing time but subsequently each sub-tasks requires less server service time if n is bigger. As indicated in the figure, the result could satisfy expectation even with different seed.

Tips: Example data can be regenerated by running code2 main function and exported to an excel file named distribution.

(2) Output

According to specification, what should be output is mean response time for each simulation.

mean response time = accumulative response time / completed requests

When master clock is larger than desired simulation time this formula in the code will be invoked. Before executing this formula, accumulative response time should be done.

What can trigger the simulation to record accumulative response time? It can be discussed in the next part.

(3) Design

In order to build a feasible simulation, according to **Harry Perros (2009)**, event-advance design could be considered. The figure9 below is from his book named **Computer Simulation Techniques: The definitive introduction**.

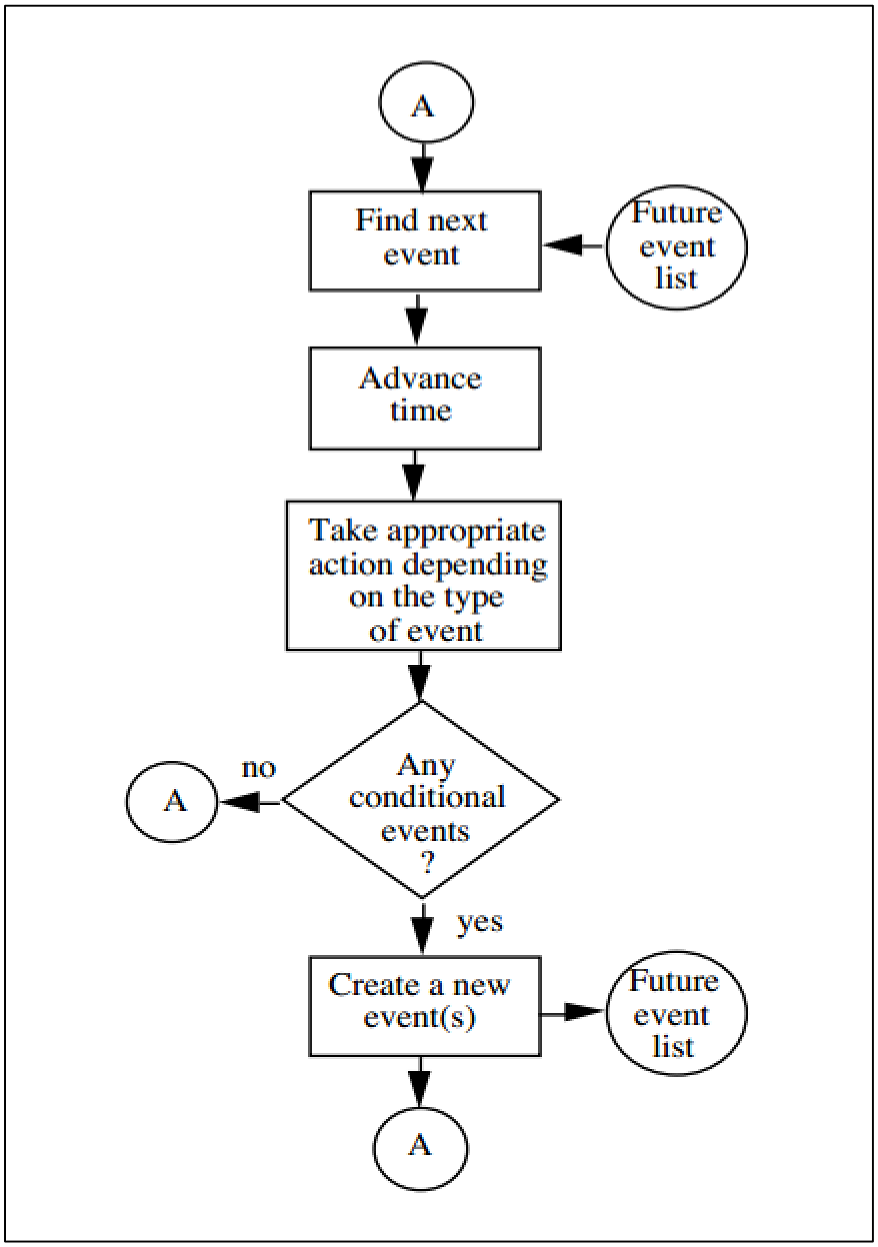


Figure9 Event Advance Simulation Design

The above logic in event-advance design could be turned into Java class **Controller** to guide the whole process. This can be done by four steps.

a. **Identify Event Type**

As we all know, this simulation can be divided into several events because there are arrivals, processing and departures at pre-processor or servers. However, not all of them make sense in terms of simplicity.

Personally, due to interval incoming requests at different time within simulation, it can be as **arrival event for next request**. After each process at pre-processor, a request is broken into n sub-tasks, which would be regarded as **departure event for next request.** Subsequently, these sub-tasks may be handled by selected servers or remained in the server buffers at a certain time; moreover, generally, in each state, there could be always one task that could be finished firstly among the tasks being processed by servers, which can be defined as **departure event for next task.**

These three above-mentioned events are enough to clarify simulation with conditional events. **Here there is only one conditional event named <u>task-joined event</u> according to project specification.** After all sub-tasks of a request finished by servers arrive at join station, it is acknowledged that this request has been done, namely, response time of this request can be calculated and accumulated. In this case, the condition of this event is that all sub-tasks of a request have to be done.

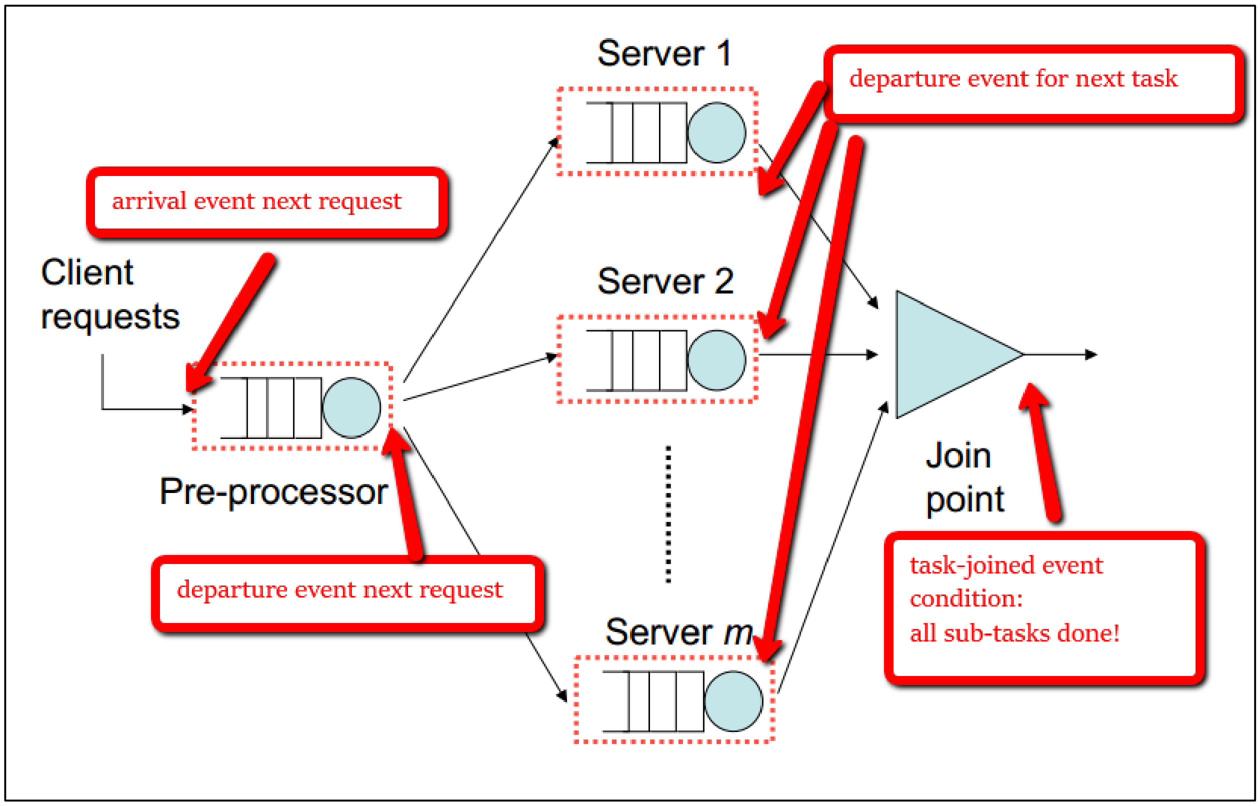

Server 1
departure event for next task
arrival event next request
Client requests
Server 2
Pre-processor
Join point
departure event next request
task-joined event condition: all sub-tasks done!
Server *m*

Figure10 event type

(red part is my work and the background of picture is from project specification)

b. Identify Event Time

Given that event types have been made correctly, corresponding event time of each event could be solved easily.

arrival time next request: obtain arrival time when a new request comes

departure time next request: obtain departure time when a request has been processed at pre-processor

departure time next task: obtain departure time when a task is done at a certain server

task-joined time: it is equal to departure time next task which should be the last finished task of a request

c. Identify Actions for Each Event Type

Table1 Event Action

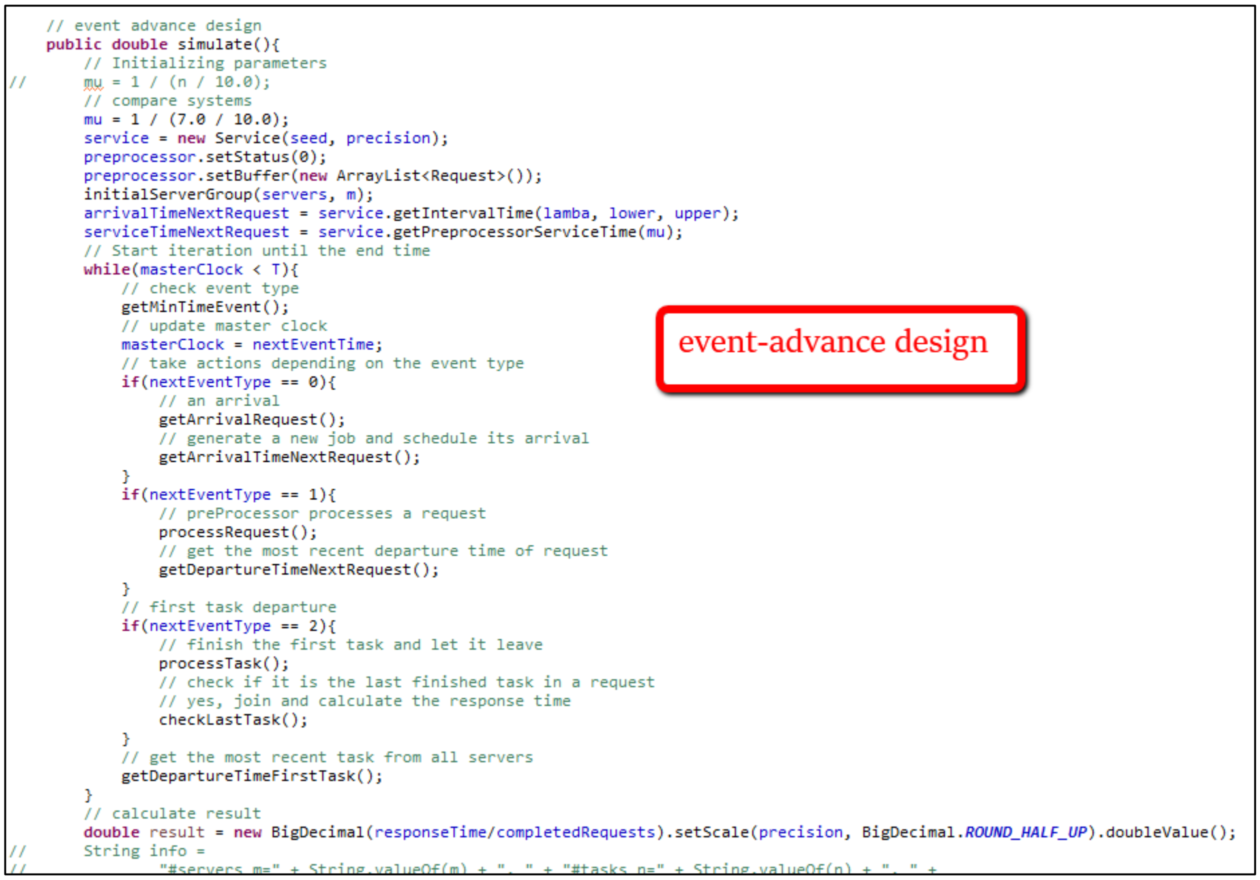
Event Type	Post-actions
arrival event for next request	• saved in buffer, if preprocessor is busy, or marked as current request, if preprocessor is idle; • advance master clock with arrival time
departure event for next request	• receive pre-processor service and advance master clock with preprocessor service time; • pre-processor service: break a request into n sub-tasks; send sub-tasks to selected servers: task is saved in server buffer if server is busy, or it is marked as a current task ready to be processed if server is idle.
departure event for next task	• receive service and advance master clock with server service time • check if it is the last finished task yes -> **task-joined event**
task-joined event (conditional)	• accumulate response time

d. Include Timeline

On a timeline, a variable named master clock can be used to keep track of the current time. It can be advanced from event to event. When master clock is still less than simulation time, the simulation would continue.

e. Put Everything Together

As discussed in part one and part two, input and output have been clear. What we need to do now is to make it clear and logical. The code in Figure11 is the core part in the program.



```
    // event advance design
    public double simulate(){
        // Initializing parameters
//      mu = 1 / (n / 10.0);
        // compare systems
        mu = 1 / (7.0 / 10.0);
        service = new Service(seed, precision);
        preprocessor.setStatus(0);
        preprocessor.setBuffer(new ArrayList<Request>());
        initialServerGroup(servers, m);
        arrivalTimeNextRequest = service.getIntervalTime(lamba, lower, upper);
        serviceTimeNextRequest = service.getPreprocessorServiceTime(mu);
        // Start iteration until the end time
        while(masterClock < T){
            // check event type
            getMinTimeEvent();
            // update master clock
            masterClock = nextEventTime;
            // take actions depending on the event type
            if(nextEventType == 0){
                // an arrival
                getArrivalRequest();
                // generate a new job and schedule its arrival
                getArrivalTimeNextRequest();
            }
            if(nextEventType == 1){
                // preProcessor processes a request
                processRequest();
                // get the most recent departure time of request
                getDepartureTimeNextRequest();
            }
            // first task departure
            if(nextEventType == 2){
                // finish the first task and let it leave
                processTask();
                // check if it is the last finished task in a request
                // yes, join and calculate the response time
                checkLastTask();
            }
            // get the most recent task from all servers
            getDepartureTimeFirstTask();
        }
        // calculate result
        double result = new BigDecimal(responseTime/completedRequests).setScale(precision, BigDecimal.ROUND_HALF_UP).doubleValue();
//      String info =
//              "#servers m=" + String.valueOf(m) + ", " + "#tasks n=" + String.valueOf(n) + ", " +
```

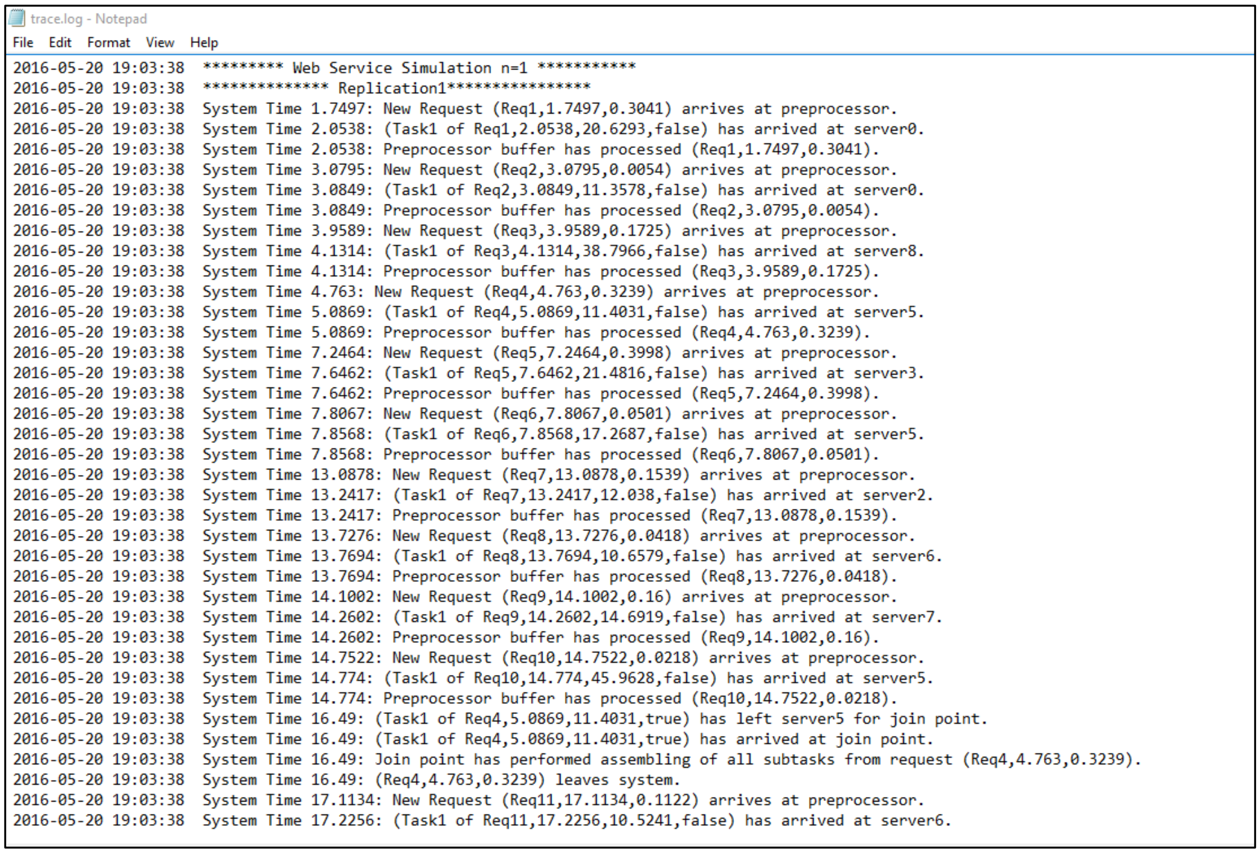
Figure11 event type

(4) Correctness

In real world, events could be sequential or concurrent at each state. What we need to do in the simulation is to identify and sort them. If they happen one after another, we capture it as a list; if they happen at the same moment, we capture, mark and process it sequentially as well but only adjust master clock only once. Moreover, input is from effective distributions proved by math and what should be output is rather clear. Based on such principles, the logic of the program can be correctly work out. Therefore, theoretically, the logic my simulation program is correct.

In fact, all operations of running simulation can be **traced** into a log file in my program. Therefore, it is convenient to verify each steps in the end. The part content of an example is shown below.

Tips: Generally, “trace” will increase running time substantially due to I/O operations. It is suggested that trace function should be off. This can be done by changing the value of property **trace** of class **Controller** to false.



```
2016-05-20 19:03:38  ********* Web Service Simulation n=1 ***********
2016-05-20 19:03:38  ************** Replication1****************
2016-05-20 19:03:38  System Time 1.7497: New Request (Req1,1.7497,0.3041) arrives at preprocessor.
2016-05-20 19:03:38  System Time 2.0538: (Task1 of Req1,2.0538,20.6293,false) has arrived at server0.
2016-05-20 19:03:38  System Time 2.0538: Preprocessor buffer has processed (Req1,1.7497,0.3041).
2016-05-20 19:03:38  System Time 3.0795: New Request (Req2,3.0795,0.0054) arrives at preprocessor.
2016-05-20 19:03:38  System Time 3.0849: (Task1 of Req2,3.0849,11.3578,false) has arrived at server0.
2016-05-20 19:03:38  System Time 3.0849: Preprocessor buffer has processed (Req2,3.0795,0.0054).
2016-05-20 19:03:38  System Time 3.9589: New Request (Req3,3.9589,0.1725) arrives at preprocessor.
2016-05-20 19:03:38  System Time 4.1314: (Task1 of Req3,4.1314,38.7966,false) has arrived at server8.
2016-05-20 19:03:38  System Time 4.1314: Preprocessor buffer has processed (Req3,3.9589,0.1725).
2016-05-20 19:03:38  System Time 4.763: New Request (Req4,4.763,0.3239) arrives at preprocessor.
2016-05-20 19:03:38  System Time 5.0869: (Task1 of Req4,5.0869,11.4031,false) has arrived at server5.
2016-05-20 19:03:38  System Time 5.0869: Preprocessor buffer has processed (Req4,4.763,0.3239).
2016-05-20 19:03:38  System Time 7.2464: New Request (Req5,7.2464,0.3998) arrives at preprocessor.
2016-05-20 19:03:38  System Time 7.6462: (Task1 of Req5,7.6462,21.4816,false) has arrived at server3.
2016-05-20 19:03:38  System Time 7.6462: Preprocessor buffer has processed (Req5,7.2464,0.3998).
2016-05-20 19:03:38  System Time 7.8067: New Request (Req6,7.8067,0.0501) arrives at preprocessor.
2016-05-20 19:03:38  System Time 7.8568: (Task1 of Req6,7.8568,17.2687,false) has arrived at server5.
2016-05-20 19:03:38  System Time 7.8568: Preprocessor buffer has processed (Req6,7.8067,0.0501).
2016-05-20 19:03:38  System Time 13.0878: New Request (Req7,13.0878,0.1539) arrives at preprocessor.
2016-05-20 19:03:38  System Time 13.2417: (Task1 of Req7,13.2417,12.038,false) has arrived at server2.
2016-05-20 19:03:38  System Time 13.2417: Preprocessor buffer has processed (Req7,13.0878,0.1539).
2016-05-20 19:03:38  System Time 13.7276: New Request (Req8,13.7276,0.0418) arrives at preprocessor.
2016-05-20 19:03:38  System Time 13.7694: (Task1 of Req8,13.7694,10.6579,false) has arrived at server6.
2016-05-20 19:03:38  System Time 13.7694: Preprocessor buffer has processed (Req8,13.7276,0.0418).
2016-05-20 19:03:38  System Time 14.1002: New Request (Req9,14.1002,0.16) arrives at preprocessor.
2016-05-20 19:03:38  System Time 14.2602: (Task1 of Req9,14.2602,14.6919,false) has arrived at server7.
2016-05-20 19:03:38  System Time 14.2602: Preprocessor buffer has processed (Req9,14.1002,0.16).
2016-05-20 19:03:38  System Time 14.7522: New Request (Req10,14.7522,0.0218) arrives at preprocessor.
2016-05-20 19:03:38  System Time 14.774: (Task1 of Req10,14.774,45.9628,false) has arrived at server5.
2016-05-20 19:03:38  System Time 14.774: Preprocessor buffer has processed (Req10,14.7522,0.0218).
2016-05-20 19:03:38  System Time 16.49: (Task1 of Req4,5.0869,11.4031,true) has left server5 for join point.
2016-05-20 19:03:38  System Time 16.49: (Task1 of Req4,5.0869,11.4031,true) has arrived at join point.
2016-05-20 19:03:38  System Time 16.49: Join point has performed assembling of all subtasks from request (Req4,4.763,0.3239).
2016-05-20 19:03:38  System Time 16.49: (Req4,4.763,0.3239) leaves system.
2016-05-20 19:03:38  System Time 17.1134: New Request (Req11,17.1134,0.1122) arrives at preprocessor.
2016-05-20 19:03:38  System Time 17.2256: (Task1 of Req11,17.2256,10.5241,false) has arrived at server6.
```

Figure12 log file

2 Analysis

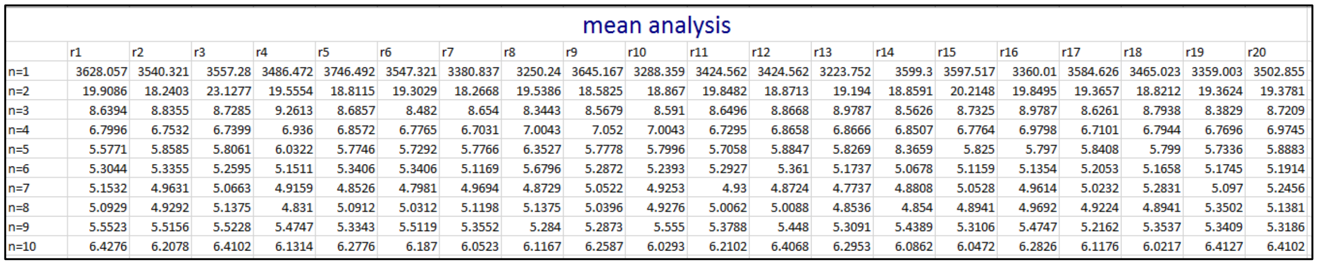
(1) Warm-up

The objective of this simulation is to solve out the value of n to be used in reference system to give it the smallest mean response time.

At the beginning, a few parameters should be determined now. Assume,

- m = 10, number of servers
- 0 < n <= m, number of selected servers
- lower = 0.05, upper = 0.25, bound of **Uniformly distribution**
- λ = 0.85, arrival rate, related to **Exponential distribution**
- μ = 1 / (n/10), service rate, related to **Exponential distribution**
- t_m = 10.3846, k = 2.08, n, related to **Pareto distribution**

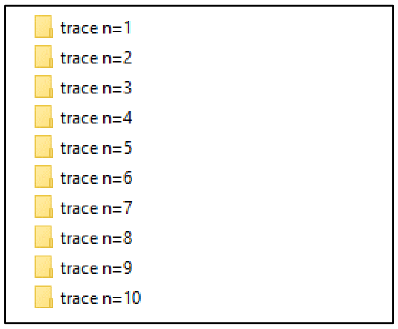
Basically by trial-and-error, the relatively accurate result could be obtained. Therefore, a rough analysis should be considered at first. **Through previous experience, assume the length of simulation time is 50000 units** and **number of independent replications is 20 each value of n**.



mean analysis

	r1	r2	r3	r4	r5	r6	r7	r8	r9	r10	r11	r12	r13	r14	r15	r16	r17	r18	r19	r20
n=1	3628.057	3540.321	3557.28	3486.472	3746.492	3547.321	3380.837	3250.24	3645.167	3288.359	3424.562	3424.562	3223.752	3599.3	3597.517	3360.01	3584.626	3465.023	3359.003	3502.855
n=2	19.9086	18.2403	23.1277	19.5554	18.8115	19.3029	18.2668	19.5386	18.5825	18.867	19.8482	18.8713	19.194	18.8591	20.2148	19.8495	19.3657	18.8212	19.3624	19.3781
n=3	8.6394	8.8355	8.7285	9.2613	8.6857	8.482	8.654	8.3443	8.5679	8.591	8.6496	8.8668	8.9787	8.5626	8.7325	8.9787	8.6261	8.7938	8.3829	8.7209
n=4	6.7996	6.7532	6.7399	6.936	6.8572	6.7765	6.7031	7.0043	7.052	7.0043	6.7295	6.8658	6.8666	6.8507	6.7764	6.9798	6.7101	6.7944	6.7696	6.9745
n=5	5.5771	5.8585	5.8061	6.0322	5.7746	5.7292	5.7766	6.3527	5.7778	5.7996	5.7058	5.8847	5.8269	8.3659	5.825	5.797	5.8408	5.799	5.7336	5.8883
n=6	5.3044	5.3355	5.2595	5.1511	5.3406	5.3406	5.1169	5.6796	5.2872	5.2393	5.2927	5.361	5.1737	5.0678	5.1159	5.1354	5.2053	5.1658	5.1745	5.1914
n=7	5.1532	4.9631	5.0663	4.9159	4.8526	4.7981	4.9694	4.8729	5.0522	4.9253	4.93	4.8724	4.7737	4.8808	5.0528	4.9614	5.0232	5.2831	5.097	5.2456
n=8	5.0929	4.9292	5.1375	4.831	5.0912	5.0312	5.1198	5.1375	5.0396	4.9276	5.0062	5.0088	4.8536	4.854	4.8941	4.9692	4.9224	4.8941	5.3502	5.1381
n=9	5.5523	5.5156	5.5228	5.4747	5.3343	5.5119	5.3552	5.284	5.2873	5.555	5.3788	5.448	5.3091	5.4389	5.3106	5.4747	5.2162	5.3537	5.3409	5.3186
n=10	6.4276	6.2078	6.4102	6.1314	6.2776	6.187	6.0523	6.1167	6.2587	6.0293	6.2102	6.4068	6.2953	6.0862	6.0472	6.2826	6.1176	6.0217	6.4127	6.4102

Figure13 mean response time analysis **without transient removal**



trace n=1
trace n=2
trace n=3
trace n=4
trace n=5
trace n=6
trace n=7
trace n=8
trace n=9
trace n=10

Figure14 20 independent replications for each value of n

As shown in the Figure13, there is a big difference from other mean values when n is equal 1 or 2. Obviously, they could be possible to be neglected due to a large distinction.

However, there is another factor leading to make mistakes that the length of simulation time may not be longer than the transient because transient removal has not been done yet.

To be further, **30000 data points** of each independent experiments of each value of n are output in the folder **replications**. With the help of week08_q1_a.m in week8 lecture notes, **batch transient removal** can be done with w=8000 for n between 1 and 10 except n = 5 with w=18000. The visual steady state can lead to the same decision to neglect the case of n=1 and n=2 as well after transient removal.

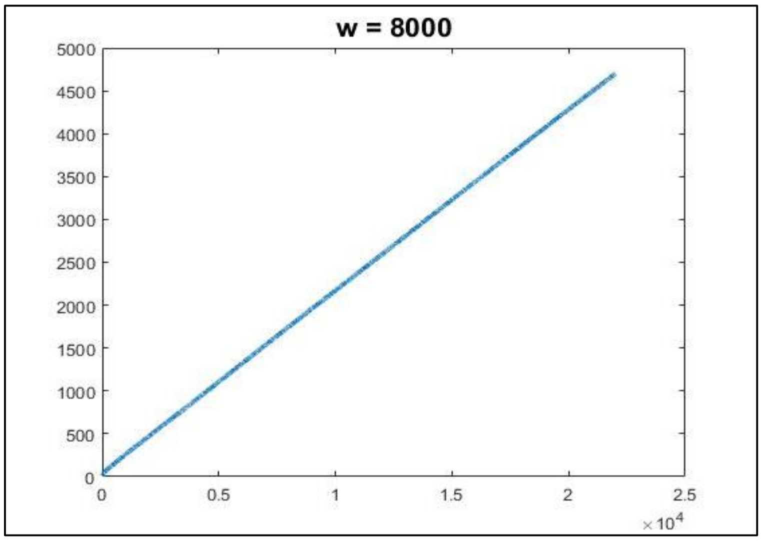


Figure15 response time and data points, **n=1**

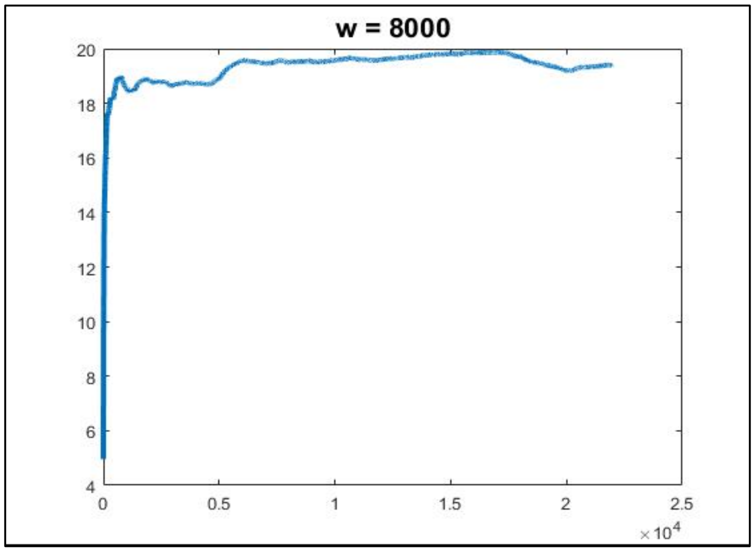
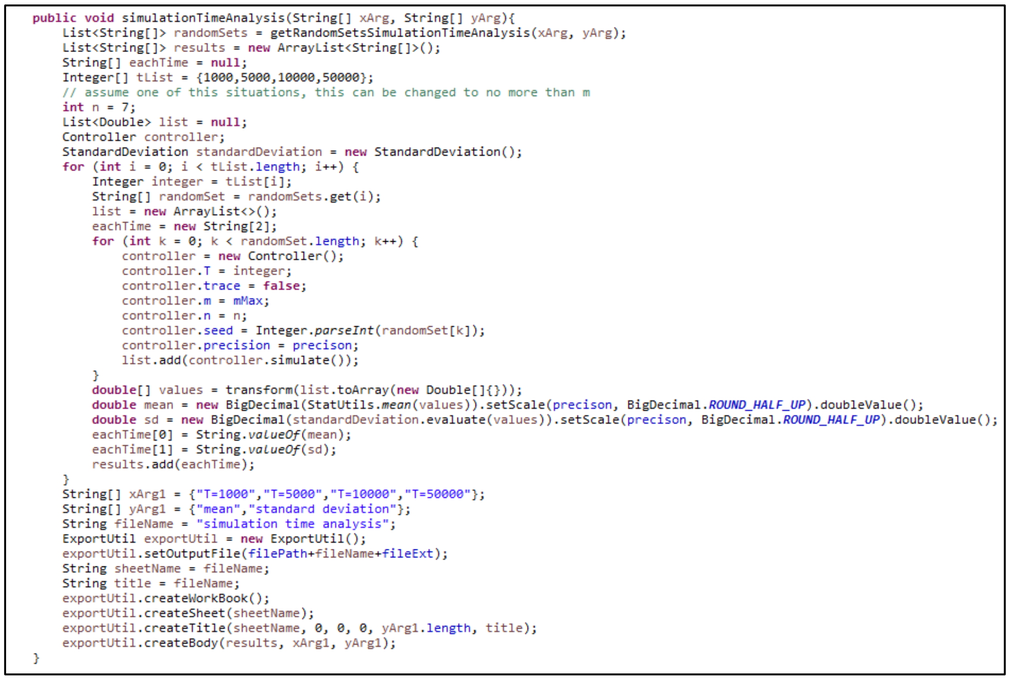


Figure16 response time and data points, **n=2**

(2) Determine Length of Simulation

Now, n has been narrowed to the range between 3 and 10. Next step is to determine the length of simulation time. Generally, if time is longer, data would be more accurate after transient removal. So far, 50000 has been used as simulation time, which has been longer than transient part in last section trial. However, it is better to obtain data than approximate visual test.

In the program, T is used to represent the length of simulation and there is a function named **simulationTimeAnalysis** in class **Analysis** which performs 20 simulation replications for each value of T. When this function runs, there will be two generated file named **simulation time analysis.xlsx** and **st-random sets.xlsx.**

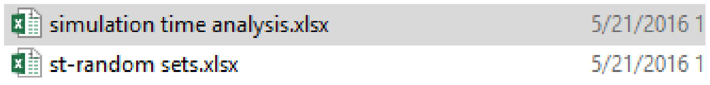


```java
public void simulationTimeAnalysis(String[] xArg, String[] yArg){
    List<String[]> randomSets = getRandomSetsSimulationTimeAnalysis(xArg, yArg);
    List<String[]> results = new ArrayList<String[]>();
    String[] eachTime = null;
    Integer[] tList = {1000,5000,10000,50000};
    // assume one of this situations, this can be changed to no more than m
    int n = 7;
    List<Double> list = null;
    Controller controller;
    StandardDeviation standardDeviation = new StandardDeviation();
    for (int i = 0; i < tList.length; i++) {
        Integer integer = tList[i];
        String[] randomSet = randomSets.get(i);
        list = new ArrayList<>();
        eachTime = new String[2];
        for (int k = 0; k < randomSet.length; k++) {
            controller = new Controller();
            controller.T = integer;
            controller.trace = false;
            controller.m = mMax;
            controller.n = n;
            controller.seed = Integer.parseInt(randomSet[k]);
            controller.precision = precison;
            list.add(controller.simulate());
        }
        double[] values = transform(list.toArray(new Double[]{}));
        double mean = new BigDecimal(StatUtils.mean(values)).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue();
        double sd = new BigDecimal(standardDeviation.evaluate(values)).setScale(precison, BigDecimal.ROUND_HALF_UP).doubleValue();
        eachTime[0] = String.valueOf(mean);
        eachTime[1] = String.valueOf(sd);
        results.add(eachTime);
    }
    String[] xArg1 = {"T=1000","T=5000","T=10000","T=50000"};
    String[] yArg1 = {"mean","standard deviation"};
    String fileName = "simulation time analysis";
    ExportUtil exportUtil = new ExportUtil();
    exportUtil.setOutputFile(filePath+fileName+fileExt);
    String sheetName = fileName;
    String title = fileName;
    exportUtil.createWorkBook();
    exportUtil.createSheet(sheetName);
    exportUtil.createTitle(sheetName, 0, 0, 0, yArg1.length, title);
    exportUtil.createBody(results, xArg1, yArg1);
}
```

Figure17 function **simulationTimeAnalysis**

Table2 simulation time analysis, n = 7

simulation time analysis		
	mean	standard deviation
T=1000	4.8128	0.3213
T=5000	4.9354	0.2966
T=10000	4.9879	0.1547
T=50000	4.9758	0.1104



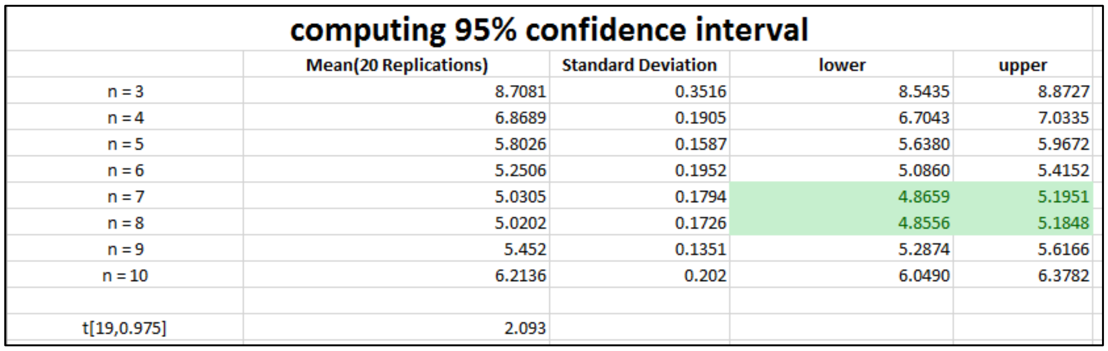
simulation time analysis.xlsx	5/21/2016 1
st-random sets.xlsx	5/21/2016 1

Figure18 simulation time analysis files

As demonstrated in table2, when n = 7, standard deviation is around 0.1104. The data of n=7 is just an example. Other values of n are also tested to prove T=50000 is good enough to capture steady state. Combined with the trial result in warm-up, T=50000 should be reliable. In addition to the accuracy, program would not run too long as well.

(3) Determine Number of Replications

In this part, due to previous trial with batch transient removal, data of 20 replications is proved to be able to compute confidence interval for n to determine number of replication. The content of file "**confidence interval.xlsx"** is created by obtaining data from running test_a.m and test_b.m in folder **replications** with the help of excel function.



computing 95% confidence interval

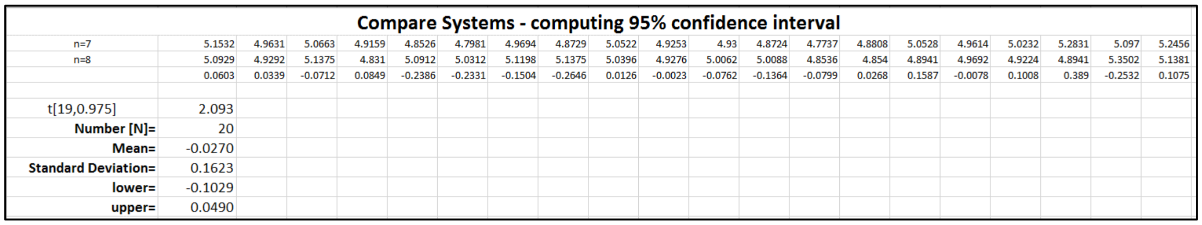
	Mean(20 Replications)	Standard Deviation	lower	upper
n = 3	8.7081	0.3516	8.5435	8.8727
n = 4	6.8689	0.1905	6.7043	7.0335
n = 5	5.8026	0.1587	5.6380	5.9672
n = 6	5.2506	0.1952	5.0860	5.4152
n = 7	5.0305	0.1794	4.8659	5.1951
n = 8	5.0202	0.1726	4.8556	5.1848
n = 9	5.452	0.1351	5.2874	5.6166
n = 10	6.2136	0.202	6.0490	6.3782
t[19,0.975]	2.093			

Figure19 95% confidence interval in "**confidence interval.xlsx"**

The width of confidence interval can satisfy my desired accuracy as indicated in Figure19. Therefore, there is no need to increase the number of replications. The green part shows that n=7 or n=8 could be the best choice when there are 10 servers.

(4) Compare Systems

As discussed in last part, n=7 or n=8 could make the response time smallest. In this part, the goal is to **compare System1(n=7) and System2(n=8).** The first file named "**compare systems not CNR.xlsx"** is created to compute confidence interval.



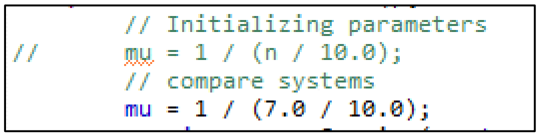
Compare Systems - computing 95% confidence interval

n=7	5.1532	4.9631	5.0663	4.9159	4.8526	4.7981	4.9694	4.8729	5.0522	4.9253	4.93	4.8724	4.7737	4.8808	5.0528	4.9614	5.0232	5.2831	5.097	5.2456
n=8	5.0929	4.9292	5.1375	4.831	5.0912	5.0312	5.1198	5.1375	5.0396	4.9276	5.0062	5.0088	4.8536	4.854	4.8941	4.9692	4.9224	4.8941	5.3502	5.1381
	0.0603	0.0339	-0.0712	0.0849	-0.2386	-0.2331	-0.1504	-0.2646	0.0126	-0.0023	-0.0762	-0.1364	-0.0799	0.0268	0.1587	-0.0078	0.1008	0.389	-0.2532	0.1075
t[19,0.975]	2.093																			
Number [N]=	20																			
Mean=	-0.0270																			
Standard Deviation=	0.1623																			
lower=	-0.1029																			
upper=	0.0490																			

Figure20 95% confidence interval of comparing systems in "**compare systems not CNR.xlsx"**

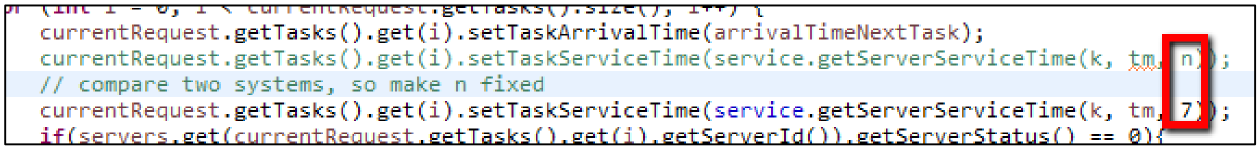
The confidence interval is: [-0.1029, 0.0490]. Therefore, with 95% probability that the mean response times of two systems are not different, but they cannot be distinguished.

In order to distinguish two systems, CNR method has to be utilized. This can be done by running function **compareSystemsCNR** with the same random set and service rate. Before simulating, there are a few modifications in the program so that **the same arrival time, preprocessor service time and server service time sequence can be generated for system1 and system2**, that is to say, two systems can have similar conditions.



```
        // Initializing parameters
//      mu = 1 / (n / 10.0);
        // compare systems
        mu = 1 / (7.0 / 10.0);
```

Figure21 modification1, control preprocessor time



```
for (int i = 0; i < currentRequest.getTasks().size(); i++) {
  currentRequest.getTasks().get(i).setTaskArrivalTime(arrivalTimeNextTask);
  currentRequest.getTasks().get(i).setTaskServiceTime(service.getServerServiceTime(k, tm, n));
  // compare two systems, so make n fixed
  currentRequest.getTasks().get(i).setTaskServiceTime(service.getServerServiceTime(k, tm, 7));
  if(servers.get(currentRequest.getTasks().get(i).getServerId()).getServerStatus() == 0){
```

Figure22 modification2, control server service time

After modifications, function **compareSystemsCNR** runs out 5, 10 and 20 replications when n=7 and n=8. An excel file named "**compare systems CNR**" is created. Using data in this file can get the 95% confidence interval by excel functions. In the end, there are three files renamed "**compare systems CNR5**", "**compare systems CNR10**" and "**compare systems CNR20**" in the folder. From three files, a 95% confidence interval tables below can be obtained.

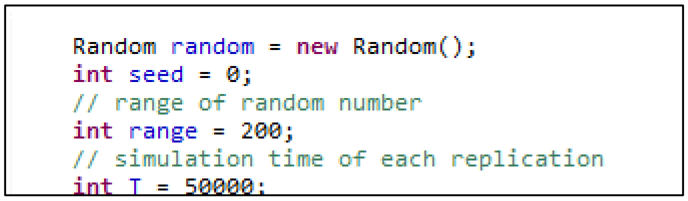
Table3 Computing Confidence Interval by CNR

# independent replications	95% Confidence interval of EMRT System2 – EMRT System1
5	[0.5938, 0.8092]
10	[0.3670, 1.9780]
20	[0.7538, 0.8515]

Overall, by using CNR, all 95% confidence interval does not include 0. Therefore, System1 is better than System2 with probability 95%; in other words, **n=7 can make the reference system have the smallest mean response time.**

3 Reproducibility

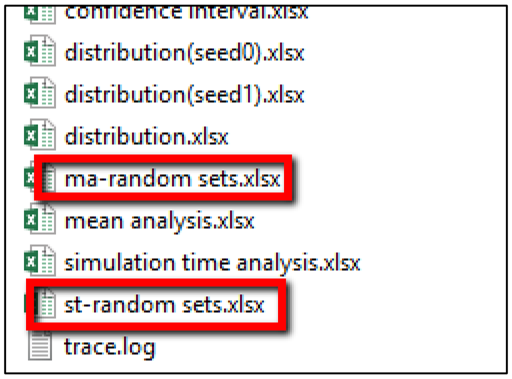
Reproducibility is based on seed in Java. As long as seed is remained, the same data could be regenerated. In this program, seed is 0. Therefore, the sequence of random numbers is controlled by seed 0.



```
Random random = new Random();
int seed = 0;
// range of random number
int range = 200;
// simulation time of each replication
int T = 50000;
```

Figure23 seed = 0

Moreover, all random numbers are stored into the corresponding files so that we can check if this program has reproducibility. Actually, the sequence of random numbers in two files are the same, but it can be checked just in case of an accident.



```
confidence interval.xlsx
distribution(seed0).xlsx
distribution(seed1).xlsx
distribution.xlsx
ma-random sets.xlsx
mean analysis.xlsx
simulation time analysis.xlsx
st-random sets.xlsx
trace.log
```

Figure24 files store random sets

4 References

[1] Law and Kelton, “Simulation modelling and analysis”

[2] Harry Perros, “Computer Simulation Techniques: The definitive introduction

[3] COMP9334 Week7&Week8 problem solution