UNIVERSITY OF DEBRECEN

PERFORMANCE ANALYSIS OF FINITE-SOURCE RETRIAL QUEUEING SYSTEMS WITH HETEROGENEOUS NON-RELIABLE SERVERS AND DIFFERENT SERVICE POLICIES

JÁNOS SZTRIK, JÁNOS ROSZIK

PREPRINTS NO. 316
(TECHNICAL REPORT NO. 2004/6)

INSTITUTE OF INFORMATICS

2004

Performance Analysis of Finite-Source Retrial Queueing Systems with Heterogeneous Non-Reliable Servers and Different Service Policies

J. Sztrik, J. Roszik
Department of Informatics Systems and Networks, University of Debrecen
P.O. Box 12, H-4010, Debrecen, Hungary
{jsztrik,jroszik}@inf.unideb.hu

Abstract

This paper deals with the performance analysis of multiple server retrial queueing systems with a finite number of homogeneous sources of calls, where the heterogeneous servers are subject to random breakdowns and repairs. The requests are serviced according to random and fastest free server disciplines.

The novelty of this investigation is the different service rates and different service policies with the non-reliability of the servers, which has essential influence on the performance of the system, thus it plays an important role in practical modeling of computer and communication systems. All random variables involved in the model construction are assumed to be exponentially distributed and independent of each other.

The main steady-state performability measures are derived, and several numerical calculations are carried out by the help of the MOSEL tool under different service disciplines. The numerical results are graphically displayed to illustrate the effect of arrival, retrial, and failure rates on the mean response time and on the overall system's utilization.

Keywords: retrial queueing systems, finite number of sources, multiple server queues, heterogeneous servers, ordered service, non-reliable server, performance tool, performance and reliability measures

1 Introduction

Retrial queueing systems (or queueing systems with repeated attempts or returning customers) are characterized by the following feature: a primary request finding all servers busy on arrival does not wait in a queue, but leaves the service area and after some random time repeats his demand. For the most important results on this type of queues see, for example [1], [2], [3], [4]. This feature plays a special role in many computer and communication systems having a significant negative impact on the performance characteristics of the system. For some examples in the field of computer systems and communication networks see: [5], [6], [7] and [8].

[1]Research is partially supported by Hungarian Scientific Research Fund OTKA under T0-34280/2000 and FKFP grant under 0191/2001.

In general, the components of the practical computer systems are subject to random breakdowns, what has a heavy influence on the performance measures just as the retrial phenomenon, so if we model computer systems containing non-reliable components it is important to take it into account in the model construction. Of course, the breakdown of the servers has the most significant negative impact on the performance of the most frequently used client-server architecture. For modeling of these type of systems, both infinite- and finite-source retrial queues with server breakdowns were applied , see for example [9], [10], [11] and [12]. Queueing systems with heterogeneous servers are still an interesting topic, for recent results, confer for example [13], [14]. However, for retrial queueing systems with heterogeneous servers we have found only [15]. To the best knowledge of the authors there is no paper on finite-source retrial queues with heterogeneous servers even in reliable case.

In this paper, we analyze the finite-source retrial queue with non-reliable heterogeneous (asymmetric) servers, that is the servers has different parameters in service, failure and repair rates. In the present study, the most important heterogeneous characteristic is the service rate, since we compare two service policies, namely Random and Fastest Free Server (FFS). In the case of Random service discipline, the requests are assigned to the idle servers randomly, and in the other case, the requests are assigned to the fastest available free server. The purpose of this paper is to generalize the model of [4], [12]. The novelty of this investigation is the different service rates and different service policies with the non-reliability of the servers.

The organization of the paper is as follows. In the next section we give the mathematical model description and derive the performance measures. Then we use an efficient software tool MOSEL (Modeling, Specification and Evaluation Language), see [16], to formulate the model, and to obtain the performance measures. In Section 3, we present some numerical examples for the models under different service disciplines. The results are graphically displayed using IGL (Intermediate Graphical Language) which belongs to MOSEL. By the help of these figures we illustrate the effect of arrival, retrial and failure rates on the mean response time and on the overall system's utilization. Section 4 is devoted to some conclusions, and in the Appendices, we give the MOSEL source codes for the two service policies. These MOSEL programs were generated with our shell scripts for the 4 servers and 20 sources case.

2 The $M/\vec{M}/c//K$ retrial queueing model with non-reliable servers and different service policies

Consider a finite-source retrial queueing system with c servers, where the primary calls are generated by K, $c < K < \infty$ sources. Each server can be in operational (up) or non-operational (down) states, and it can be idle or busy. Each source can be in three states: generating a primary call (free), sending repeated calls and under service by one of the servers. If a source is free at time t, it can generate a primary call during the interval $(t, t + dt)$ with probability $\lambda dt + o(dt)$. If one of the servers is up and idle at the moment of the arrival of the call then the service of the call starts. In the case of Random Service discipline, the requests are assigned to the available free servers randomly, but in the Fastest Free Server case the availability and idleness of the servers are always examined according to the increasing order of the servers indices. The service is finished during the interval $(t, t + dt)$ with probability $\mu_i dt + o(dt)$ if the ith server is available.

Server i can fail during the interval $(t, t + dt)$ with probability $\delta_i dt + o(dt)$ if it is idle, and with probability $\gamma_i dt + o(dt)$ if it is busy. If the server fails in busy state, the interrupted request returns to the orbit, i.e. the source moves into the sending repeated calls state. If $\delta_i = 0, \gamma_i > 0$ or $\delta_i = \gamma_i > 0$ *active or independent breakdowns* can be discussed, respectively. The repairman follows FIFO discipline to fix up the servers' breakdowns, the repair time of the ith server is exponentially distributed with a finite mean $1/\tau_i$. If all the servers are failed two different cases can be treated. Namely, *blocked sources* case when all the operations are stopped expect from the repair of the servers. In the *unblocked (intelligent) sources* case only service is interrupted but all the other operations are continued. In this paper, we compare the service disciplines in the unblocked sources case with independent server's breakdowns.

If all the servers are busy or failed at the moment of the arrival of a call the source starts generation of a Poisson flow of repeated calls with rate ν until it finds an available free server. After service the source becomes free, and it can generate a new primary call, and the server becomes idle so it can serve a new call. All the times involved in the model are assumed to be mutually independent of each other.

The state of the system at time t can be described by the process $X(t) = (\alpha_1(t), ..., \alpha_c(t); N(t))$, where $N(t)$ is the number of sources of repeated calls, $\alpha_i(t)$, i=1, ..., c, denotes the state of the ith server at time t. If there is a customer under service at the ith server, $\alpha_i(t) = 1$, if it is operational and idle then $\alpha_i(t) = 0$, otherwise the server is failed and $\alpha_i(t) = -1$.

Because of the exponentiality of the involved random variables this process is a Markov chain with a finite state space. Since the state space of the process $(X(t), t \geq 0)$ is finite, the process is ergodic for all reasonable values of the rates involved in the model construction. From now on we assume that the system is in the steady-state. Because of the fact that the state space of the describing Markov chain is very large, it is difficult to calculate the system measures in the traditional way of solving the system of steady-state equations. To simplify this procedure we used the software tool MOSEL.
Let us define the stationary probabilities by:

$$P(i_1, ..., i_c, j) = \lim_{t\to\infty} P\{\alpha_1(t) = i_1, ..., \alpha_c(t) = i_c, N(t) = j\}, \;\; i_1, ...i_c = -1, 0, 1, \;\; j = 0, ..., K^*,$$

$$\text{where} \quad K^* = K - \sum_{i_k, i_k=1} i_k.$$

Furthermore, let us denote by $C(t)$ the number of busy servers, by $A(t)$ the number of available servers at time t, and denote by $p_{kj} = \lim_{t\to\infty} P\{C(t) = k, N(t) = j\}$ the joint distribution of the number of busy servers and the number of repeated calls.

Once we have obtained the above defined probabilities the main steady state system performance measures can be derived as follows:

- *Mean number of sources of repeated calls*

$$N = E[N(t)] = \sum_{k=0}^{c}\sum_{j=1}^{K} j p_{kj} = \sum_{i_1,...,i_c}\sum_{j=1}^{K^*} jP(i_1, ..., i_c, j).$$

- *Utilization of the k-th server*

$$U_k = \sum_{i_1,...,i_c, i_k=1}\sum_{j=0}^{K^*} P(i_1, ..., i_c, j), \quad k = 1, ..., c.$$

- *Mean number of busy servers*

$$C = E[C(t)] = \sum_{\substack{i_1,\dots,i_c \\ K^*>0}} \sum_{j=0}^{K^*} K^* P(i_1, ..., i_c, j) = \sum_{k=1}^{c} U_k.$$

- *Mean number of calls staying in the orbit or in service*

$$M = E[N(t) + C(t)] = N + C.$$

- *Utilization of the repairman*

$$U_R = \sum_{\substack{i_1,\dots,i_c \\ -1\in\{i_1,\dots,i_c\}}} \sum_{j=0}^{K^*} P(i_1, ..., i_c, j).$$

- *Utilization of the sources*

$$U_{SO} = \begin{cases} \frac{E[K-C(t)-N(t);A(t)>0]}{K} & \text{for blocked case,} \\ \frac{E[K-C(t)-N(t)]}{K} & \text{for unblocked case.} \end{cases}$$

- *Overall utilization of the system*

$$U_O = C + KU_{SO} + U_R.$$

- *Mean rate of generation of primary calls*

$$\overline{\lambda} = \begin{cases} \lambda E[K - C(t) - N(t); A(t) > 0] & \text{for blocked case,} \\ \lambda E[K - C(t) - N(t)] & \text{for unblocked case.} \end{cases}$$

- *Mean waiting time*

$$E[W] = N/\overline{\lambda}.$$

- *Mean response time*

$$E[T] = M/\overline{\lambda}.$$

2.1 Validation of results

The results of the tool in the reliable case were validated by the Pascal program given in the book of Falin and Templeton [1]. The service rates are the same for all servers in each cases. In Table 1 we can see that the corresponding performance measures are very close to the reliable case and to each other with Random and Fastest Free Server (FFS) disciplines with very low failure and very high repair rates. The results are the same up to the 6th decimal digit.

In the non-reliable single server case, the results were tested by the $M/M/1//K$ retrial model with server's breakdowns which was studied in [12].

	Pascal [1]	random	FFS
Number of servers:	4	4	4
Number of sources:	20	20	20
Request's generation rate:	0.1	0.1	0.1
Service rate:	1	1	1
Retrial rate:	1.2	1.2	1.2
Server's failure rate:	–	1e-25	1e-25
Server's repair rate:	–	1e+25	1e+25
Mean waiting time:	0.1064954794	0.1064959317	0.1064959929
Mean number of busy servers:	1.8007480431	1.8007485102	1.8007485548
Mean number of sources of repeated calls:	0.1917715262	0.1917717923	0.1917718470

Table 1: Validations in the reliable case

3 Numerical examples

In this section we present some numerical results to illustrate graphically the differences between the service disciplines in the mean response time, in the utilizations of the servers and in the overall system's utilization. In the legends of the figures, the Fastest Free Server policy is referenced as *ordered*, and the random case where the service rate of the servers is the average of the rates of the heterogeneous cases is referred to as *averaged random*.

The input system parameters of the figures are collected in Table 2.

	c	K	λ	$\mu_1, ..., \mu_c - \mu_{avg}$	ν	δ,γ	τ
Figure 1,5	4	20	x axis	8,5,4,1 – 4.5	4	0.01	0.2
Figure 2,6	4	20	4	8,5,4,1 – 4.5	x axis	0.01	0.2
Figure 3,4,7	4	20	1	8,5,4,1 – 4.5	4	x axis	0.2

Table 2: Input system parameters

3.1 Comments

- In Figure 1 we can see the difference between the three cases in the mean response time depending on the primary request generation rate. The difference between the two random cases are not too significant, but the Fastest Free Server (ordered) case always has better response time, especially when more and more requests arrive.

- In Figure 2 it is demonstrated how long the retrial rate has a significant influence on the mean response time, after that the decrease is not considerable.

- In Figure 3 it is shown how the increase of the server's failure rate affects the mean response time. The averaged random case has a little better response time than the not averaged random case like in the former figures. The surprising decrease in the mean response time of the Fastest Free Server case can be explained by the help of Figure 4.

- In Figure 4 we can see the server utilizations versus the servers' failure rate with the same parameter setup as in Figure 3. In the random case, the slowest server has the

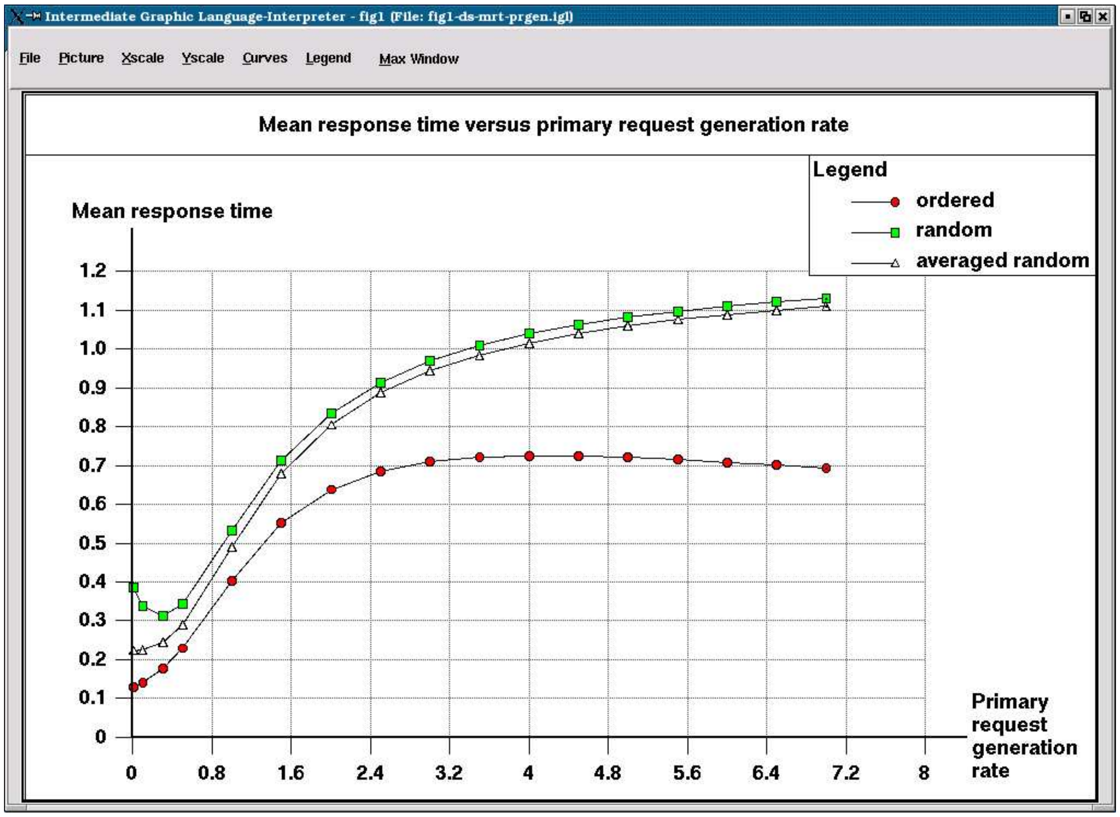

Figure 1: Mean response time versus primary request generation rate

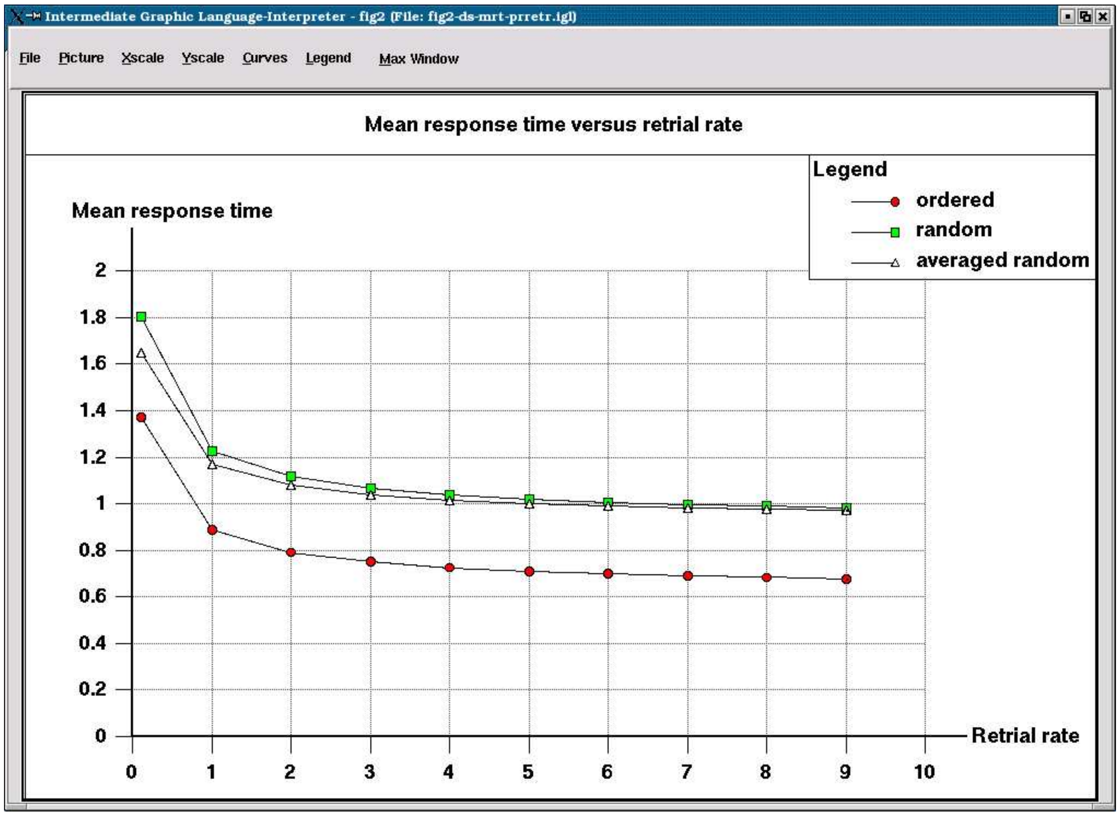

Figure 2: Mean response time versus retrial rate

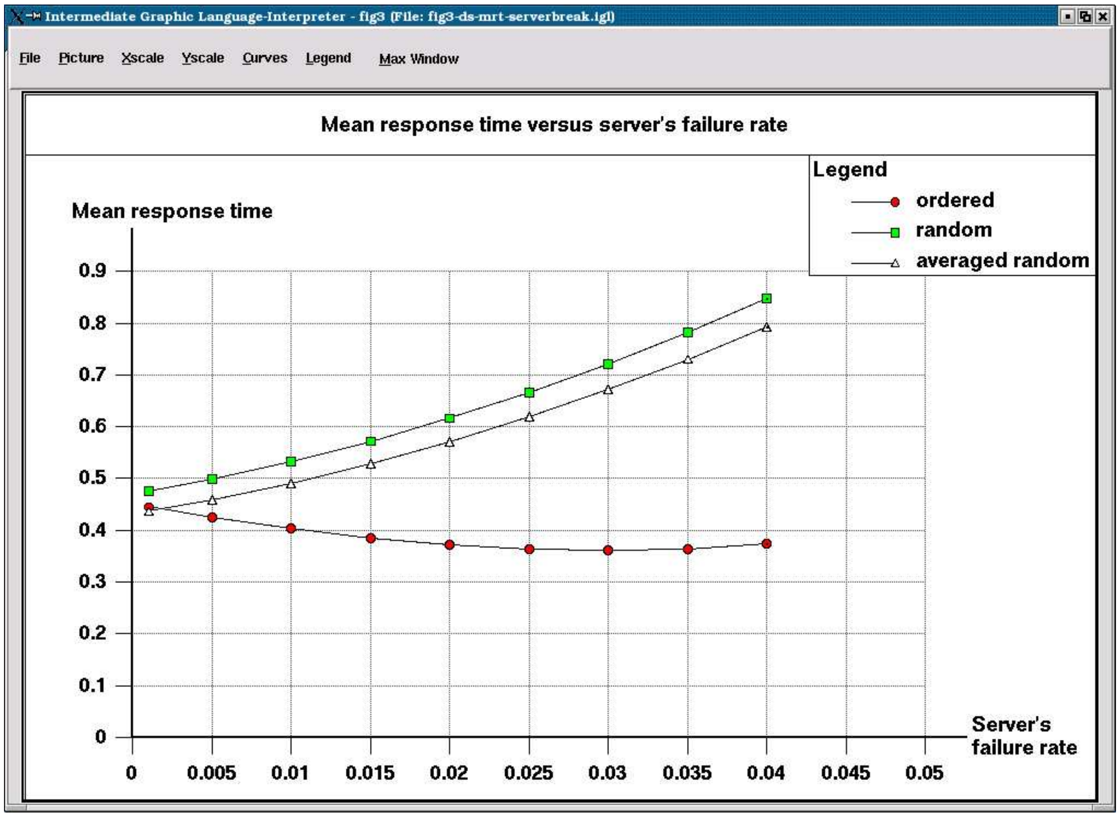

Figure 3: Mean response time versus server's failure rate

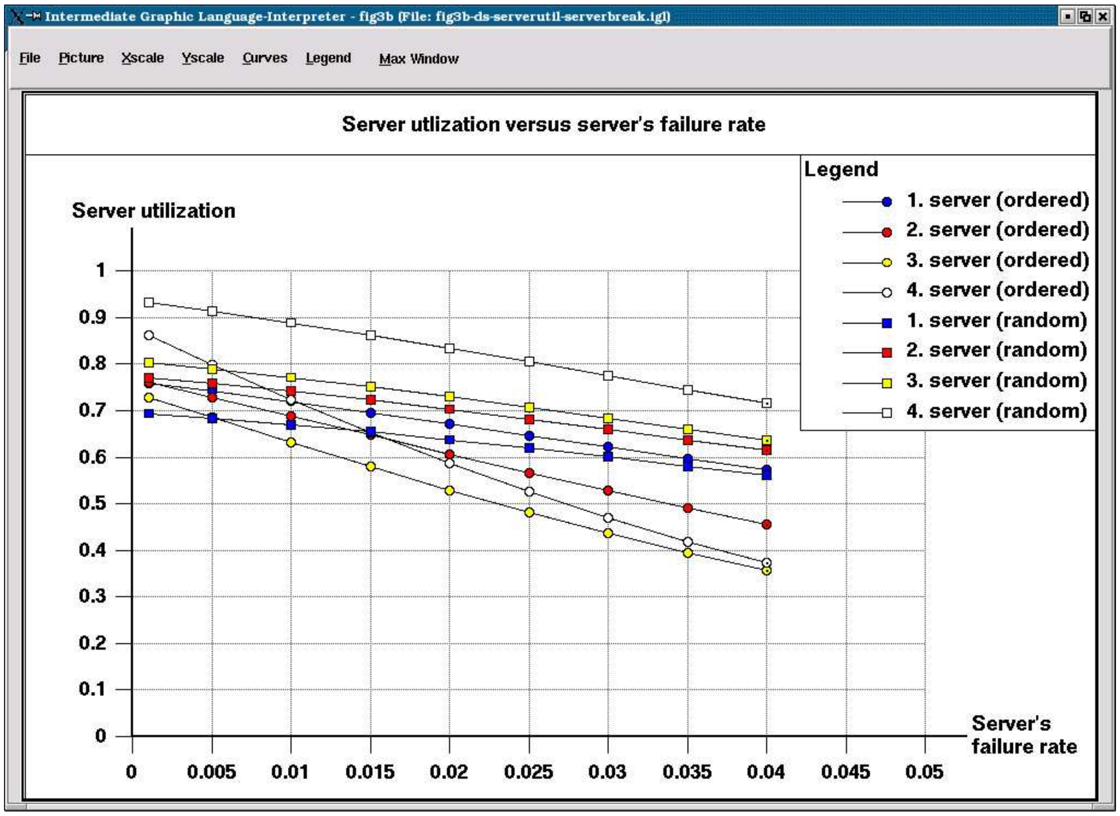

Figure 4: Server utilization versus server's failure rate

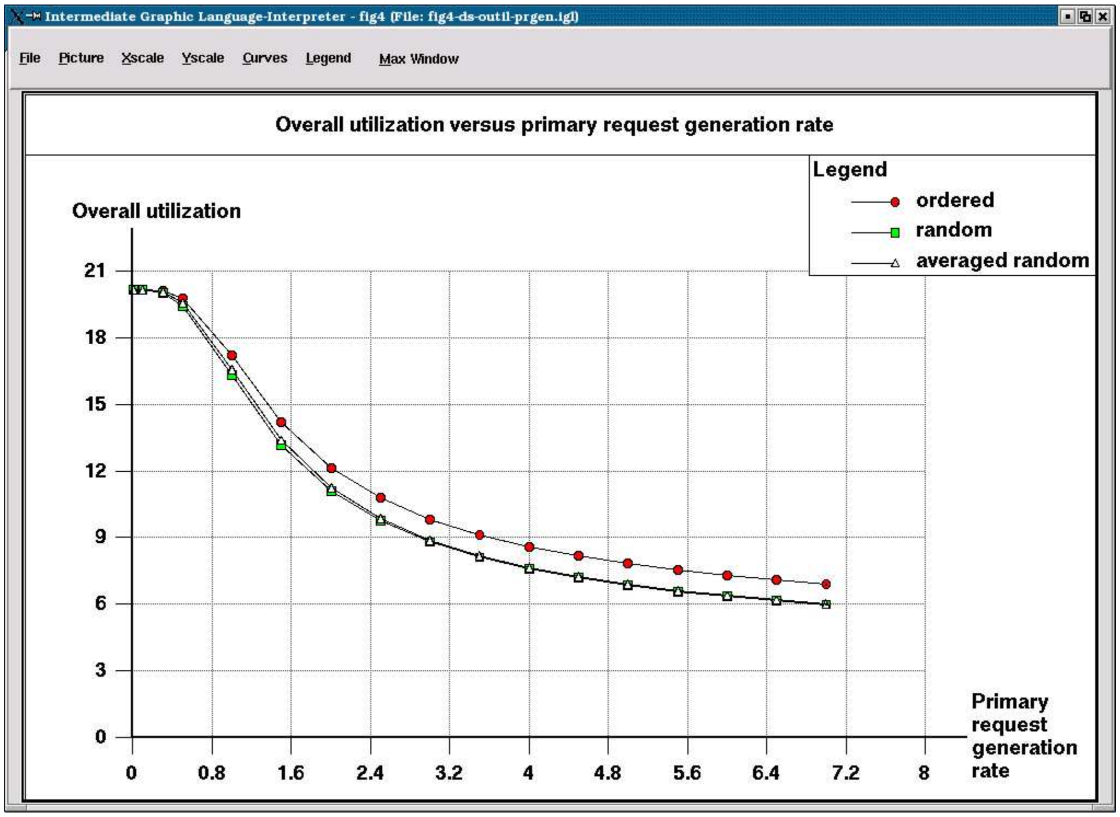

Figure 5: Overall utilization versus primary request generation rate

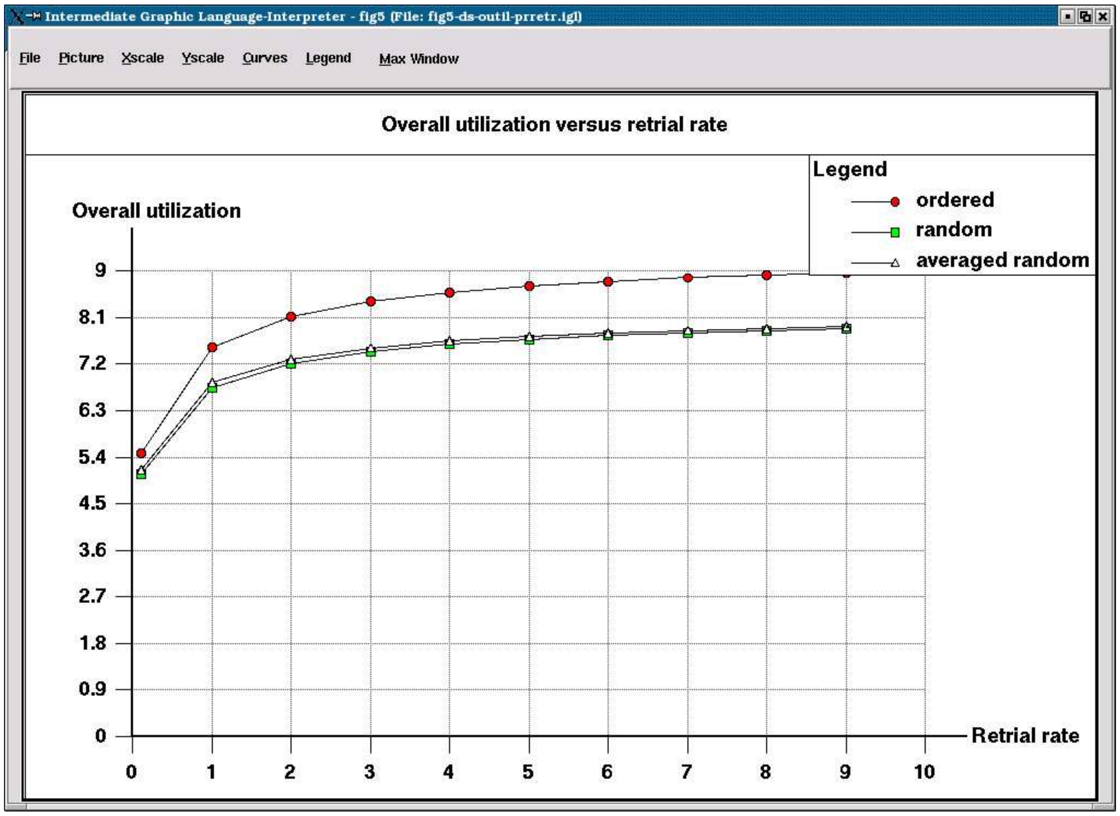

Figure 6: Overall utilization versus retrial rate

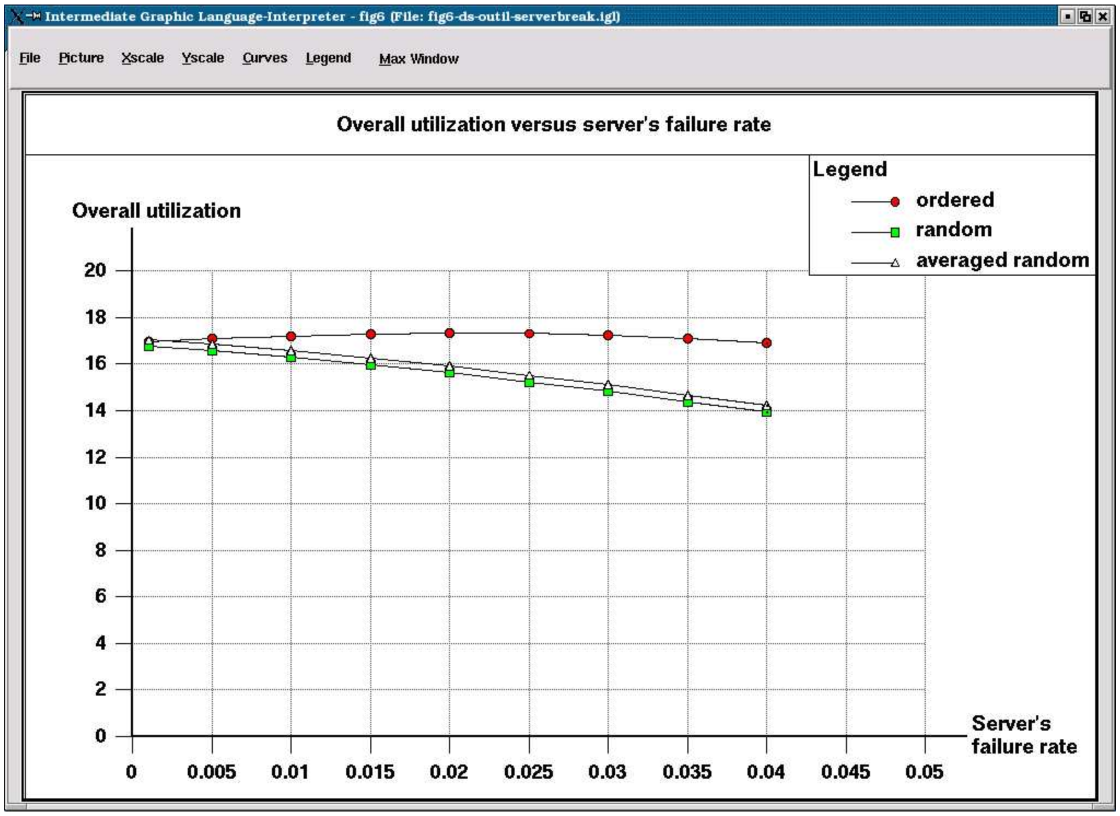

Figure 7: Overall utilization versus server's failure rate

highest utilization and the fastest has the lowest, since it services the request much faster and the requests are assigned to the available and free servers with the same probability. In the beginning of the ordered case, the slowest server has the highest utilization too, but as it fails more often, its service is interrupted more often and looses from its utilization much faster than the faster servers, since it gets requests to serve only if all the other servers busy or failed.

- In Figure 5 the overall utilizations of the systems are displayed versus the primary request generation rate. We can see that the random cases have almost the same while the ordered case has higher utilization.

- In Figure 6 we can see the overall utilization versus the retrial rate. Similarly to Figure 2, after a time the increase of the retrial rate does not affect this measure significantly.

- In Figure 7 the overall utilization is displayed versus the server's failure rate. Like in Figure 3 the mean response time, the overall utilization is getting better for a while in the FFS case as the server's failure rate increases.

4 Conclusions

In this paper, the performance of finite-source retrial queueing systems with homogeneous sources and non-reliable heterogeneous (asymmetric) servers is studied. The novelty of the investigation is the different service rates and different service policies with the non-reliability of the servers. The MOSEL software package was used to formulate the model and to calculate the system performance measures which were graphically displayed to show the differences between the service disciplines in the mean response time, in the utilization of the servers and in the overall system's utilization. It was demonstrated that the Fastest Free Server discipline has more favourable system measures, as it was expected, and in the case of Random Service policy, it is more worthy to apply homogeneous servers with the average service rates of the heterogeneous case, at least with these setup of the parameters.

References

[1] **Falin G.I. and Templeton J.G.C.** *Retrial queues,* Chapman and Hall, London, 1997.

[2] **Artalejo J.R.** Retrial queues with a finite number of sources, *J. Korean Math. Soc.* 35(1998), 503-525.

[3] **Artalejo J.R.** Accessible bibliography on retrial queues, *Mathematical and Computer Modelling* 30(1999), 1-6.

[4] **Falin G.I.** A multiserver retrial queue with a finite number of sources of primary calls, *Mathematical and Computer Modelling* 30(1999), 33-49.

[5] **Li H. and Yang T.** A single server retrial queue with server vacations and a finite number of input sources, *European Journal of Operational Research* 85(1995), 149-160.

[6] **Janssens G.K.** The quasi-random input queueing system with repeated attempts as a model for collision-avoidance star local area network, *IEEE Transactions on Communications* 45(1997), 360-364.

[7] **Tran-Gia P. and Mandjes M.** Modeling of customer retrial phenomenon in cellular mobile networks, *IEEE Journal of Selected Areas in Communications* 15(1997), 1406-1414.

[8] **Onur E., Delic H., Ersoy C. and Caglayan M.U.** Measurement-based replanning of cell capacities in GSM networks, *Computer Networks* 39(2002), 749-767.

[9] **Artalejo J.R.** New results in retrial queueing systems with breakdown of the servers, *Statistica Neerlandica* 48(1994), 23-36.

[10] **Aissani A. and Artalejo J.R.** On the single server retrial queue subject to breakdowns, *Queueing Systems* 30(1998), 309-321.

[11] **Wang J., Cao J. and Li Q.** Reliability analysis of the retrial queue with server breakdowns and repairs, *Queueing Systems* 38(2001), 363-380.

[12] **Almási B., Roszik J. and Sztrik J.** Homogeneous finite-source retrial queues with server subject to breakdowns and repairs, *Computers and Mathematics with Applications* (accepted for publication).

[13] **Rykov V.V.** Monotone control of queueing systems with heterogeneous servers, *Queueing Systems* 37(2001), 391-403.

[14] **Nobel R.D. and Tijms Henk C.** Optimal control of a queueing system with heterogeneous servers and setup costs, *IEEE Trans. Autom. Control* 45(2000), 780-794.

[15] **Pourbabai B.** Markovian queueing systems with retrials and heterogeneous servers, *Computers and Mathematics with Applications* 13(1987), 917-923.

[16] **Begain K., Bolch G. and Herold H.** *Practical performance modeling, application of the MOSEL language*, Kluwer Academic Publisher, Boston, 2001.

A The MOSEL program for the Random Service policy

```
// ==================================== Constant definitions ====
#define NT 20
#define NS 4

// ============================ Variables (input parameters) ====
VAR double prgen;
VAR double prretr;
<1..NS> VAR double prrun#;
<1..NS> VAR double cpubreak_idle#;
<1..NS> VAR double cpubreak_busy#;
<1..NS> VAR double cpurepair#;

// ======================================== Node definitions ====
enum cpu_states {cpu_busy, cpu_idle, cpu_failed};
NODE busy_terminals[NT] = NT;
NODE retrying_terminals[NT] = 0;
NODE waiting_terminals[NS] = 0;
<1..NS> NODE cpu#[cpu_states] = cpu_idle;
        NODE freecpus[NS] = NS;
        NODE failedcpus[NS] = 0;
<1..NS> NODE sr#[NS] = 0;

// ============================================= Transitions ====
FROM cpu1[cpu_idle], busy_terminals, freecpus
        TO cpu1[cpu_busy], waiting_terminals
        W prgen*busy_terminals/freecpus;
FROM cpu2[cpu_idle], busy_terminals, freecpus
        TO cpu2[cpu_busy], waiting_terminals
        W prgen*busy_terminals/freecpus;
FROM cpu3[cpu_idle], busy_terminals, freecpus
        TO cpu3[cpu_busy], waiting_terminals
        W prgen*busy_terminals/freecpus;
FROM cpu4[cpu_idle], busy_terminals, freecpus
        TO cpu4[cpu_busy], waiting_terminals
        W prgen*busy_terminals/freecpus;
FROM busy_terminals
        TO retrying_terminals
        IF freecpus==0
        W prgen*busy_terminals;
FROM cpu1[cpu_idle], retrying_terminals, freecpus
        TO cpu1[cpu_busy], waiting_terminals
        W prretr*retrying_terminals/freecpus;
FROM cpu2[cpu_idle], retrying_terminals, freecpus
        TO cpu2[cpu_busy], waiting_terminals
        W prretr*retrying_terminals/freecpus;
FROM cpu3[cpu_idle], retrying_terminals, freecpus
        TO cpu3[cpu_busy], waiting_terminals
        W prretr*retrying_terminals/freecpus;
FROM cpu4[cpu_idle], retrying_terminals, freecpus
        TO cpu4[cpu_busy], waiting_terminals
        W prretr*retrying_terminals/freecpus;
<1..NS><NS> FROM cpu<#1>[cpu_busy], waiting_terminals{
  TO cpu<#1>[cpu_idle], busy_terminals, freecpus
  W prrun<#1>;
  TO cpu<#1>[cpu_failed],retrying_terminals,failedcpus,sr<#2>(<#1>)
  W cpubreak_busy<#1>;
  }
<1..NS><NS> FROM cpu<#1>[cpu_idle], freecpus
  TO cpu<#1>[cpu_failed], failedcpus, sr<#2>(<#1>)
  W cpubreak_idle<#1>;
<1..NS> IF sr1==# FROM sr1(#), cpu#[cpu_failed], failedcpus
           TO cpu#[cpu_idle], freecpus W cpurepair#;
<2..NS> IF sr<#-1>==0 FROM sr#(sr#) TO sr<#-1>(sr#);

// ================================================ Results ====
<1..NS> RESULT>> if(cpu#==cpu_busy) cpuutil# += PROB;
<1..NS> RESULT>> if(cpu#==cpu_busy) busycpus += PROB;
<1..NS> RESULT>> if(cpu#==cpu_idle OR cpu#==cpu_busy) goodcpus+=PROB;
<1..NS> RESULT>> if(cpu#==cpu_failed) nfailedcpus += PROB;
RESULT if(busy_terminals>0) busyterm += (PROB*busy_terminals);
RESULT>> termutil = busyterm / NT;
RESULT>> if(retrying_terminals>0) retravg+=(PROB*retrying_terminals);
RESULT>> if(failedcpus>0) repairutil += PROB;
RESULT if(waiting_terminals>0) waitall += (PROB*waiting_terminals);
RESULT>> resptime = (retravg + waitall) / NT / (prgen * termutil);
RESULT>> overallutil = busycpus + termutil*NT + repairutil;
```

B The MOSEL program for the Fastest Free Server policy

```
// ==================================== Constant definitions ====
#define NT 20
#define NS 4

// =========================== Variables (input parameters) ====
VAR double prgen;
VAR double prretr;
<1..NS> VAR double prrun#;
```

```
<1..NS> VAR double cpubreak_idle#;
<1..NS> VAR double cpubreak_busy#;
<1..NS> VAR double cpurepair#;

// ======================================== Node definitions ====
enum cpu_states {cpu_busy, cpu_idle, cpu_failed};
NODE busy_terminals[NT] = NT;
NODE retrying_terminals[NT] = 0;
NODE waiting_terminals[NS] = 0;
<1..NS> NODE cpu#[cpu_states] = cpu_idle;
        NODE freecpus[NS] = NS;
        NODE failedcpus[NS] = 0;
<1..NS> NODE sr#[NS] = 0;

// ============================================= Transitions ====
FROM cpu1[cpu_idle], busy_terminals, freecpus
        TO cpu1[cpu_busy], waiting_terminals
        W prgen*busy_terminals;
FROM cpu2[cpu_idle], busy_terminals, freecpus
        TO cpu2[cpu_busy], waiting_terminals
        IF cpu1==cpu_busy
        W prgen*busy_terminals;
FROM cpu3[cpu_idle], busy_terminals, freecpus
        TO cpu3[cpu_busy], waiting_terminals
        IF cpu1==cpu_busy
        AND cpu2==cpu_busy
        W prgen*busy_terminals;
FROM cpu4[cpu_idle], busy_terminals, freecpus
        TO cpu4[cpu_busy], waiting_terminals
        IF cpu1==cpu_busy
        AND cpu2==cpu_busy
        AND cpu3==cpu_busy
        W prgen*busy_terminals;
FROM busy_terminals
        TO retrying_terminals
        IF freecpus==0
        W prgen*busy_terminals;
FROM cpu1[cpu_idle], retrying_terminals, freecpus
        TO cpu1[cpu_busy], waiting_terminals
        W prretr*retrying_terminals;
FROM cpu2[cpu_idle], retrying_terminals, freecpus
        TO cpu2[cpu_busy], waiting_terminals
        IF cpu1==cpu_busy
        W prretr*retrying_terminals;
FROM cpu3[cpu_idle], retrying_terminals, freecpus
        TO cpu3[cpu_busy], waiting_terminals
        IF cpu1==cpu_busy
        AND cpu2==cpu_busy
        W prretr*retrying_terminals;
FROM cpu4[cpu_idle], retrying_terminals, freecpus
```

```
        TO cpu4[cpu_busy], waiting_terminals
        IF cpu1==cpu_busy
        AND cpu2==cpu_busy
        AND cpu3==cpu_busy
        W prretr*retrying_terminals;
<1..NS><NS> FROM cpu<#1>[cpu_busy], waiting_terminals{
  TO cpu<#1>[cpu_idle], busy_terminals, freecpus
  W prrun<#1>;
  TO cpu<#1>[cpu_failed],retrying_terminals,failedcpus,sr<#2>(<#1>)
  W cpubreak_busy<#1>;
  }
<1..NS><NS> FROM cpu<#1>[cpu_idle], freecpus
  TO cpu<#1>[cpu_failed], failedcpus, sr<#2>(<#1>)
  W cpubreak_idle<#1>;
<1..NS> IF sr1==# FROM sr1(#), cpu#[cpu_failed], failedcpus
           TO cpu#[cpu_idle], freecpus W cpurepair#;
<2..NS> IF sr<#-1>==0 FROM sr#(sr#) TO sr<#-1>(sr#);

// ================================================== Results ====
<1..NS> RESULT>> if(cpu#==cpu_busy) cpuutil# += PROB;
<1..NS> RESULT>> if(cpu#==cpu_busy) busycpus += PROB;
<1..NS> RESULT>> if(cpu#==cpu_idle OR cpu#==cpu_busy) goodcpus+=PROB;
<1..NS> RESULT>> if(cpu#==cpu_failed) nfailedcpus += PROB;
RESULT if(busy_terminals>0) busyterm += (PROB*busy_terminals);
RESULT>> termutil = busyterm / NT;
RESULT>> if(retrying_terminals>0) retravg+=(PROB*retrying_terminals);
RESULT>> if(failedcpus>0) repairutil += PROB;
RESULT if(waiting_terminals>0) waitall += (PROB*waiting_terminals);
RESULT>> resptime = (retravg + waitall) / NT / (prgen * termutil);
RESULT>> overallutil = busycpus + termutil*NT + repairutil;
```