Annales UMCS Informatica AI 5 (2006) 19-27

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

A Markovian model of the RED mechanism solved with a cluster of computers

Jarosław Bylina*, Beata Bylina

Department of Computer Science, Institute of Mathematics, Marie Curie-Skłodowska University, Pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Abstract

The paper presents a working example of distributed application which can be used to find stationary probabilities of states for queuing models – by generating a transition rate matrix and solving a linear system. The presented example is connected to the RED mechanism which can be used in the TCP/IP protocol to control packets flow. The paper also shows efficiency of the application with the use of a various number of computers connected with Ethernet.

1. Introduction

Queuing models of the communication networks are a versatile tool for investigating various characteristics of such networks – both in their project stage and during their exploitation and extension. Queuing models are able to represent properties of networks and can be relatively easily analyzed with simulation tools and numerical methods. Among many numerical approaches (mean value analysis [1], diffusion approximation [2], network calculus [3] and others) we are most interested in Markov chains [4-8].

Markov chains are discrete state space stochastic processes with an interesting feature (*Markovian property*). Namely, probabilities of the future states of a Markov chain depend only on their current probabilities – not on any past probabilities (Markov chains "lack of the memory" as it is sometimes colloquially said).

Stationary probabilities of Markov chain states (and thus states probabilities of queuing model and modeled system) are relatively easy to obtain from the following equatoin:

$$\boldsymbol{\pi}\mathbf{Q} = \mathbf{0},$$

*Corresponding author: *e-mail address*: jmbylina@hektor.umcs.lublin.pl

where $\boldsymbol{\pi} = (\pi_1, ..., \pi_N)$ is a vector of state probabilities (π_i is the stationary probability of the ith state) which is to be found (so $\sum_{i=1}^{N} \pi_i = 1$ and every $\pi_i > 0$); and $\mathbf{Q}$ is the *infinitesimal generator* of the given Markov chain. The infinitesimal generator is a sparse square matrix of the size $N \times N$ where N is the number of the chain states. Each element q_{ij} of the matrix $\mathbf{Q}$ describes a *transition intensity* from the ith state to the jth state which is given by:

$$q_{ij} = \lim_{\Delta t \to 0} \frac{p_{ij}(\Delta t)}{\Delta t},$$

$$q_{ij} = -\sum_{j \neq i} q_{ij},$$

where $p_{ij}(\Delta t)$ denotes a probability that the chain (model) being in the ith state will change its state to j during an interval Δt.

The matrix $\mathbf{Q}$ is often shown as its corresponding *transition graph* of the Markov chain (as in this paper, Fig. 3).

One difficult issue about modeling with Markov chains are huge numbers of states and hence huge sizes of the matrix $\mathbf{Q}$ – millions of states are not impossible if we want to model a system with a good accuracy (better accuracy equals to more states).

Solving such a large linear system needs a great amount of computer memory and significant speed of computation. Such possibilities are offered by modern computer systems: vector and parallel machines, supercomputers, clusters and grids.

2. A model of the RED mechanism

One of the best known algorithms helping to avoid connection congestion is RED – *Random Early Detection* [9,10]. It is an active queue management mechanism implemented in buffers of IP routers. The RED mechanism drops packages not only when there is no place in the queue, but also earlier – with a variable probability. The way the RED works is following. When a new packet arrives to the buffer, a weighted average queue length n_{avg} is calculated:

$$n_{avg} \leftarrow (1 - w) n_{avg} + w n,$$

where n is the current size of the buffer queue and w is a fixed small positive real number. Next, the decision is made about the future of the packet. When there is no place in the queue, the packet is obviously dropped. Otherwise, the probability $p_d(n_{avg})$ of dropping the packet is given by:

$$p_d\left(n_{avg}\right)=\left\{\begin{array}{lll} 0, & for & n_{avg} < n_{min,} \\ \dfrac{n_{avg}-n_{min}}{n_{max}-n_{min}}\cdot p_{max}, & for & n_{avg} \in \langle n_{min}, n_{max} \rangle, \\ 1, & for & n_{avg} > n_{max}, \end{array}\right\},$$

where m_{min}, n_{max} and p_{max} are fixed parameters (see also Fig. 1).

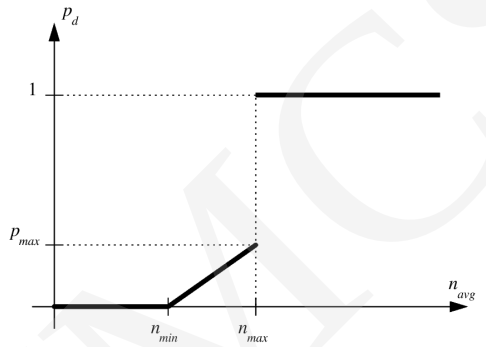

Fig. 1. Probability of packet dropping in the RED mechanism

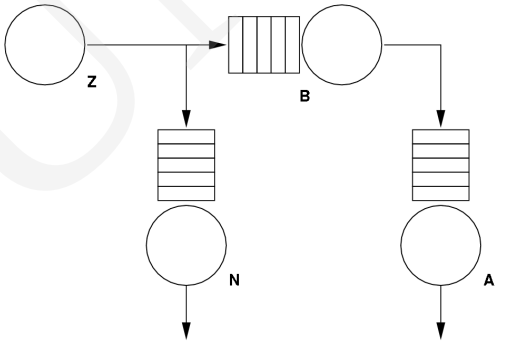

Fig. 2. A queuing model of the RED mechanism

We proposed a queuing model for a buffer with such a mechanism (Fig. 2). The model consists of a source **Z** and three service stations **B**, **A** and **N**. The source **Z** is a simple Poisson source (intervals between packets are distributed exponentially) but its intensity is not constant. First, it is an *on-off* source, which means that it can switch off (change its intensity to zero if its intensity is positive) with the intensity μ_{off} or switch on (change its intensity to minimal positive if its intensity is zero) with the intensity μ_{on} Moreover, intensity of **Z** (if it is on) can adopt values: λ, 2λ, 3λ,…, $L\lambda$ depending on the behavior of routed packets (see below).

The packet generated in **Z** is dropped (with the probability $p_d(n_{avg})$ described above) which means that it goes to **N** (really it disappears but a negative answer

returns to packet's source); or else the packet is taken – which means it goes to **B**. Packets served in **B** go to **A** (which means they go on from the router to its destination, but acknowledgments go to packet's source).

Stations **A** and **N** simulate a feedback. Namely, in real networks when a source gets an acknowledgment it increases its intensity by one step. However, when it gets a negative answer it halves its intensity. Such behavior is implemented in our model: when a packet leaves the station **A**, the source **Z** (if it is on) increases its intensity from $l\lambda$, to $(l + 1)\ \lambda$ when a packet leaves the station **N**, the source **Z** (if it is on) decreases its intensity from $l\lambda$ to $|(l/2)\lambda|$.

All stations (**B**, **A** and **N**) have exponential service times.

The states of the described model can be represented by the vectors $(l,n_{\mathbf{B}}, n_{\mathbf{A}},n_{\mathbf{N}},a)$ where $l\lambda$ gives the current intensity of the source **Z** (or $l = 0$ when it is off), $n_{\mathbf{B}}$, $n_{\mathbf{A}},n_{\mathbf{N}}$ are the numbers of packets waiting in respective stations and a is an auxiliary number approximating the weighted average queue length n_{avg} (a^* denotes its new value, changed after a packet arrival). Transitions between the states are schematically shown in Figure 3.

Such a model gives quite a lot of states – which is shown in section 4 (Table 1).

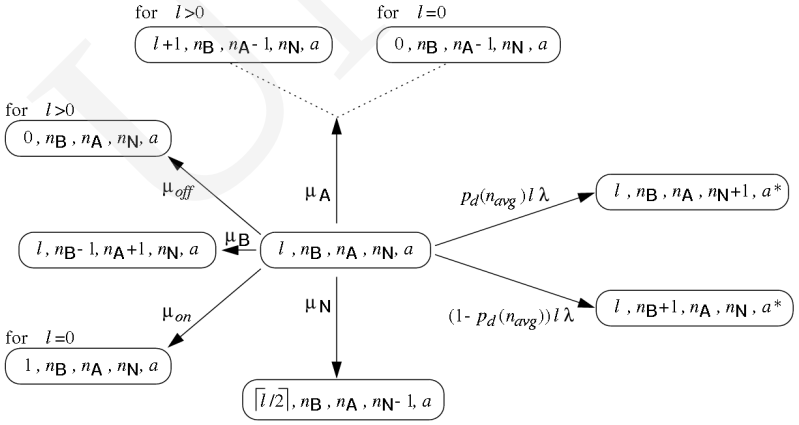

Fig. 3. Transitions scheme for a Markov chain corresponding to the presented model of the RED mechanism

3. A cluster application

We created and parallelized some algorithms suitable for finding states probabilities for such models as described in section 2.

First, we needed an algorithm which can generate the matrix **Q** from a simple description. Such an algorithm (and its implementation) was presented in [7,11,12]. Moreover, this algorithm is adapted to work in a distributed environment – for any kinds of computers connected with some network (especially clusters).

Our algorithm is a modified Breadth-First Search algorithm (an algorithm for traversing all the vertices of a graph) and it is based on the *master-slave* idea – one of the processes (machines) is a *master* (a kind of supervising process) and all the others are *slaves* (doing the actual work). Each slave generates its own portion of the transition matrix (states are divided into separate pools a priori). Each slave starts from a different state and generates all states adjacent to it – and so on. If it happens that the slave generates a state not belonging to it, then it sends the state to the respective machine (that owns the state). The master checks if all the machines finished the generation and when they did, it gathers (and then broadcasts) some summary information.

The slave algorithm for the generation of the matrix **Q** is following.

1. Get from the master description (in the form of simple conditions) of the pools of states (that is, how to find out what pool a given state belongs to). This is the **only input** that is distributed to slaves – everything else is generated.
2. Initialize data structures (an empty queue **L**, among others).
3. Create a (random) state belonging to your pool and attach it to **L**.
4. Take the first state **v** from **L**, find all adjacent states **w** and detach **v** from **L**.
5. For every state **w**:
   – compute the transition rates from **v** to **w**,
   – if **w** is in your pool and has not been generated yet,
     then attach it to **L**,
     else ask another slave (**w'**s owner) about **w**'s index in **Q**,
   – insert the transition rate into the matrix **Q**.
6. If any slave asks you about a state and this state has not been generated yet, then attach it to **L**.
7. If your queue **L** is empty, then let the master know it, else go to 4. When the master gets such information from every slave, the algorithm is over. Else go to 4.

Steps 5 and 6 are conducted in parallel.

After these steps every slave has its own (horizontal) part of the matrix **Q**, as shown below (*p* is the number of slaves):

$$\mathbf{Q} = \begin{pmatrix} \mathbf{Q}_0 \\ \hline \mathbf{Q}_1 \\ \hline \vdots \\ \hline \mathbf{Q}_{p-2} \\ \hline \mathbf{Q}_{p-1} \end{pmatrix}.$$

The second step was to parallelize one of the methods of solving huge linear systems [13]. We decided to use iterative GMRES [14], because of its relative

good speed and accuracy and ease of vectorization [15] and parallelization [5,11].

We decided to choose a master-slave approach. It was rather natural decision because after the generation we have the transition rate matrix **Q** distributed evenly (more or less) among all the generating slaves. Now, those slaves are going to solve the equation $\boldsymbol{\pi}\mathbf{Q} = 0$ together. In this part of our application the master has more work to do than in the generation. Namely, it makes all the operations on vectors (there are quite a lot of them in GMRES) with the use of the ATLAS library [16]. However, the matrix **Q** resides in the slaves, so operations on the matrix (that is, multiplying it by various vectors) are held as follows. Respective parts of a vector to be multiplied are sent to the slaves by the master, the slaves multiply them by their own parts of the matrix **Q** and send results to the master, which totals them.

So, the second part of the application – namely solving the equation $\boldsymbol{\pi}\mathbf{Q} = 0$ – starts where the generation ended, and it is following (steps for the master, slaves are used only for the matrix-vector multiplication, which is a greater part of all the computations).

1. Choose a (random) initial solution vector **x0**.
2. Multiply the matrix **Q** by **x0**, that is:
   - send a part of the vector **x0** to each of the slaves (each slave gets elements of **x0** with indices corresponding to row indices of the slave's part of the matrix **Q**),
   - each slave multiplies its part of **Q** by its part of **x0**,
   - receive the partial results from the slaves and sum them up to get overall result.
3. Find the orthonormal basis and the Hessenberg matrix with the use of Arnoldi process (there are many multiplications in this step – all are conducted as in step 2).
4. Find a new approximate solution vector **x0** from the above basis and Hessenberg matrix [14].
5. If the approximation is not good, go to 2.

Our application is written as a set of C-language files, and the whole communication between machines is implemented with the use of BSD sockets what makes it portable to many operating systems (Linux, Unixes, etc.). Moreover, with the use of such a low-level tool we have more control over the communication and contents sent (so we can keep it reasonably low).

4. A numerical experiment

We tested our application in the following cluster environment: 15 Linux-powered computers (Intel Pentium4 2.80GHz, 512MB RAM) connected with Ethernet 100Mb/s. We tested it for various models, the RED mechanism model described in section 2 among others. Our model was tested with three sets of

parameters which are shown in the left part of Table 1. Here we can also see the size of each model (the number of its states).

Table 1. Parameters and times of computations of tested models

maximal				number of states	g	G*g*	*r*	R*r*	s	G*s*+R*s*
l	n_B	n_A	n_N							
8	4	4	4	1125	1	4	1	1	1	5
8	6	12	12	39546	7	9	5	54	5	68
8	8	8	8	45684	7	12	6	72	6	86

The second part of Table 1 presents times (in seconds) of computations (G*g*, R*r*, G*s*+R*r*) for the selected number of computers (*g*, *r*, *s*). The times are overall times of the respective parts of the application activity, measured with the function `time()`. Exact meaning of these notations is following:

- *g* – the number of computers, for which the generation was the fastest;
- G*g* – the generation time for *g* computers;
- *r* – the number of computers for which solving was the fastest;
- R*r* – the solving time for *r* computers;
- *s* – the number of computers for which the total working time was the shortest;
- G*s*+R*s* – the total working time for *s* computers.

Figures 4-6 show working times for different model parameters and for various numbers of computers.

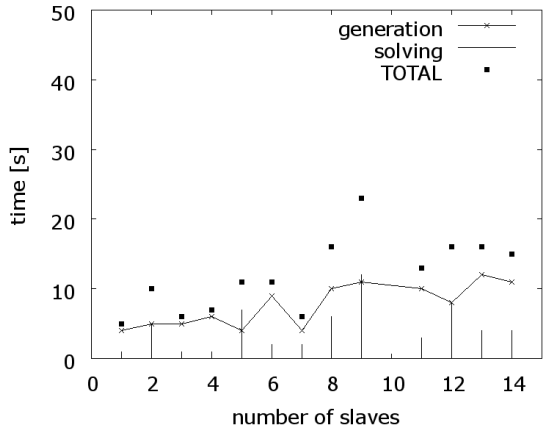

Fig. 4. Computation times for the model with 1125 states

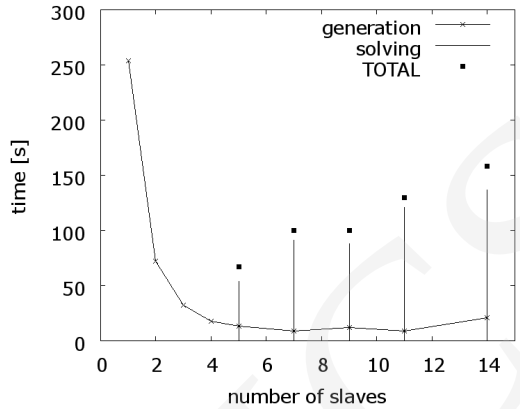

Fig. 5. Computation times for the model with 39546 states

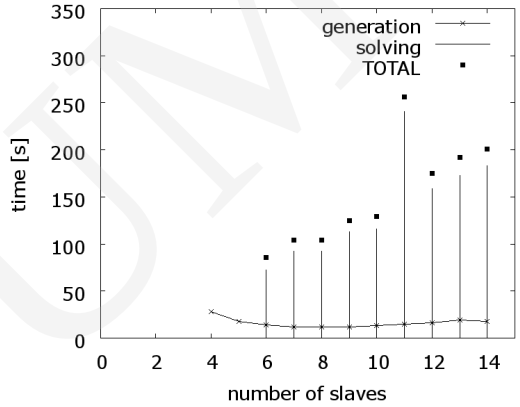

Fig. 6. Computation times for the model with 45684 states

Conclusions

Our experiments show usefulness of distributed application for generating and solving queuing models. Particularly, the first part of the implementation – the matrix generation – gains a lot of efficiency when about seven computers are used – especially for large numbers of space (here: about 40000).

The second part (the linear system solving) unfortunately does not use the whole computing power which is in the computers (because of communication issues), but for a large number of states solving such a system becomes at least possible for some number of connected computers. Although GMRES is well-known as a scalable algorithm [17] we could not show the adequate scalability for our application.

References

[1] Reiser M., Kobayashi H., *Queuing Network Models with Multiple Closed Chains: Theory and Computational Algorithms*, IBM J. Res. Develop., 19 (1975).
[2] Fourneau J.-M., Pekergin N., *Brief Introduction to an Algorithmic Approach for Strong Stochastic Bounds*, Proceedings from EuroNGI Workshop: New Trends in Modeling, Quantitive Methods and Measurements, Jacek Skalmierski Computer Studio, Gliwice, (2004) 57.
[3] Le Boudec J.-Y., Thiran P., *Network Calculus. A Theory of Deterministic Queuing Systems for the Internet*, LNCS 2050, Springer Verlag, (2001).
[4] Bolch G., Greiner S., de Meer H., Trivedi K. S., *Queuing Networks and Markov Chains, Modeling and Performance Evaluation with Computer Science Application*, John Wiley, New York (1998).
[5] Bylina J., *Distributed solving of Markov chains for computer network models*, Annales UMCS Informatica 1, (2003) 15.
[6] Bylina B., *The inverse iteration with the WZ factorization used to the Markovian models*, Annales UMCS Informatica, 2 (2004) 15.
[7] Bylina J., *Distributed generation of Markov chains infenitesimal generator matrices queueing network models*, Annales UMCS Informatica, 2 (2004) 25.
[8] Bylina B., Bylina J., *Using Markov chains for modelling networks*, Annales UMCS Informatica, 3 (2005) 27.
[9] Floyd S., Jacobson V., *Random Early Detection Gateways for Congestion Avoidance*, IEEE/ACM Transaction on Networking, 1(4) (1997) 397.
[10] Hassan M., Jain R., *Wysoko wydajne sieci TCP/IP*, Helion, Gliwice, (2004), in Polish.
[11] Bylina J., *A distributed approach to solve large Markov chains*, Proceedings from EuroNGI Workshop: New Trends in Modeling, Quantitive Methods and Measurements, Jacek Skalmierski Computer Studio, Gliwice, (2004) 145.
[12] Bylina J., Bylina B., *Distributed generation of Markov chains infinitesimal generators with the use of the low level network interface*, Proceedings of 4rd International Conference Aplimat 2005, part II, Bratislava, (2005) 257.
[13] Bylina B., Bylina J., *A review of numerical methods for solving large Markov chains*, Proceedings from EuroNGI Workshop: New Trends in Modeling, Quantitive Methods and Measurements, Jacek Skalmierski Computer Studio, Gliwice, (2004) 75.
[14] Saad Y., Schultz M. H., *GMRES: A generalized minimal residual algorithm for solving non-symmetric linear systems*, SIAM Journal of Scientific and Statistical Computing, 7 (1986) 856.
[15] Bylina J., Bylina B., *GMRES dla rozwiązywania łańcuchów Markowa na komputerze wektorowym CRAY SV1*, Algorytmy, metody i programy naukowe, Polskie Towarzystwo Informatyczne, Lublin, (2004) 19, in Polish.
[16] http://www.netlib.org/atlas/
[17] Sosonkina M., Allison D.C.S., Watson L.T., *Scalability analysis of parallel GMRES implementations*, Parallel Algorithms and Applications, 17 (2002) 263.