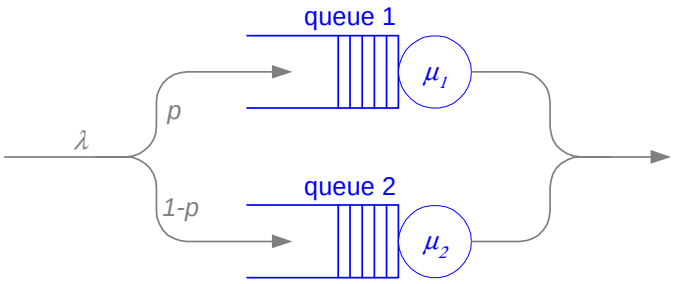
Problem 1

Assume that we have two M/M/1 systems in parallel, system one having a service rate of μ_1 *customers per second, system two serves* μ_2 *customers per second. The overall arrival rate is* λ*. Each arriving customer is subjected to an independent Bernoulli random experiment: with probability* $p \in (0,1)$ *the customer enters system one, with probability* $1-p$ *he enters system two.*

- *What is the stability condition for this system ?*
- *Find* p *such that the expected system response time of a customer is minimized.*

A graphical representation of the system described might look like this:



In general, for stability the mean arrival rate should be less than the mean service rate:

$$\lambda < m \cdot \mu$$

That equation ought to hold true for both queue 1 and queue 2. Since they are M/M/1 queues we obtain:

$$p \cdot \lambda < \mu_1$$
$$(1-p) \cdot \lambda < \mu_2$$

The mean response time of a single M/M/1 queue can be shown as (by Little's Law):

$$E[T] = \frac{1}{1-\rho} \cdot \frac{1}{\lambda}$$
$$= \frac{1}{\mu - \lambda}$$

So if we distinguish between these two queues:

$$E[T_1] = \frac{1}{\mu_1 - p \cdot \lambda}$$
$$E[T_2] = \frac{1}{\mu_2 - (1-p) \cdot \lambda}$$

Then, the expected mean response time can be described as:

$$E[T] = p \cdot E[T_1] + (1-p) \cdot E[T_2]$$
$$= \frac{p}{\mu_1 - p \cdot \lambda} + \frac{1-p}{\mu_2 - (1-p) \cdot \lambda}$$

Minimizing that equation leads to very complex expressions we were unable to handle without using software. So Maple's features helped us a lot – below are Maple's results:

```
> et:=p/(mu1-p*lambda)+(1-p)/(mu2-(1-p)*lambda);
```

$$et := \frac{p}{\mu1 - p\,\lambda} + \frac{1-p}{\mu2 - (1-p)\,\lambda}$$

```
> diffet:=diff(et,p);
```

$$diffet := \frac{1}{\mu1 - p\,\lambda} + \frac{p\,\lambda}{(\mu1 - p\,\lambda)^2} - \frac{1}{\mu2 - (1-p)\,\lambda} - \frac{(1-p)\,\lambda}{(\mu2 - (1-p)\,\lambda)^2}$$

```
> solve(diffet=0);
```

$$\{p = (2\,\mu1\,\lambda - 4\,\mu1\,\mu2 + 2\sqrt{-2\,\mu1^2\,\lambda\,\mu2 + 2\,\mu1^2\,\mu2^2 + \mu1^3\,\mu2 + \mu2\,\mu1\,\lambda^2 - 2\,\mu1\,\mu2^2\,\lambda + \mu1\,\mu2^3})/(2(\mu1 - \mu2)\,\lambda), \lambda = \lambda, \mu1 = \mu1, \mu2 = \mu2\}, \{p = (2\,\mu1\,\lambda - 4\,\mu1\,\mu2 - 2\sqrt{-2\,\mu1^2\,\lambda\,\mu2 + 2\,\mu1^2\,\mu2^2 + \mu1^3\,\mu2 + \mu2\,\mu1\,\lambda^2 - 2\,\mu1\,\mu2^2\,\lambda + \mu1\,\mu2^3})/(2(\mu1 - \mu2)\,\lambda), \lambda = \lambda, \mu1 = \mu1, \mu2 = \mu2\}$$

Maple found *two* solutions. Obviously, only *one* of them can be the actual minimum, i.e. the minimized system response time. We have to verify that by looking at:

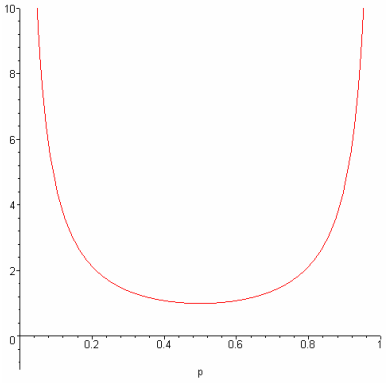
```
> diff(diffet,p);
```

$$\frac{2\,\lambda}{(\mu1 - p\,\lambda)^2} + \frac{2\,p\,\lambda^2}{(\mu1 - p\,\lambda)^3} + \frac{2\,\lambda}{(\mu2 - (1-p)\,\lambda)^2} + \frac{2\,(1-p)\,\lambda^2}{(\mu2 - (1-p)\,\lambda)^3}$$

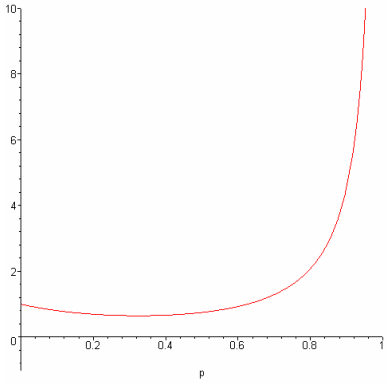
Additionally keep the stability conditions and the target interval [0..1] in mind. The formulas given above are only valid for a stable system!

A rule of thumb is that $p = 0.5$ for $\mu_1 = \mu_2$. Moreover, for $\mu_1 < \mu_2$ we get $p < 0.5$ and vice versa. Some screenshots of the system response times "prove" these statements:

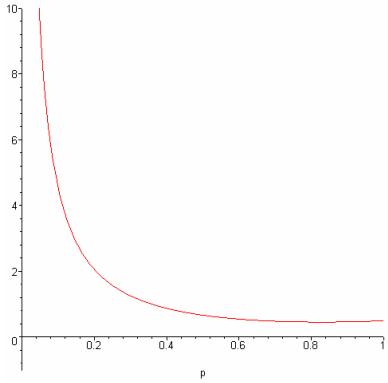
$\mu_1 = \mu_2 = 2 = \lambda = 2$



$\mu_1 = \lambda = 2,\ \mu_2 = 3$



$\mu_2 = \lambda = 2,\ \mu_1 = 4$



Problem 2

Consider the following system: there is a number N of queueing stations (QS), each having a single server and an infinite waiting room. The service times at QS i are independent and identically distributed with arbitrary service time distributions. However, the mean service time of QS i is known and given by $1/\mu_i$. Different QS might have different service time distributions. Customers arrive to the overall system according to a Poisson process with rate $\lambda > 0$. An arriving customer is subjected to one of the following policies:

***Random Selection (RS)**: The customer enters a randomly chosen QS, all QS are equiprobable.*

***Minimum Selection (MS)**: The customer enters the QS which at the time of his arrival has the fewest number of customers in the system (if multiple queues have the same minimal number of customers in the system, one of them is chosen randomly).*

***Load-Balancer Selection (LBS)**: The decision is made by a separate load-balancer entity, which works as follows:*

The load-balancer queries periodically with period $\Delta_t > 0$ the number of customers in all queues. Lets say the resulting values from sampling at time $n \cdot \Delta_t$ are $M_1(n),\ldots,M_N(n)$. For each QS i we compute the expected time to serve all the present customers, this time is given by:

$$S_i = \frac{M_i(n)}{\mu_i}$$

After sorting these times, we can determine indices $i_1,\ldots,i_N$ such that $S_{i_1} \le S_{i_2} \le \ldots \le S_{i_N}$. A customer which arrives before the next sampling instant $(n+1)\cdot\Delta_t$ is subjected to one of the following policies:

***LBS-I**: the customer is sent to QS i_1 (i.e. to the fastest one)*

***LBS-II**: assign QS i_1 the weight $w_{i_1} = 1$, and assign QS i_k $(k \ge 2)$ the weight:*

$$w_{i_k} = \begin{cases} \dfrac{S_{i_1}}{S_{i_k}} & : \quad S_{i_k} > 0,\ S_{i_1} > 0 \\ \dfrac{1}{1+S_{i_k}} & : \quad S_{i_k} > 0,\ S_{i_1} = 0 \\ 1 & : \quad S_{i_k} = 0 \end{cases}$$

An arriving customer is assigned to queue i_k with probability

$$p_{i_k} = \frac{w_{i_k}}{\sum_{j=1}^{N} w_{i_j}}$$

We want to compare the four policies RS, MS, LBS-I, and LBS-II with respect to the following steady-state performance measures:

- *System throughput*
- *Mean delay of a single customer*
- *Variance of a single customers delay*
- *Distribution of a single customers delay*

Develop a simulation model with OMNet++ which allows to vary the following parameters (e.g. by setting them in the omnetpp.ini file):

- *The number of queueing stations* N
- *For each station* i *the distribution of the service times (give the mean service time* $\frac{1}{\mu_i}$ *as a separate parameter, which must be consistent with the given distribution)*
- *The interval* Δ_t
- *The arrival rate* λ

Use the Replication/Deletion method (including proper transient removal !) to jointly estimate the steady-state system throughput and the mean delay of a single customer at a confidence level of $\alpha = 10\%$ *and relative error (confidence interval half-width) of at most 5%. Run the following experiments:*

- $\lambda = 100$ *customers per second,* $N = 5$*, all servers have exponential service times with a rate of 30 Customers per second,* $\Delta_t = 1s$
- $\lambda = 100$ *customers per second,* $N = 5$*, all servers have exponential service times with a rate of 30 Customers per second,* $\Delta_t = 0.01s$
- $\lambda = 100$ *customers per second,* $N = 5$*, all but one servers have exponential service times with a rate of 30 Customers per second, the remaining server has lognormal service times with mean 30 Customers per second and coefficient of variation of 10,* $\Delta_t = 0.01s$

Submit:

- *Your simulation program*
- *The .ini files for the different experiments*
- *For all simulation runs the following data both as tables and as figures, where appropriate: the length of the initial transient, the overall number of observations, the steady state throughput, the steady state mean system response time per customer, the variance of the steady state system response times per customer, minimum and maximum observed system response times.*
- *Your evaluation of the actually needed number of replications for each configuration and the achieved confidence interval width.*

Calculations

Our figures show the graphs used to determine the initial transient. Please note that we did not use the recommended moving average procedure with a constant window size but the cumulative average. This is similar to the moving average with an infinite window size. This approach was used due to speed increase during the evaluation.

The estimation for the minimum number of replications was calculated on the given sample mean and variance by testing different degrees of freedom for the student_t-distribution until the 5%-level was reached.

Summary

In our simulations the MinimumSelector is the optimal solution for the problem. Unfortunately, it cannot be created in real world with separated machines because it "looks" directly into the associated servers to determine the one with minimum queue length. Therefore the load balancers are used. With reasonable small update intervals the first variant approximates the MinimumSelector with much fewer communication overhead. The second variant is comparable with the RandomSelector.

Remarks

We computed the numbers using the German version of Microsoft Excel. That's why you will find a "," instead of "." as a decimal point. We also print just the "core" parts of the ini-files (the `[Parameters]` section) in order to save space. The electronically submitted solution contains the complete, working files.

The system response time is always stated in seconds. Additionally, figures show the SRT for the given number of jobs. We were unable to denote the axes due to problems with Microsoft Word.

Random Selection

Experiment I

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	10000	2000	100	0,10548945	0,01312668	4,15761E-06	1,03487073
2	10000	2000	100	0,1010675	0,01027379	1,26288E-06	0,705275728
3	10000	2000	100	0,10947101	0,01163283	4,15761E-06	0,712888969
4	10000	2000	100	0,09776952	0,00903154	1,26288E-06	0,674345401
5	10000	2000	100	0,10043146	0,0108966	1,26288E-06	0,749356118
6	10000	2000	100	0,11094009	0,01428327	2,08341E-06	0,810051238
7	10000	2000	100	0,10319781	0,01200754	1,26288E-06	0,803192443
8	10000	2000	100	0,10075805	0,00965704	2,08341E-06	0,608779467
9	10000	2000	100	0,10523674	0,01170525	1,26288E-06	0,940895578
10	10000	2000	100	0,09950537	0,01122316	1,26288E-06	0,700993973

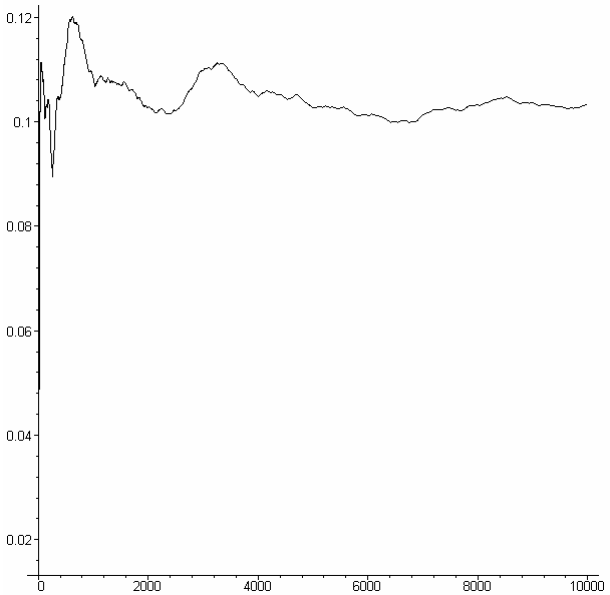


Figure 1: System Response Time (Random Selection, Experiment I)

Confidence interval: [0,10087035; 0,10590305]
Relative error: 2,43E-02
Estimated mean value: 0,1033867
Estimated minimum number of replications for 5% relative error: 5

random1.ini:

```
[Parameters]
servicenet.selector_type = "RandomSelector"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[*].service_time = exponential( 0.03333 )
servicenet.server[*].rate = 30
servicenet.sample_count = 10000
```

Experiment III

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	50000	20000	100	0,09688118	0,00925384	1,9485E-06	0,995231308
2	50000	20000	100	0,10038696	0,00990253	2,85097E-07	0,912039854
3	50000	20000	100	0,10006983	0,00952703	1,9485E-06	0,674696279
4	50000	20000	100	0,09994471	0,00997197	1,85537E-06	0,923038857
5	50000	20000	100	0,09863506	0,0096845	1,85537E-06	0,821231855
6	50000	20000	100	0,09619618	0,00869377	1,85537E-06	0,843944276
7	50000	20000	100	0,11823172	3,40886515	2,85097E-07	165,131508
8	50000	20000	100	0,11202878	2,14258485	1,9485E-06	170,638049
9	50000	20000	100	0,1001251	0,00984432	2,85097E-07	0,803207343
10	50000	20000	100	0,10013501	0,01053369	2,31948E-06	0,891649602

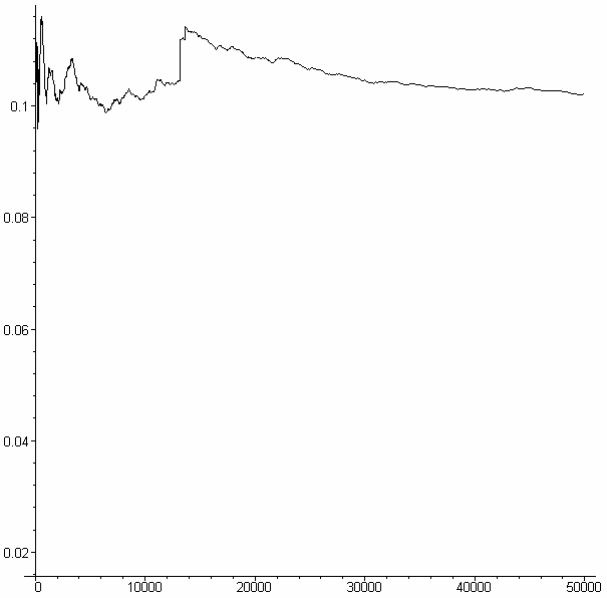


Figure 2: System Response Time (Random Selection, Experiment III)

Confidence interval: [0,098154911; 0,106372]
Relative error: 0,040176075
Estimated mean value: 0,102263455
Estimated minimum number of replications for 5% relative error: 8

random3.ini:

```
[Parameters]
servicenet.selector_type = "RandomSelector"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[0].service_time = exponential( 0.03333 )
servicenet.server[0].rate = 30
servicenet.server[1].service_time = exponential( 0.03333 )
servicenet.server[1].rate = 30
servicenet.server[2].service_time = exponential( 0.03333 )
servicenet.server[2].rate = 30
servicenet.server[3].service_time = exponential( 0.03333 )
```

```
servicenet.server[3].rate = 30
servicenet.server[4].service_time = lognormal( 0.03333, 10 )
servicenet.server[4].rate = 30
servicenet.sample_count = 50000
```

Minimum Selection

Experiment I

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	10000	4000	100	0,04939898	0,00241435	8,2526E-06	0,35778763
2	10000	4000	100	0,04820009	0,00285472	4,4635E-06	0,45987192
3	10000	4000	100	0,04974369	0,00271626	8,26912E-06	0,43030787
4	10000	4000	100	0,04557062	0,00222766	7,87038E-07	0,37267933
5	10000	4000	100	0,04689555	0,00238021	6,32901E-07	0,40230638
6	10000	4000	100	0,04614576	0,00213128	2,07832E-06	0,37565626
7	10000	4000	100	0,04703829	0,00237548	6,25882E-06	0,4300085
8	10000	4000	100	0,04330122	0,00186736	6,51164E-06	0,31675847
9	10000	4000	100	0,04939898	0,00241435	8,2526E-06	0,35778763
10	10000	4000	100	0,04820009	0,00285472	4,4635E-06	0,45987192

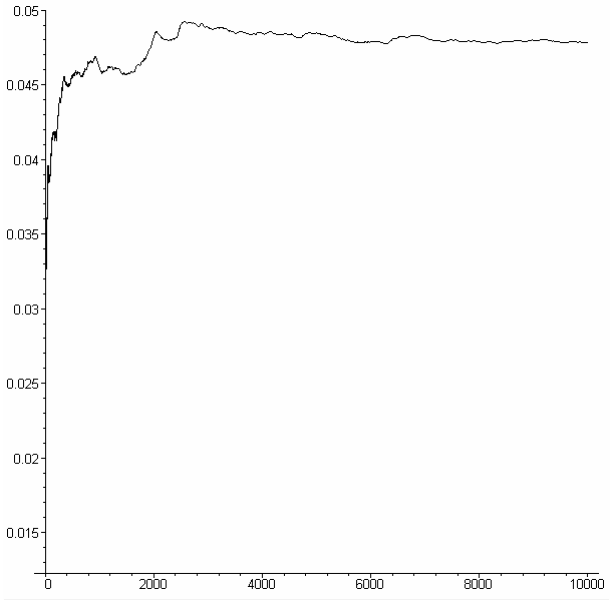


Figure 3: System Response Time (Minimum Selection, Experiment I)

Confidence interval: [0,046216057; 0,048562598]
Relative error: 0,024758122
Estimated mean value: 0,047389327
Estimated minimum number of replications for 5% relative error: 5

minimum1.ini:

```
[Parameters]
servicenet.selector_type = "MinimumSelector"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[*].service_time = exponential( 0.03333 )
servicenet.server[*].rate = 30
servicenet.sample_count = 10000
```

Experiment III

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	50000	10000	100	0,05729221	0,00351311	6,6925E-07	0,74766579
2	50000	10000	100	0,05720924	0,00346675	4,95198E-06	0,56054884
3	50000	10000	100	0,05790525	0,00383575	1,10067E-06	0,67967853
4	50000	10000	100	0,05717687	0,00365816	3,45426E-06	0,62340387
5	50000	10000	100	0,05636357	0,00346758	2,06853E-06	0,57525094
6	50000	10000	100	0,07119789	4,74058507	1,52101E-08	307,869147
7	50000	10000	100	0,05603428	0,00345572	1,90968E-06	0,66044544
8	50000	10000	100	0,05662469	0,00345123	1,0835E-06	0,57759992
9	50000	10000	100	0,05729221	0,00351311	6,6925E-07	0,74766579
10	50000	10000	100	0,05720924	0,00346675	4,95198E-06	0,56054884

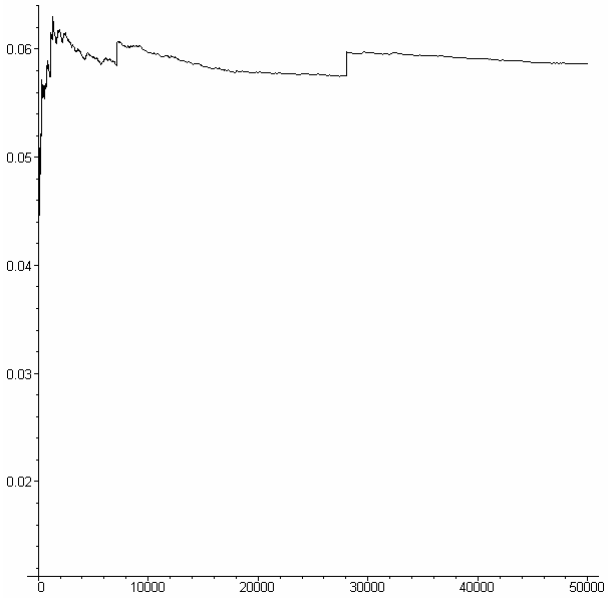


Figure 4: System Response Time (Minimum Selection, Experiment III)

Confidence interval: [0,055811601; 0,061049491]
Relative error: 0,044821504
Estimated mean value: 0,058430546
Estimated minimum number of replications for 5% relative error: 10

minimum3.ini:

```
[Parameters]
servicenet.selector_type = "MinimumSelector"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 0.1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[0].service_time = exponential( 0.03333 )
servicenet.server[0].rate = 30
servicenet.server[1].service_time = exponential( 0.03333 )
servicenet.server[1].rate = 30
servicenet.server[2].service_time = exponential( 0.03333 )
servicenet.server[2].rate = 30
servicenet.server[3].service_time = exponential( 0.03333 )
```

```
servicenet.server[3].rate = 30
servicenet.server[4].service_time = lognormal( 0.03333, 10 )
servicenet.server[4].rate = 30
servicenet.sample_count = 50000
```

Load Balancer I

Experiment I

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	200000	50000	100	3,535226171	8,688695471	5,21267E-06	18,1543452
2	200000	50000	100	3,754340833	11,21804347	1,06954E-05	23,0899427
3	200000	50000	100	3,416331071	8,799907439	2,23254E-05	18,772105
4	200000	50000	100	3,332601366	8,032249148	1,08979E-05	19,0908202
5	200000	50000	100	3,464411533	9,44853645	5,60472E-05	18,1533329
6	200000	50000	100	3,822943835	12,84355874	2,3917E-05	21,9594855
7	200000	50000	100	3,532716807	10,70177182	3,48337E-05	20,6859635
8	200000	50000	100	3,510522913	9,090038747	1,71571E-05	18,7214019
9	200000	50000	100	3,535226171	8,688695471	5,21267E-06	18,1543452
10	200000	50000	100	3,754340833	11,21804347	1,06954E-05	23,0899427

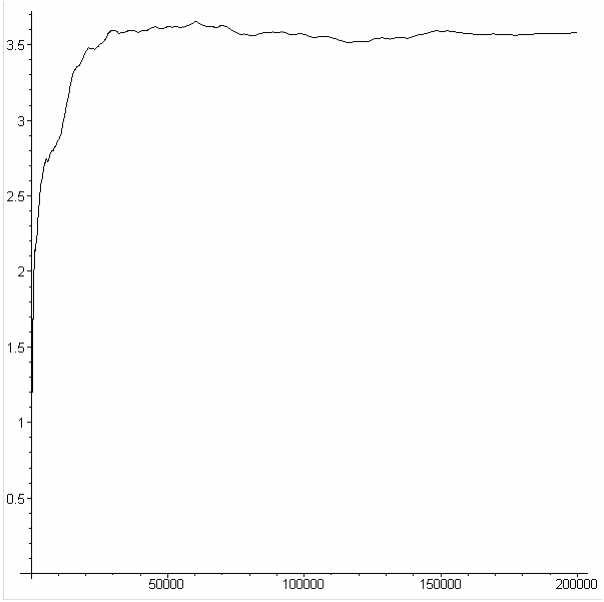


Figure 5: System Response Time (Load Balancer I, Experiment I)

Confidence interval: [3,47314938; 3,658582928]
Relative error: 0,026001193
Estimated mean value: 3,565866154
Estimated minimum number of replications for 5% relative error: 5

lbs_i1.ini:

```
[Parameters]
servicenet.selector_type = "LBS_I"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[*].service_time = exponential( 0.03333 )
servicenet.server[*].rate = 30
servicenet.sample_count = 100000
```

Experiment II

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	100000	20000	100	0,496218718	0,242280983	1,45411E-05	3,24727651
2	100000	20000	100	0,508372599	0,273728772	2,51303E-05	4,42676667
3	100000	20000	100	0,496483292	0,229323205	7,77346E-06	3,65645975
4	100000	20000	100	0,468661096	0,209201135	7,64377E-07	3,83100703
5	100000	20000	100	0,444883947	0,169609143	6,40615E-06	3,04770036
6	100000	20000	100	0,484517517	0,215415176	1,27651E-06	3,52656844
7	100000	20000	100	0,452004066	0,176683804	6,86363E-06	3,4471082
8	100000	20000	100	0,456215711	0,170302165	1,40616E-06	3,18833772
9	100000	20000	100	0,496218718	0,242280983	1,45411E-05	3,24727651
10	100000	20000	100	0,508372599	0,273728772	2,51303E-05	4,42676667

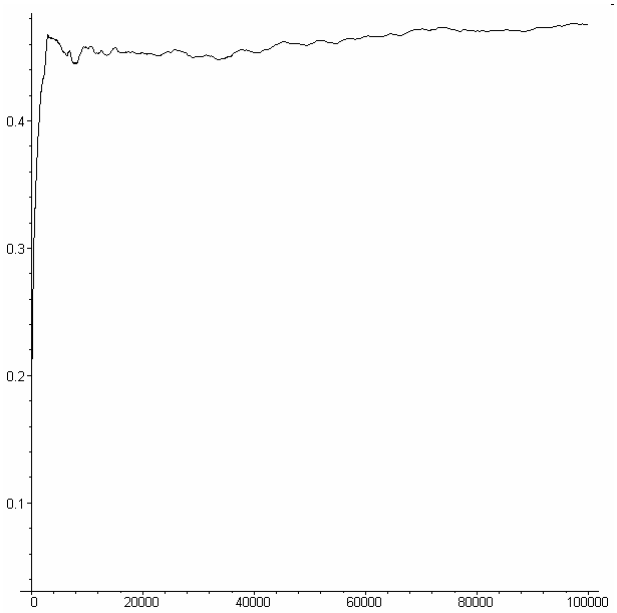


Figure 6: System Response Time (Load Balancer I, Experiment II)

Confidence interval: [0,46736173; 0,495027922]
Relative error: 0,028747391
Estimated mean value: 0,481194826
Estimated minimum number of replications for 5% relative error: 5

lbs_i2.ini:

```
[Parameters]
servicenet.selector_type = "LBS_I"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 0.1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[*].service_time = exponential( 0.03333 )
servicenet.server[*].rate = 30
servicenet.sample_count = 100000
```

Experiment III

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	50000	6000	100	0,511228865	5,667776936	5,12424E-05	223,243744
2	50000	6000	100	0,4499223	0,173065386	4,21877E-05	3,67346951
3	50000	6000	100	0,437713028	0,148413358	1,10983E-05	2,63473562
4	50000	6000	100	0,44723598	0,165664254	1,07026E-05	3,38701523
5	50000	6000	100	0,436723357	0,148323528	2,3245E-05	2,63931103
6	50000	6000	100	0,521010076	1,729264086	9,19786E-06	433,997589
7	50000	6000	100	0,435047999	0,166666497	7,44278E-07	3,67858474
8	50000	6000	100	0,450358767	0,179029355	1,8042E-05	3,12756051
9	50000	6000	100	0,511228865	5,667776936	5,12424E-05	223,243744
10	50000	6000	100	0,4499223	0,173065386	4,21877E-05	3,67346951

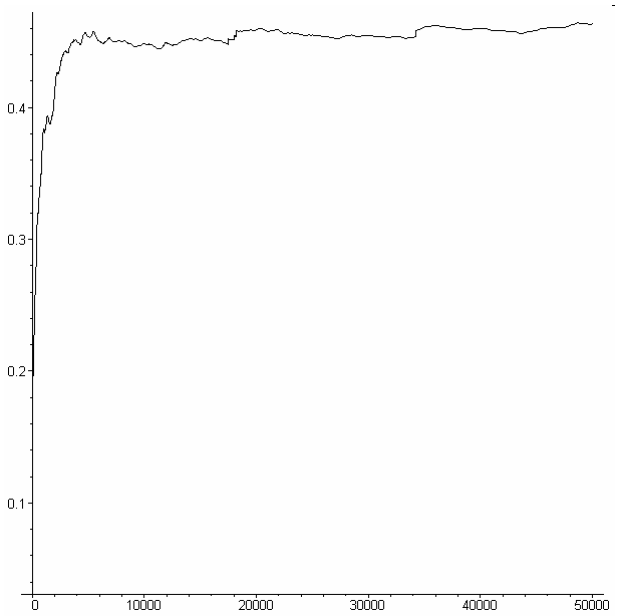


Figure 7: System Response Time (Load Balancer I, Experiment III)

Confidence interval: [0,444923855; 0,485154453]
Relative error: 0,043255066
Estimated mean value: 0,465039154
Estimated minimum number of replications for 5% relative error: 9

lbs_i3.ini:

```
[Parameters]
servicenet.selector_type = "LBS_I"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 0.1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[0].service_time = exponential( 0.03333 )
servicenet.server[0].rate = 30
servicenet.server[1].service_time = exponential( 0.03333 )
servicenet.server[1].rate = 30
servicenet.server[2].service_time = exponential( 0.03333 )
servicenet.server[2].rate = 30
```

```
servicenet.server[3].service_time = exponential( 0.03333 )
servicenet.server[3].rate = 30
servicenet.server[4].service_time = lognormal( 0.03333, 10 )
servicenet.server[4].rate = 30
servicenet.sample_count = 50000
```

Load Balancer II

Experiment I

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	100000	20000	100	0,101409755	0,010250111	6,6925E-07	0,922955257
2	100000	20000	100	0,099897329	0,009770517	2,64537E-06	0,854306807
3	100000	20000	100	0,101299294	0,010931962	1,13946E-06	1,20560426
4	100000	20000	100	0,100355843	0,010574762	4,91134E-07	1,17357419
5	100000	20000	100	0,098917519	0,009785474	2,23361E-06	0,916173814
6	100000	20000	100	0,099494525	0,009745105	2,05197E-07	0,912387498
7	100000	20000	100	0,09852845	0,009557463	9,87568E-07	0,849013604
8	100000	20000	100	0,101631845	0,010408413	1,06891E-06	1,04791611
9	100000	20000	100	0,101409755	0,010250111	6,6925E-07	0,922955257
10	100000	20000	100	0,099897329	0,009770517	2,64537E-06	0,854306807

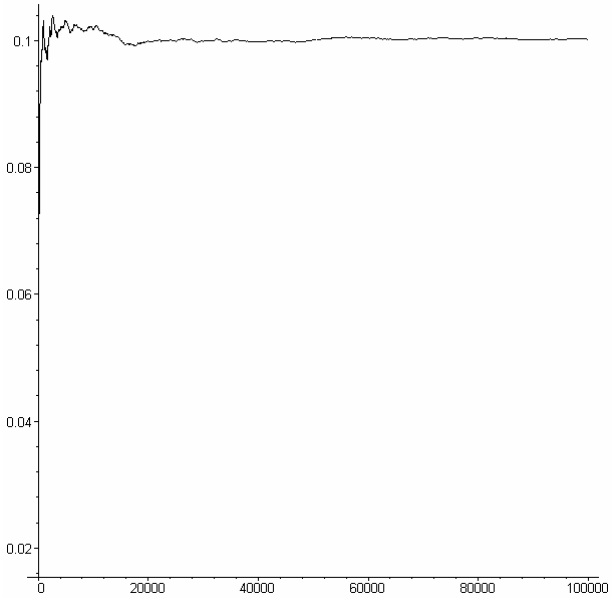


Figure 8: System Response Time (Load Balancer II, Experiment I)

Confidence interval: [0,099635806; 0,100932522]
Relative error: 0,006465208
Estimated mean value: 0,100284164
Estimated minimum number of replications for 5% relative error: 3

lbs_ii1.ini:

```
[Parameters]
servicenet.selector_type = "LBS_II"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[*].service_time = exponential( 0.03333 )
servicenet.server[*].rate = 30
servicenet.sample_count = 100000
```

Experiment II

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	100000	20000	100	0,101409755	0,010250111	6,6925E-07	0,922955257
2	100000	20000	100	0,099897329	0,009770517	2,64537E-06	0,854306807
3	100000	20000	100	0,101299294	0,010931962	1,13946E-06	1,20560426
4	100000	20000	100	0,100355843	0,010574762	4,91134E-07	1,17357419
5	100000	20000	100	0,098917519	0,009785474	2,23361E-06	0,916173814
6	100000	20000	100	0,099494525	0,009745105	2,05197E-07	0,912387498
7	100000	20000	100	0,09852845	0,009557463	9,87568E-07	0,849013604
8	100000	20000	100	0,101631845	0,010408413	1,06891E-06	1,04791611
9	100000	20000	100	0,101409755	0,010250111	6,6925E-07	0,922955257
10	100000	20000	100	0,099897329	0,009770517	2,64537E-06	0,854306807

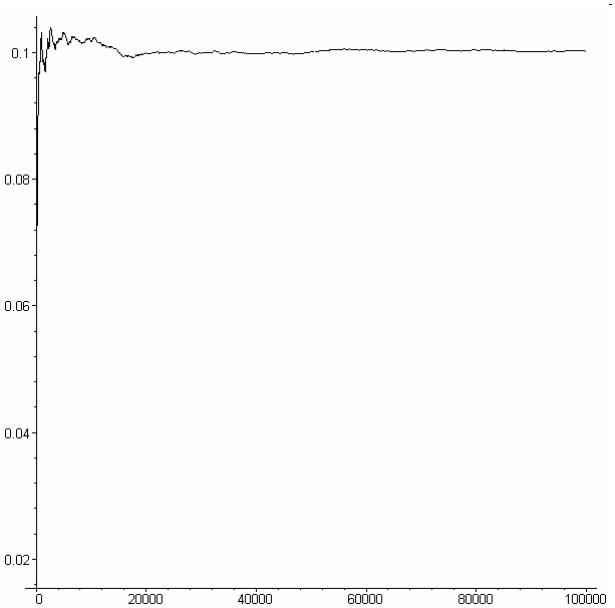


Figure 9: System Response Time (Load Balancer II, Experiment II)

Confidence interval: [0,099635806; 0,100932522]
Relative error: 0,006465208
Estimated mean value: 0,100284164
Estimated minimum number of replications for 5% relative error: 3

lbs_ii2.ini:

```
[Parameters]
servicenet.selector_type = "LBS_II"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 0.1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[*].service_time = exponential( 0.03333 )
servicenet.server[*].rate = 30
servicenet.sample_count = 100000
```

Experiment III

run	observations	initial transient	throughput	mean srt	var srt	min srt	max srt
1	50000	6000	100	0,099977335	0,009781141	6,6925E-07	0,849362174
2	50000	6000	100	0,101603144	0,010312708	2,29994E-06	0,864748994
3	50000	6000	100	0,103203034	0,011098821	1,10067E-06	1,01991757
4	50000	6000	100	0,097956872	0,009210619	7,95737E-06	0,779974076
5	50000	6000	100	0,096099666	0,009227438	2,08541E-06	0,764347643
6	50000	6000	100	0,11016598	4,313547894	1,52101E-08	307,869147
7	50000	6000	100	0,099925376	0,011099772	9,87568E-07	1,01492983
8	50000	6000	100	0,09792388	0,008784311	1,0835E-06	0,936253026
9	50000	6000	100	0,099977335	0,009781141	6,6925E-07	0,849362174
10	50000	6000	100	0,101603144	0,010312708	2,29994E-06	0,864748994

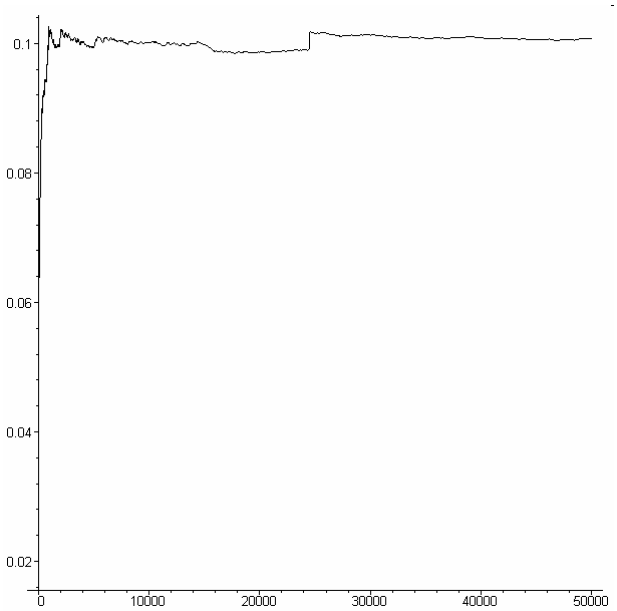


Figure 10: System Response Time (Load Balancer II, Experiment III)

Confidence interval: [0,098595703; 0,103091451]
Relative error: 0,022290701
Estimated mean value: 0,100843577
Estimated minimum number of replications for 5% relative error: 4

lbs_ii3.ini:

```
[Parameters]
servicenet.selector_type = "LBS_II"
servicenet.selector.random_generator = 2
servicenet.selector.delta_t = 0.1
servicenet.generator_rate = 100
servicenet.generator_rg = 1
servicenet.server_count = 5
servicenet.server[0].service_time = exponential( 0.03333 )
servicenet.server[0].rate = 30
servicenet.server[1].service_time = exponential( 0.03333 )
servicenet.server[1].rate = 30
servicenet.server[2].service_time = exponential( 0.03333 )
servicenet.server[2].rate = 30
```

```
servicenet.server[3].service_time = exponential( 0.03333 )
servicenet.server[3].rate = 30
servicenet.server[4].service_time = lognormal( 0.03333, 10 )
servicenet.server[4].rate = 30
servicenet.sample_count = 50000
```

As you can see from the numbers above, we ran each simulation ten times. That value has been chosen more or less randomly after observing that the relative error and the confidence intervals matched the requested boundaries. OMNet++ is quite fast on modern computers – the simulations took just a few seconds each. It may be necessary to further reduce the number of runs if we want to have some longer simulations. For the assignment, adapting the scripts would mean to spend more time writing .ini files than actually simulating.

The source code consists of many files. First we show the NED files (for each selection scheme plus the component setup) and afterwards present their corresponding .h and .cpp files.

randomselector.ned:

```
// RandomSelector -- a Selector selecting a random server
simple RandomSelector
    parameters:
        random_generator : const numeric,   // the random number generator to use
        server_count : const numeric;       // the server count
    gates:
        in: in;                         // source of requests
        out: out[];                     // one gate for each server
endsimple
```

minimumselector.ned:

```
// MinimumSelector -- a Selector selecting a random server
simple MinimumSelector
    parameters:
        random_generator : const numeric,   // the random number generator to use
        server_count : numeric;       // the server count
    gates:
        in: in;                         // source of requests
        out: out[];                     // one gate for each server
endsimple
```

lbs_i1.ned:

```
// LBS-I -- a Load balancer of scheme 1
simple LBS_I
    parameters:
        random_generator : const numeric,   // the random number generator to use
        server_count : numeric,       // the server count
        delta_t : numeric;            // time interval to update balancing data
    gates:
        in: in;                         // source of requests
        out: out[];                     // one gate for each server
endsimple
```

lbs_i2.ned:

```
// LBS-II -- a Load balancer of scheme 2
simple LBS_II
    parameters:
        random_generator : const numeric,   // the random number generator to use
        server_count : numeric,       // the server count
        delta_t : numeric;            // time interval to update balancing data
    gates:
        in: in;                       // source of requests
        out: out[];                   // one gate for each server
endsimple
```

server.ned:

```
// server -- the module serving requests
simple Server
    parameters:
        address : numeric,
        service_time : numeric,
        rate : numeric;
    gates:
        out: out;
        in: in;
endsimple
```

generator.ned:

```
// generator -- the module generating requests with exponential(rate) - distribution
simple Generator
    parameters:
       rate : const numeric,
        random_generator : const numeric;
    gates:
        out: out;
endsimple
```

sink.ned:

```
// sink-- the module collecting all packets and calculates statistics
simple Sink
    parameters:
        sample_count : const numeric;
    gates:
        in: in[];
endsimple
```

servicenet.ned:

```
// use the default sink which makes the statistical evaluation,
// the standard server, and the standard generator
import "sink", "server", "generator";

// Selector -- the interface of a selector implementation
simple Selector
    parameters:
        server_count : numeric;
    gates:
        out: out[];
        in: in;
endsimple


// ServiceNet -- the whole network consisting of Generator,
// Selector, Servers, and Sink
module ServiceNet
    parameters:
        server_count : numeric const,   // Anzahl Server im System
        selector_type : string,         // Typ des Selectors
        generator_rate : numeric const, // Erzeugungssrate
        generator_rg : numeric const,   // random number generator to use
        sample_count : numeric const;   // number of samples to record

    submodules:
        server:    Server[server_count];    // alle Server
                       parameters:
                           address = index; // server index

        selector:  selector_type like Selector; // der Selector
                       parameters:
                           server_count = server_count;
                       gatesizes:
                           out[server_count];

       sink:     Sink;                        // der Statistikgenerator
                       parameters:
                           sample_count = sample_count;
                       gatesizes:
                           in[server_count];

       generator: Generator;                    // der Auftragserzeuger
                       parameters:
                           rate = generator_rate,
                           random_generator = generator_rg;

    connections:
        for i=0..server_count-1 do
            selector.out[i] --> server[i].in,
            server[i].out --> sink.in[i];
        endfor;
        generator.out --> selector.in;
endmodule


network servicenet : ServiceNet
endnetwork
```

randomselector.h:

```
#ifndef RANDOMSELECTOR_H
#define RANDOMSELECTOR_H

#include <omnetpp.h>

class RandomSelector : public cSimpleModule {
public:
    Module_Class_Members(RandomSelector,cSimpleModule,16384)
    virtual void activity();
    virtual void finish();
};

#endif
```

randomselector.cpp:

```
#include "randomselector.h"

Define_Module( RandomSelector );

void RandomSelector::activity()
{
    // get parameters
    int rg = par("random_generator");
    int serverCount = par("server_count");

    for (;;) {
        // get message
        cMessage *msg = receive();

        // and send it to a randomly chosen server
        send(msg, "out", (int) uniform(0, serverCount, rg));
    }
}

void RandomSelector::finish()
{
}
```

minimumselector.h:

```
#ifndef MINIMUMSELECTOR_H
#define MINIMUMSELECTOR_H


#include <omnetpp.h>
#include "server.h"

class MinimumSelector : public cSimpleModule {
public:
    Module_Class_Members(MinimumSelector,cSimpleModule,16384)
    virtual void activity();
    virtual void finish();
    virtual void initialize();

private:
    Server **server;
};

#endif
```

minimumselector.cpp:

```
#include "minimumselector.h"

Define_Module( MinimumSelector );

void MinimumSelector::activity()
{
    // get parameters
    int rg = par("random_generator");
    int serverCount = par("server_count");

    // search servers and save reference to its workload parameter
    cModule *compound = parentModule();

    for (int i = 0; i<serverCount; i++) {
        cModule *svr = compound->submodule("server", i);
        if ((svr==0) || (strcmp(svr->className(), "Server")!=0)) {
            throw cException("server[] contains non-Server");
        }
        server[i] = (Server*) svr;
    }

    // message loop
    for (;;) {
        // get message
        cMessage *msg = receive();

        // search server with minimum workload (count the number of occurences)
        int i;
        int count = 0;
        long load = 100000000;
        for (i = 0; i<serverCount; i++) {
            long l = server[i]->getJobsInSystem();
            if (l<load) {
                load = l;
                count = 1;
            } else if (l==load) {
                count++;
            }
        }
        // choose one server with minimum workload randomly
        int serverNo = (int) uniform(0, count, rg);
        // find this server
        for (i = 0; i<serverCount; i++) {
            if (load==server[i]->getJobsInSystem()) {
                if (serverNo==0) break;
                serverNo--;
            }
        }

        if (serverNo!=0) throw cException("error while choosing server");

        // and send it to that server
        send(msg, "out", i);
    }

}

void MinimumSelector::initialize()
{
    server = new Server *[par("server_count")];
}

void MinimumSelector::finish()
{
    delete [] server;
}
```

lbs_i.h:

```
#ifndef LBS_I_H
#define LBS_I_H

#include <omnetpp.h>
#include "server.h"

class LBS_I : public cSimpleModule
{
    Module_Class_Members(LBS_I,cSimpleModule,8192)
    virtual void activity();
    virtual void finish();
    virtual void initialize();

private:
    Server **server;
    double *rate;
    double *balancing;
    int serverToUse;
    int serverCount;
    int rg;
    void update();
};


#endif
```

lbs_i.cpp:

```
#include "lbs_i.h"

Define_Module( LBS_I )

void LBS_I::activity()
{
    // get parameters
    simtime_t delta_t = par("delta_t");

    // search servers and save reference to its workload parameter
    // as well as their rate
    cModule *compound = parentModule();

    for (int i = 0; i<serverCount; i++) {
        cModule *svr = compound->submodule("server", i);
        if ((svr==0) || (strcmp(svr->className(), "Server")!=0)) {
            throw cException("server[] contains non-Server");
        }
        server[i] = (Server*) svr;
        rate[i] = svr->par("rate");
    }

    // create update-message and issue it immediately
    cMessage *updateMsg = new cMessage("update-balancer");
    scheduleAt(simTime() + delta_t, updateMsg);
    update();

    // message loop
    for (;;) {
        // get message
        cMessage *msg = receive();

        if (msg==updateMsg) {
            // update received, so do it and issue the message again delta_t secs later
            update();
            scheduleAt(simTime() + delta_t, updateMsg);
        } else {
            // send the message to the current server to use
            send(msg, "out", serverToUse);
        }
```

```
    }

}

void LBS_I::update()
{
    // search server with minimum workload (count the number of occurences)
    int i;
    int count = 0;
    double load = 100000000;
    for (i = 0; i<serverCount; i++) {
        // get queue length
        long l = server[i]->getJobsInSystem();
        if (l>0) --l;
        // calculate performance measure and save it
        double bl = (double) l/rate[i];
        balancing[i] = bl;
        if (bl<load) {
            load = bl;
            count = 1;
        } else if (bl==load) {
            count++;
        }
    }
    // choose one server with minimum workload randomly
    int serverNo = (int) uniform(0, count, rg);
    // find this server
    for (i = 0; i<serverCount; i++) {
        if (load==balancing[i]) {
            if (serverNo==0) break;
            serverNo--;
        }
    }
    // check on error
    if (serverNo!=0) throw cException("error while choosing server");

    // use the selected server until next update()
    serverToUse = i;
}

void LBS_I::initialize()
{
    // get parameters
    rg = par("random_generator");
    serverCount = par("server_count");
    // create arrays
    server = new Server *[serverCount];
    rate = new double[serverCount];
    balancing = new double[serverCount];
}

void LBS_I::finish()
{
    delete [] server;
    delete [] rate;
    delete [] balancing;
}
```

lbs_ii.h:

```
#ifndef LBS_II_H
#define LBS_II_H

#include <omnetpp.h>
#include "server.h"

class LBS_II : public cSimpleModule
{
    Module_Class_Members(LBS_II,cSimpleModule,8192)
    virtual void activity();
    virtual void finish();
    virtual void initialize();

private:
    Server **server;
    double *rate;
    double *balancing;
    int serverCount;
    int rg;

    void update();
    int chooseServer();
};


#endif
```

lbs_ii.cpp:

```
#include "lbs_ii.h"

Define_Module( LBS_II )

void LBS_II::activity()
{
    // get parameters
    simtime_t delta_t = par("delta_t");

    // search servers and save reference to its workload parameter
    // as well as their rate
    cModule *compound = parentModule();

    for (int i = 0; i<serverCount; i++) {
        cModule *svr = compound->submodule("server", i);
        if ((svr==0) || (strcmp(svr->className(), "Server")!=0)) {
            throw cException("server[] contains non-Server");
        }
        server[i] = (Server*) svr;
        rate[i] = svr->par("rate");
    }

    // create update-message and issue it immediately
    cMessage *updateMsg = new cMessage("update-balancer");
    scheduleAt(simTime() + delta_t, updateMsg);
    update();

    // message loop
    for (;;) {
        // get message
        cMessage *msg = receive();

        if (msg==updateMsg) {
            // update received, so do it and issue the message again delta_t secs later
            update();
            scheduleAt(simTime() + delta_t, updateMsg);
        } else {
            // send the message to a server
            send(msg, "out", chooseServer());
```

```
        }
    }

}

void LBS_II::update()
{
    // search minimum workload
    int i;
    double load = 100000000;
    for (i = 0; i<serverCount; i++) {
        // get queue length
        long l = server[i]->getJobsInSystem();
        if (l>0) --l;
        // calculate performance measure and save it
        double bl = (double) l/rate[i];
        balancing[i] = bl;
        // record smallest workload
        if (bl<load) {
            load = bl;
        }
    }

    // calculate weights
    if (load==0.0) {
        // minimum workload is zero => use different set of equations
        for (i = 0; i<serverCount; i++) {
            if (balancing[i]!=0.0) balancing[i] = 1/(1+balancing[i]);
            else balancing[i] = 1.0;
        }
    } else {
        // standard weight calculation
        for (i = 0; i<serverCount; i++) {
            if (balancing[i]!=0.0) balancing[i] = load/balancing[i];
            else balancing[i] = 1.0;
        }
    }

    // total sum of weights
    double wSum = 0.0;
    for (i = 0; i<serverCount; i++) wSum += balancing[i];

    // normalize weights
    for (i = 0; i<serverCount; i++) {
        balancing[i] /= wSum;
        ev << "balancing[" << i << "] = " << balancing[i] << endl;
    }

    // change balancing into distribution
    for (i = 1; i<serverCount; i++) balancing[i] += balancing[i-1];
}

int LBS_II::chooseServer()
{
    // get random value
    double rVal = uniform(0, 1, rg);

    // get server no via inverse transformation method
    int i = 0;
    while ((i<serverCount) && (balancing[i]<rVal)) ++i;
    if (i==serverCount) throw cException("error while choosing server!");

    ev << "chose server " << i << endl;

    return i;
}

void LBS_II::initialize()
{
    // get parameters
    rg = par("random_generator");
    serverCount = par("server_count");
    // create arrays
    server = new Server *[serverCount];
    rate = new double[serverCount];
    balancing = new double[serverCount];
}
```

```
void LBS_II::finish()
{
    delete [] server;
    delete [] rate;
    delete [] balancing;
}
```

server.h:

```
#ifndef SERVER_H
#define SERVER_H

#include <omnetpp.h>

class Server : public cSimpleModule {
public:
    Module_Class_Members(Server,cSimpleModule,16384)
    virtual void activity();
    virtual void finish();
    virtual void initialize();

    long getJobsInSystem() {return jobsInSystem;}

private:
    long jobsInSystem;   // parameter holding current number of jobs in the server
    long messageCount;   // number of jobs served
    cQueue queue;        // waiting line
};

#endif
```

server.cpp:

```
#include "server.h"

Define_Module( Server );

void Server::activity()
{
    // init counting
    messageCount = 0;
    jobsInSystem = 0;

    // get parameters
    cPar &service_time = par("service_time");

    // setup finish-message
    cMessage finishMessage("finish");

    // the currently processed message
    cMessage *currentMessage = 0;

    // process messages
    for (;;) {
        // fetch next message
        cMessage *msg = receive();
        if (msg==&finishMessage) {
            // finished processing, so send away current message
            messageCount++;
            send(currentMessage, "out");
            // handle next message if available
            if (!queue.empty()) {
                currentMessage = (cMessage*) queue.pop();
                scheduleAt(simTime() + (simtime_t) service_time, &finishMessage);
            } else currentMessage = 0;
        } else {
            // new message arrival
            if (currentMessage==0) {
                // server is idle => process message immediately
                currentMessage = msg;
                scheduleAt(simTime() + (simtime_t) service_time, &finishMessage);
            } else {
                // server is busy => enqueue message
                queue.insert(msg);
            }
        }
        // update job count
```

```
        jobsInSystem = queue.length() + (currentMessage!=0) ? 1 : 0;
    }
}

void Server::initialize()
{

}

void Server::finish()
{
    int index = par("address");
    ev << "Server " << index << ": Total messages processed: " << messageCount << endl;
}
```

generator.h:

```
#ifndef GENERATOR_H
#define GENERATOR_H

#include <omnetpp.h>

// a class sending messages through its out-gate
// in a poisson process, i.e, with exponentially
// distributed interarrival times
class Generator : public cSimpleModule {
public:
    Module_Class_Members(Generator,cSimpleModule,16384)
    virtual void activity();
    virtual void finish();

private:
    long messageCount;
};

#endif
```

generator.cpp:

```
#include "generator.h"

Define_Module( Generator );

void Generator::activity()
{
    // init counting
    messageCount = 0;

    // get parameters
    double rate = par("rate");
    double lambda = 1/rate;
    int rg = par("random_generator");

    // create messages
    for (;;) {
        // create message and send it
        cMessage *msg = new cMessage("job");
        msg->setTimestamp();
        send(msg, "out");
        messageCount++;

        // wait for exponential time (poisson process!)
        double waitTime = exponential(lambda, rg);
        wait(waitTime);
    }
}

void Generator::finish()
```

```
{
    ev << "Total messages sent: " << messageCount << endl;
}
```

sink.h:

```
#ifndef SINK_H
#define SINK_H

#include <omnetpp.h>

class Sink : public cSimpleModule {
public:
    Module_Class_Members(Sink,cSimpleModule,16384)
    virtual void activity();
    virtual void finish();

    // discard data recorded so far
    // call after initial transient to get steady state values
    void reset();

private:
    cStdDev sInfo;      // statistics since last reset()
    long totalCount;    // total count of samples since start of activity()
};

#endif
```

sink.cpp:

```
#include "sink.h"

// Module registration:
Define_Module( Sink );

void Sink::activity()
{
    // reset statistics
    sInfo.clearResult();
    totalCount = 0;

    // get number of samples to record
    int sampleCount = par("sample_count");

    // file to write history to
    cOutVector srt("system response time", 1);

    // message loop
    for(int i = 0; i<sampleCount; i++)
    {
        // receive a message and record time statistics
        cMessage *msg = receive();
        // create statistics
        simtime_t msrt = simTime() - msg->timestamp();
        sInfo.collect(msrt);
        srt.record(msrt);
        // delete msg
        delete msg;
    }
    endSimulation();
}

void Sink::finish()
{
    ev << "Total jobs processed: " << totalCount + sInfo.samples() << endl;
    ev << "Number of jobs used for statistics: " << sInfo.samples() << endl;
    ev << "Avg system response time:    " << sInfo.mean() << endl;
    ev << "Standard deviation:   " << sInfo.stddev() << endl;
```

```
    ev << "Max system response time:    " << sInfo.max() << endl;
    ev << "Min system response time:    " << sInfo.min() << endl;

    ev << endl;
 }

void Sink::reset()
{
    // update totalCount
    totalCount += sInfo.samples();
    sInfo.clearResult();
}
```