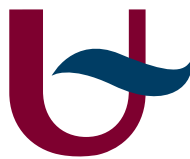


# The power of two paths in grid computing networks

Wouter ibens  
University of Antwerp  
Supervisor: Prof. dr. Benny Van Houdt  
May 28, 2012



## **Abstract**

In ring structured distributed systems, busy nodes will forward new jobs to other nodes. This thesis focusses on the algorithms for choosing a successor node for a job. ...

## Acknowledgements

This thesis is not only the work of myself, I could never accomplish this without the people around me. I would like to take this opportunity to thank some of them specifically.

First and foremost, my supervisor Professor dr. Benny Van Houdt. Firstly, he taught me a whole new field in the area of computer science, the part that interested me the most during the course of my studies. Secondly, for being my supervisor: he helped me out when he could and suggested ideas when I was stuck. I was always welcome to drop by and reflect thoughts.

I would also like to thank my family, especially my parents. They gave me the chance to complete my education without worrying just one second about the financial cost. They gave me the freedom of making my own choices and motivated me when I needed it. My sister Anneleen should be mentioned for proofreading this thesis and other tasks in english during my education.

Also, my friends have been a great help to me. Playing games, eating together every evening and just having fun together is what made the past 5 years with no doubt the best years of my life so far. Thank you all.

Finally, I would like to thank my girlfriend Nicky. Although she might have caused some failed exams at the beginning of our relationship, she has been a great help further on. She motivated me to take hard but rewarding options when easy ones were available. A girl that understands me and never fails to cheer me up.

# Contents

<b>1</b>	<b>Setup</b>	<b>5</b>
1.1	Forwarding algorithms . . . . .	5
1.2	Forward to neighbour . . . . .	6
1.2.1	Forward right . . . . .	6
1.2.2	Left/Right forward . . . . .	6
1.2.3	Random Left/Right forward with parameter $p$ . . . . .	6
1.2.4	Position-dependant forwarding . . . . .	6
1.3	Forward anywhere . . . . .	7
1.3.1	Random unvisited . . . . .	7
1.3.2	Round Robin unvisited . . . . .	7
1.3.3	Coprime offset . . . . .	7
1.3.4	Random Coprime offset . . . . .	7
<b>2</b>	<b>Simulation</b>	<b>8</b>
2.1	Measure . . . . .	8
2.2	Results . . . . .	9
2.3	Multiple execution units . . . . .	11
<b>3</b>	<b>Numerical Validation</b>	<b>12</b>
3.1	Forward Right . . . . .	12
3.2	Random Left/Right forward with parameter $p$ . . . . .	12
3.3	Random Coprime offset . . . . .	13
3.4	Random Unvisited . . . . .	16
3.5	Lumped states . . . . .	16
3.6	Equivalent techniques . . . . .	18
<b>4</b>	<b>Conclusion</b>	<b>18</b>
<b>A</b>	<b>Simulator source code</b>	<b>18</b>
<b>B</b>	<b>MATLAB Numerical evaluation code</b>	<b>40</b>

# Introduction

This thesis researches the behaviour of forwarding algorithms in a ring-structured distributed system. Section 1 precisely describes the setup of the system. It specifies the general assumptions made in this document and gives a short overview of the different algorithms we have reviewed.

Section 2 gives a short introduction about the simulator we wrote and how to use it. Furthermore, it contains the results of the tests, performed by the simulator.

In the next part, section 3, we reviewed the output of the simulator. Using numerical algorithms we try to match our results with a Markov Chain model.

Finally, section 4 contains the conclusions and other thoughts on the algorithms.

**\*\* COMPLETE LATER ON**

## 1 Setup

We are using a ring-structured network of  $N$  nodes. Each node is connected to two neighbours, left and right. The purpose of these nodes is to process incoming jobs. When a node is busy while a job arrives, it must forward to another node. When a job has visited all nodes and none of them was found empty, the job is dropped.

Jobs have an arrival time, a length and optional metadata. They arrive at each node independently as a poisson process at rate  $\lambda$ . Their length is exponentially distributed with mean  $\mu$  (unless otherwise noted, assume  $\mu = 1$ ). Although each job has a length, this length may not be known in advance. Finally, the metadata is optional and may be used by the nodes to pass information among the job (e.g. a list of visited nodes).

Nodes can use different algorithms to determine where to a job will be forwarded. The performance of these algorithms is the main focus of this thesis. Different techniques will be discussed and simulated. Afterwards, some results of the simulation will be validated. Note that the cost of forwarding a job is neglected. Together with the presumption a job must visit each node before being dropped, this means a job arriving at any node will be processed if and only if at least one server is idle.

The performance of a forwarding algorithm is measured by the average number of hops a job must visit before being executed. The goal of the algorithms is to minimize this number by spreading the load evenly along the ring.

### 1.1 Forwarding algorithms

Nodes that must forward a job must choose another node of the ring. Nodes have no information about other nodes, so it has no idea whether the node is idle or busy. The algorithms are grouped in two categories: forward to neighbour and forward anywhere. The first technique allows a busy node to forward an incoming job to either its left or right neighbour, where the latter may forward these jobs to any node in the ring. Since the amount of dropped jobs is equal for each forwarding algorithm. These jobs will be ignored when computing the

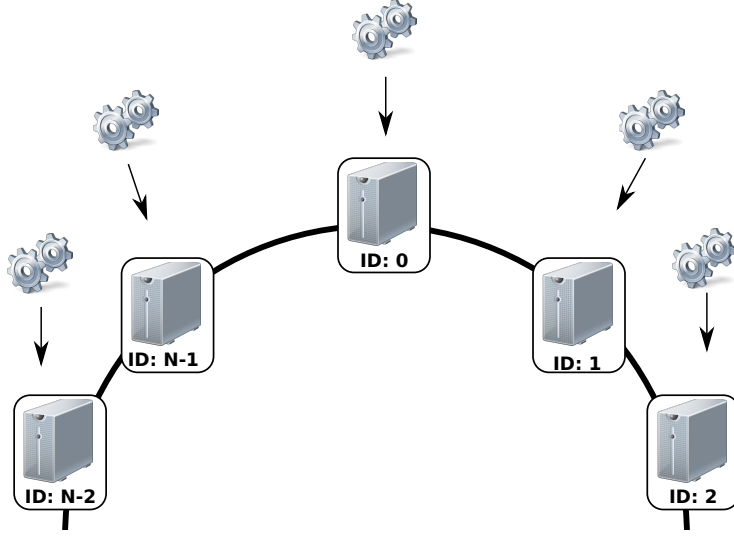


Figure 1: A ring structured network

average number of hops. One should note that the loss rate of jobs in the system is the same as the Erlang-b loss rate.

## 1.2 Forward to neighbour

### 1.2.1 Forward right

A busy node using this technique will forward a job to its right neighbour. The job will keep travelling clockwise until an idle node is found, where it will be processed. This algorithm is used as base line in all further tests.

### 1.2.2 Left/Right forward

A variant to the previous algorithm is the Left/Right forward technique. Instead of forwarding each job to its right neighbour, a busy node will alternate the direction after forwarding such a job. To avoid a job coming back, this initial direction is saved in the job's metadata. Busy nodes receiving a job from another neighbour must forward it the same direction as specified in the job's metadata.

### 1.2.3 Random Left/Right forward with parameter $p$

This technique is a variant of the Left/Right forward algorithm. However, instead of alternating the direction for each new job, a node will forward a job to its right with probability  $p$  and to its left with probability  $1 - p$ . As the previous technique, the direction is saved in the job's metadata and subsequent nodes must maintain this direction when forwarding.

### 1.2.4 Position-dependant forwarding

As shown in figure 1, each node in the ring has an unique ID. Except the for node with id 0 and  $N - 1$ , neighbouring are succeeding. When nodes uses this

algorithm, nodes will always forward a new job in the same direction: to the right when the node's id is even, to the left otherwise. As previous algorithms, the direction is saved in the job's metadata and this direction must be used if other nodes must forward the job.

### 1.3 Forward anywhere

The ring structure can be used in real networks, however in many cases the ring is no more than a virtual overlay over another structure (e.g. the internet). In these networks each node is able to connect to each other node and other forwarding algorithms can be used.

#### 1.3.1 Random unvisited

The Random unvisited algorithm is the most basic algorithm in this category. Everytime a job is forwarded, a list of unvisited nodes is generated and a random node is chosen from this list. The current node is added to the list of visited nodes, which is found in the job's metadata.

#### 1.3.2 Round Robin unvisited

This algorithm is similar to Random unvisited, but instead of choosing the next node at random a different technique is used. When a node must forward a job, it saves its id. When another job is forwarded, it will be forwarded to the saved id + 1. However, when that id is a visited node, the job will be forwarded to the next node that is unvisited.

#### 1.3.3 Coprime offset

Another algorithm is Coprime offset. This algorithm generates a list of all numbers smaller than  $N$ , and coprime to  $N$ . The first time a job is forwarded the next number of this list is selected. This is the job's forward offset and saved in its metadata. When a job is forwarded, it is sent exactly this many hops farther. Because this number and  $N$  are coprime, it will visit all nodes exactly once before being returned to its originating node.

Example: Consider a ring size of  $N = 10$  in which every node is busy. The list of coprimes is then generated: 1, 3, 7, 9. Assume a job arrives at node 3 and the last time node 3 forwarded a job it was given offset 1. Because this node is busy, the next number on the list (3) is selected and saved in the job's metadata. All nodes are busy so the job visits these nodes before being dropped: 3 (arrival), 6, 9, 2, 5, 8, 1, 4, 7, 0. Node 0 will drop the job because the next node would be 3, which is the node on which the job arrived.

#### 1.3.4 Random Coprime offset

The Random Coprime offset algorithm is almost equal to Coprime offset. The difference between them is the decision of the offset value. Where it is the next number on the list in Coprime offset, a random value is taken from the list when using Random Coprime offset.

## 2 Simulation

To evaluate the different algorithms discussed in the previous section, 2 methods will be used. Firstly using a simulation, the second method is the evaluation of this simulation using MATLAB. The validation method is further discussed in section 3.

The simulation is accomplished using a custom simulator. A continuous time simulator is written in C++, using no external requirements but the STL and OpenMP [2]. The source code of the simulator can be found in appendix A or on <http://code.google.com/p/powerofpaths/>.

The simulator can be controlled using a command line interface, its usage is described below.

```

1 Usage: -r -s long -j double -a double -n long -p long -l long -t
      long -h type
2       -r      Random seed
3       -s      Set seed                      (default: 0)
4       -j      Job length                    (default: 1.0)
5       -a      Load                          (default: 1.0)
6       -n      Ring size                     (default: 100)
7       -c      Processing units per node    (default: 1)
8       -p      Print progress interval      (default: -1 -
              disabled)
9       -l      Simulation length             (default: 3600)
10      -t      Repetition                     (default: 1)
11      -h      Print this help
12      type    right | switch | randswitch | evenswitch | prime |
              randprime | randunvisited | totop

```

Listing 1: Simulator usage description

### 2.1 Measure

The goal of the algorithms is to distribute the jobs evenly along the ring. This implies the number of hops a job must travel should be low. As a measure for our experiments, we will be using the number of times a job was forwarded before it was executed. Since the number of forwards of a job that could not be executed is the same for each algorithm, and the loss rate of each algorithm is the same, we will not take these jobs into account when computing the average.

It is clear that when the system load approaches 0, the probability that a node is busy will also approach 0 and the average number of forwards will therefore also approach 0. On the other hand, when the load approaches  $\infty$ , each node's probability of being busy will approach 1 and therefore the number of forwards will be  $N - 1$  and the job will fail. A system with load  $> 1$  is called an overloaded system.

We will compare each algorithms to a baseline result. The baseline used in this thesis is the Forward right algorithm, meaning that each graph will show its result relative to the Forward right results. The results given by the simulator were obtained using a ring size of 100 and using a random seed for each run.

The absolute performance of the baseline algorithm is shown in figure 2.



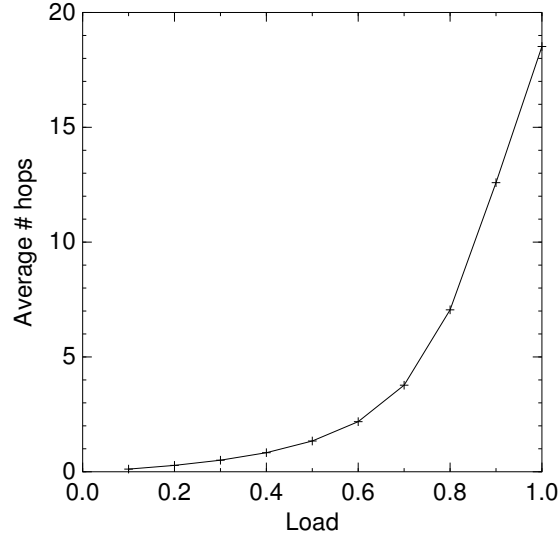


Figure 2: The Forward Right baseline result

## 2.2 Results

### Left/Right Forward

It is intuitively clear that alternating the forwarding direction of arriving jobs should distribute the load better than keeping the same direction. Figure 3 shows the improvement made by the Left/Right forward algorithm over the Forward right method. The performance gain is at least 1% and up to over 4% under medium load.

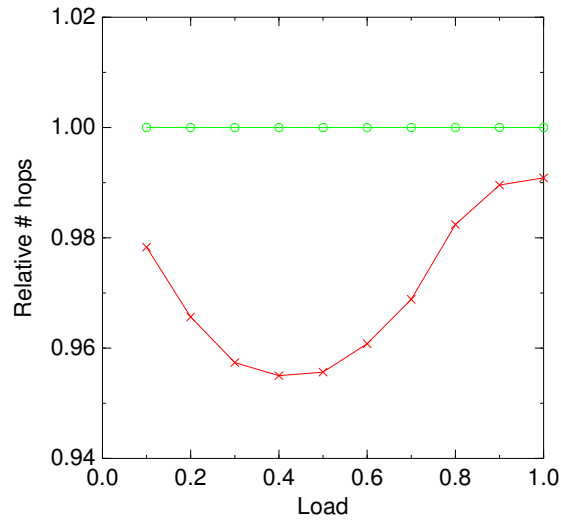


Figure 3: Left/Right

### Random Left/Right forward with parameter $p$

For  $p = 0.5$ , one would expect the results of this algorithm being similar to those obtained in the previous simulation. However, it seems the small change in the algorithm worsened the results significantly.

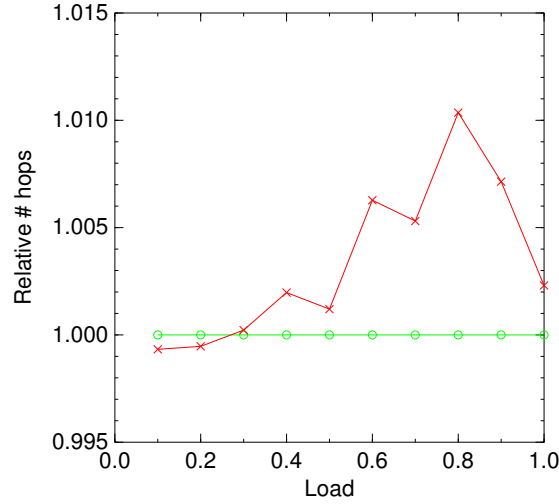


Figure 4: Random Left/Right forward with parameter 0.5

Figure 4 shows the results of this algorithm for  $p = 0.5$ . How this parameter influenced the performance is shown in figure 5. For  $p = 0$ , this algorithm is equivalent to the Forward right method. The performance decreases fast when increasing  $p$ , until around 0.4, where it increases a little until arriving at 0.5.

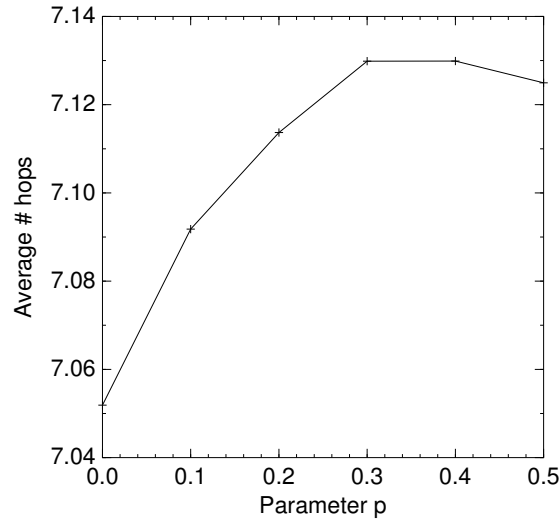


Figure 5: Random Left/Right forward with load 0.8

### Position-dependant forwarding

This technique groups nodes in virtual clusters. When a job arrives in a node and that node is busy, the job will be forwarded to the other node in the cluster. Jobs leaving a cluster will do this in a random direction ( $p = 0.5$ ). Since the load is concentrated per cluster instead of being distributed over the whole system, this technique performs worse than other techniques. The results are represented in figure 6.

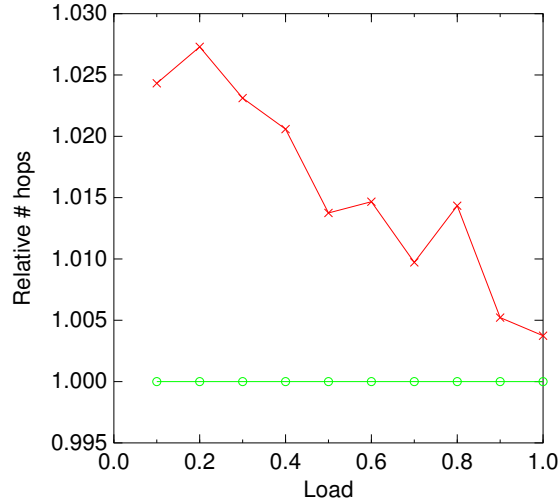


Figure 6: Position-dependant forwarding

### Random unvisited

This algorithm is the most straight forward and is the best performing from any of these techniques. However, it should be noted that each visited node must be stored into the job's metadata.

## 2.3 Multiple execution units

So far, all simulations were executed using 1 cpu per server. This is not a realistic assumption for most distributed systems. This section is intended to research the behaviour of the algorithms when using multiple cpu's, and comparing the results to a system with 1 cpu per server, and to each other.

Per algorithm, the two simulations are performed and compared, see table 1 for more information about the tests.

Algorithm	Forward right, Left/Right, Random left/right (0.5), Random unvisited, Random Coprime offset	
Ring size	100	25
Cpu's per node	1	4
Load	0.1 – 1.0	

Table 1: Comparison of algorithms using multiple cpu's per server

Since the ring size for the second test is 25 instead of 100, the number of forwards of the second test will be multiplied by 4 to get meaningful results. This actually makes sense because a job that is forwarded once has encountered 4 busy cpu's.

### 3 Numerical Validation

\*\* BEPAALDE ALGORITMES NIET BESPROKEN, WAAROM?

To validate the results obtained in the previous section, we modelled the scheduling-techniques into Markov Chains. Using the steady state distribution of these chains, we can derive the average number of hops and the average loss. For  $N$  nodes in a ring, the markov chain consists of  $2^N$  states, where the  $n$ -th bit represents whether the  $n$ -th server is busy (1) or idle (0). To optimize the computation time and memory requirements, we used sparse matrixes for the validation. The validation code is written in MATLAB, it can be found in appendix B or on <http://code.google.com/p/powerofpaths/>.

The validation of the results happens in a different environment than the simulation. Because of the non-polynomial execution time of the algorithm, the size of the ring is reduced to 10. Therefore, the results of this validation are smaller but the relative results can still be analysed.

#### 3.1 Forward Right

Modelling a technique into a Markov Chain is an easy operation for most algorithms. The example given below is for a ring of 3 nodes. For convenience, the states are represented by their binary form.

$$Q = \begin{matrix} & \begin{matrix} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{matrix} \\ \begin{matrix} 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{matrix} & \begin{bmatrix} -3\lambda & \lambda & \lambda & 0 & \lambda & 0 & 0 & 0 \\ \mu & -3\lambda - \mu & 0 & \lambda & 0 & 2\lambda & 0 & 0 \\ \mu & 0 & -3\lambda - \mu & 2\lambda & 0 & 0 & \lambda & 0 \\ 0 & \mu & \mu & -3\lambda - 2\mu & 0 & 0 & 0 & 3\lambda \\ \mu & 0 & 0 & 0 & -3\lambda - \mu & \lambda & 2\lambda & 0 \\ 0 & \mu & 0 & 0 & \mu & -3\lambda - 2\mu & 0 & 3\lambda \\ 0 & 0 & \mu & 0 & \mu & 0 & -3\lambda - 2\mu & 3\lambda \\ 0 & 0 & 0 & \mu & 0 & \mu & \mu & -3\mu \end{bmatrix} \end{matrix}$$

Analogue to the simulation section, this method will be the baseline result in our other results.

#### 3.2 Random Left/Right forward with parameter $p$

This matrix is very similar to the one above. But we need to take into account the parameters  $p$  and  $1 - p$  instead of 1 and 0.

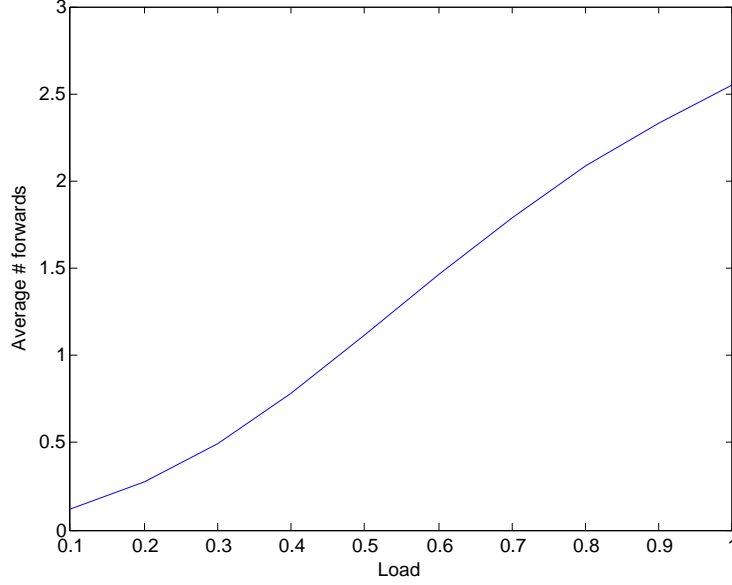


Figure 7: Validation of Forward right

$$Q = \begin{matrix} & \begin{matrix} 000 & 001 & 010 & 011 & 100 & 101 & 110 & 111 \end{matrix} \\ \begin{matrix} 000 \\ 001 \\ 010 \\ 011 \\ 100 \\ 101 \\ 110 \\ 111 \end{matrix} & \begin{bmatrix} -3\lambda & \lambda & \lambda & 0 & \lambda & 0 & 0 & 0 \\ \mu & -3\lambda - \mu & 0 & (2-p)\lambda & 0 & (1+p)\lambda & 0 & 0 \\ \mu & 0 & -3\lambda - \mu & (1+p)\lambda & 0 & 0 & (2-p)\lambda & 0 \\ 0 & \mu & \mu & -3\lambda - 2\mu & 0 & 0 & 0 & 3\lambda \\ \mu & 0 & 0 & 0 & -3\lambda - \mu & (2-p)\lambda & (1+p)\lambda & 0 \\ 0 & \mu & 0 & 0 & \mu & -3\lambda - 2\mu & 0 & 3\lambda \\ 0 & 0 & \mu & 0 & \mu & 0 & -3\lambda - 2\mu & 3\lambda \\ 0 & 0 & 0 & \mu & 0 & \mu & \mu & -3\mu \end{bmatrix} \end{matrix}$$

The lumped matrix (section 3.5) of  $Q$  is equal to the lumped matrix of the example above, i.e. the matrix defines the exact same behaviour. However, for  $N > 6$  the matrices and so the results of the steady state distribution begin to differ.

As in the simulation section, we have validated the results for different values for  $p$ . These results are shown in figure 9.

### 3.3 Random Coprime offset

Modelling this technique yields different results for various ring sizes. The performance of this algorithm is very dependant on the number of coprimes that can be used. This technique yields the same results as Forward Right for ring sizes of up to 4. For  $N = 3$ , the matrix  $Q$  is identical to Random Left/Right forward with parameter 0.5, as the coprimes of 3 are 1 and 2. Which means forwarding a job left or right, both with the same probability.

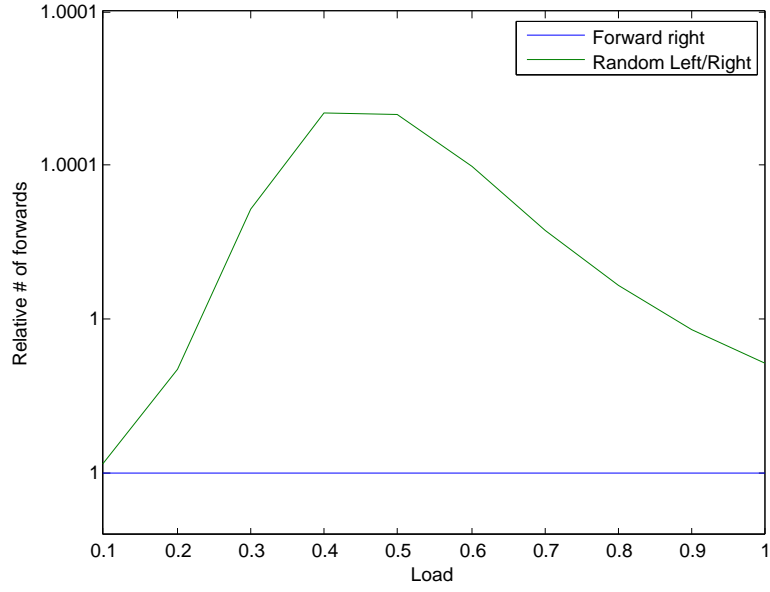


Figure 8: Validation of Random Left/Right with  $p=0.5$

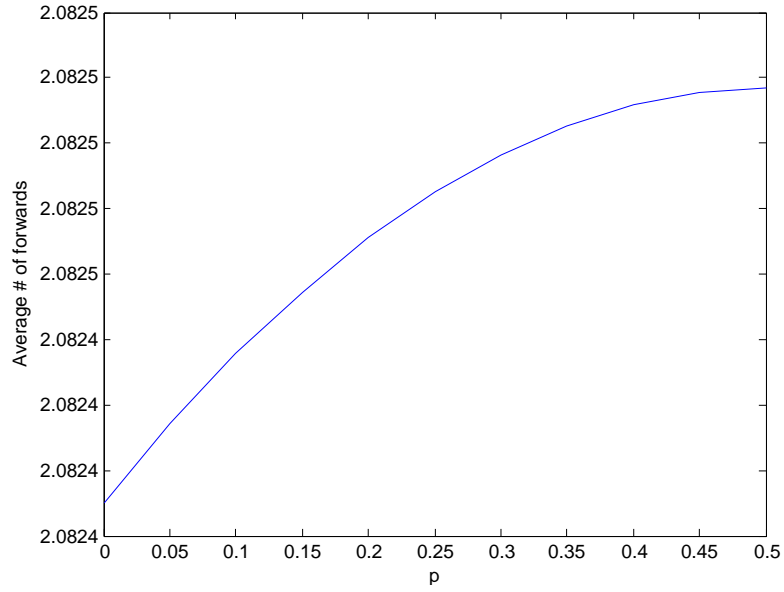


Figure 9: Performance of Random Left/Right with load=0.8

As shown in figure 10, the performance gain of this algorithm is up to 5% for a ring size of  $N = 10$ . The list of coprimes in that scenario is 1, 3, 7, 9, thus 4 possible choices. When increasing the ring size to 11, a prime number, the list of coprimes expands to 1..10 (because 11 is prime), thus 10 possible choices. This increases the relative performance gain up to 8%. To make clear the increased performance is not due to the increase of  $N$ , figure 12 shows the performance

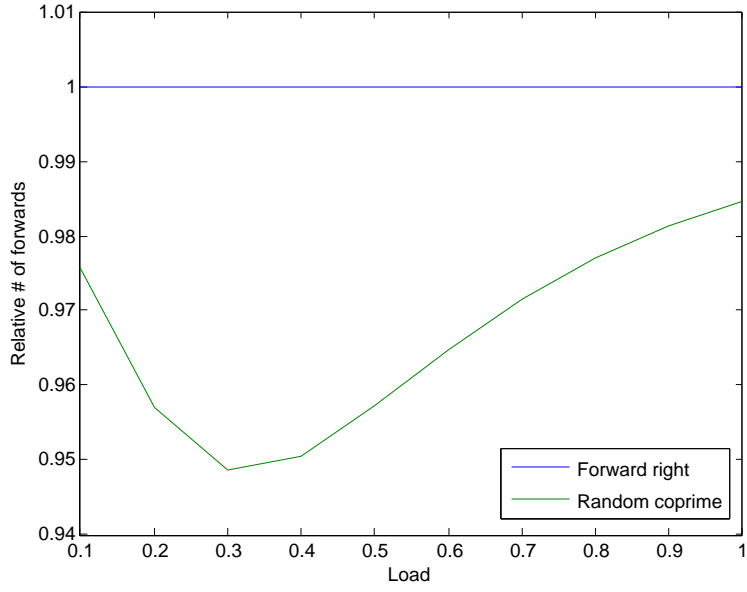


Figure 10: Validation of Coprime algorithm for  $N = 10$

gain for  $N = 12$ .

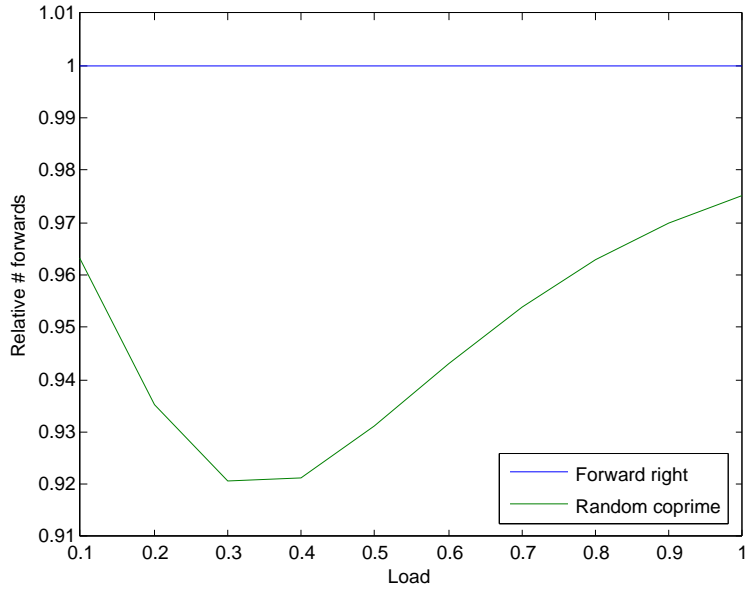


Figure 11: Validation of Coprime algorithm for  $N = 11$

The performance gain is dependant on the number of different paths a job can follow. For  $N = 10$  and  $N = 12$ , a job can follow 4 possible routes, for  $N = 11$ , 10 different routes can be chosen.

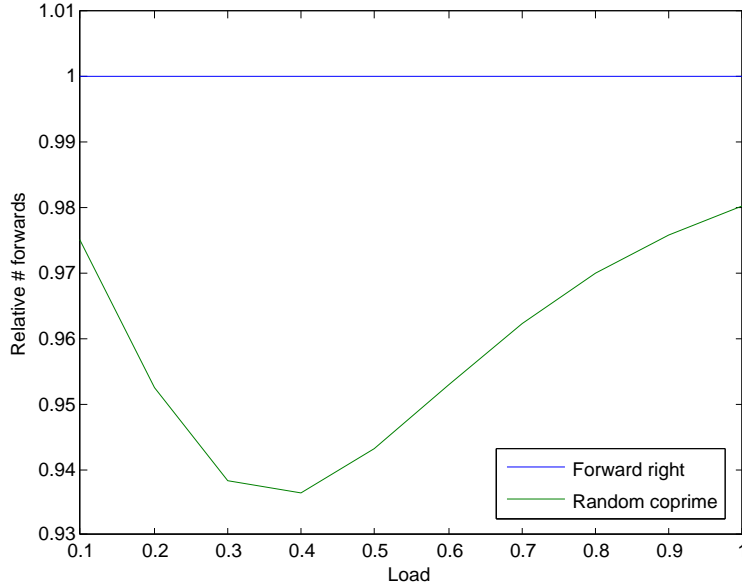


Figure 12: Validation of Coprime algorithm for  $N = 12$

### 3.4 Random Unvisited

This problem can be modelled much more efficiently than the techniques. Since the next node is chosen at random, the information we need to save consists only of the number of servers which are currently busy. This problem is analogue to modelling an Erlang-B loss system. The number of states in this Markov Chain is linear to  $N$ , is much more dense and already represents a lumped Markov Chain. For  $N = 3$ , the matrix is given below.

$$Q = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \end{matrix} & \begin{bmatrix} -3\lambda & 3\lambda & 0 & 0 \\ \mu & -3\lambda - \mu & 3\lambda & 0 \\ 0 & \mu & -3\lambda - \mu & 3\lambda \\ 0 & 0 & \mu & -\mu \end{bmatrix} \end{matrix}$$

### 3.5 Lumped states

Except for Random Unvisited, each discribed technique is modelled into a Markov Chain with  $N^2$  states. However, many of these states are redundant: for example, for  $N = 3$  the states 001, 010 and 100 all represent one of the nodes being busy. For states representing multiple busy nodes, the space between these servers is critical information. Multiple states can be lumped when bitrotating one state can result in another state. Example: the states below are analogue and can therefore be lumped into one state:

$$001101 = 011010 = 110100 = 101001 = 010011 = 100110$$

The example model in 3.1 can be lumped into the following Markov Chain:



$$Q = \begin{array}{c} \begin{array}{cc} & \begin{array}{cccc} 000 & 001 & 011 & 111 \end{array} \\ \begin{array}{c} 000 \\ 001 \\ 011 \\ 111 \end{array} & \begin{bmatrix} -3\lambda & 3\lambda & 0 & 0 \\ \mu & 3\lambda - \mu & 3\lambda & 0 \\ 0 & 2\mu & 3\lambda - 2\mu & 3\lambda \\ 0 & 0 & 3\mu & -3\mu \end{bmatrix} \end{array}$$

Computing the steady state distribution of a Markov Chain is subject to time constraints. Using sparse matrices for our algorithm already solved the memory constraints. Two factors are important when working with matrices: the number of elements and the number of nonzero elements. We will show that both factors are reduced significantly.

For unlumped Markov Chains modelling the Random forward algorithm, a matrix consists of  $2^N$  states, an exponential growth. Lumping these matrices results in a number of states equal to:  $\frac{1}{N} \sum_{d|N} (2^{N/d} \cdot \text{phi}(d))$  with  $\text{phi}(d) = d \cdot \prod_{p|d, p \text{ is prime}} (1 - \frac{1}{p})$  [1]. Although this result greatly reduces the number of states, its complexity is still non-polynomial.

The number of nonzero elements for unlumped Markov Chains is  $(N+1)2^N$ . For lumped matrices, we were not able to derive an exact formula, however, figure 14 shows a clear reduction as well. Yet, this result doesn't seem polynomial either.

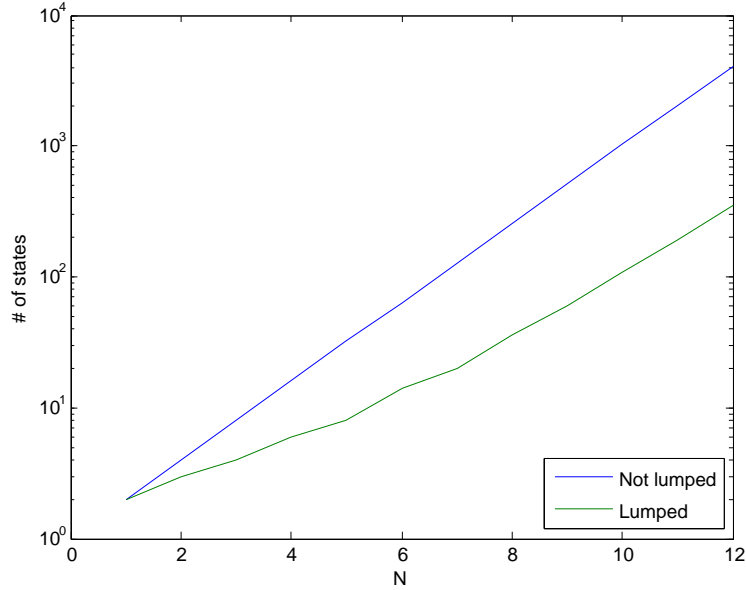


Figure 13: A

It seems lumping is a good technique to push the boundaries of the validation by reducing two important factors of the computation time. However, it is no silver bullet: both the number of states and the number of nonzero elements are nonpolynomial after lumping the matrices.

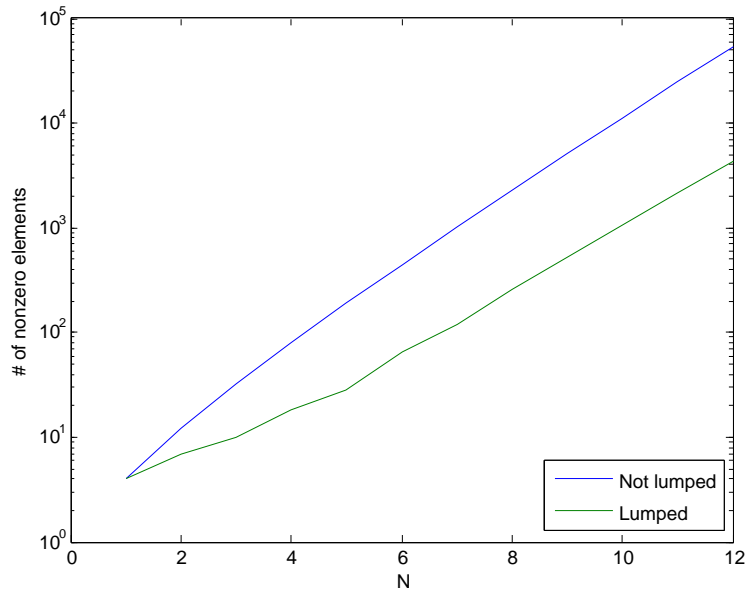


Figure 14: B

### 3.6 Equivalent techniques

Lumping states of a Markov Chain produces an equivalent Markov Chain. This can be used to prove some forwarding techniques are equal up to a certain  $N$ .

## 4 Conclusion

Which techniques work best in which environments? Why? Runner up? Why do some techniques don't work as expected?

## References

- [1] The Online Encyclopedia of Integer Sequences. *A000031*. June 2009. URL: <https://oeis.org/A000031>.
- [2] *The OpenMP® API specification for parallel programming*. URL: <http://openmp.org/wp/>.

## A Simulator source code

```

1 /*
2  * nodes.h
3  *
4  * Created on: Sep 27, 2011
5  * Author: ibensw
6  */
7

```

```

8 #ifndef NODES_H_
9 #define NODES_H_
10
11 #include "ring/node.h"
12 #include "ring/ring.h"
13 #include "ring/job.h"
14 #include <set>
15
16 struct DirectionInfo: public pop::JobInfo{
17     inline DirectionInfo(double length):
18         fLength(length), fDirection(0), fFirst(0)
19     {}
20
21     double fLength;
22     short fDirection;
23     pop::Node* fFirst;
24 };
25
26 struct VisitedInfo: public DirectionInfo{
27     inline VisitedInfo(double length):
28         DirectionInfo(length)
29     {}
30
31     std::set<unsigned int> visited;
32 };
33
34 class RightNode: public pop::Node {
35 public:
36     typedef DirectionInfo info_type;
37
38     RightNode(unsigned int id, pop::Ring* ring, unsigned int size);
39
40     virtual ~RightNode(){}
41
42     bool pushJob(pop::Job* j);
43     void clearJob(pop::Job* j);
44
45     bool wasHereFirst(pop::Job* j);
46     bool accept(pop::Job* j);
47 };
48
49 class SwitchNode: public RightNode {
50 public:
51     typedef DirectionInfo info_type;
52
53     inline SwitchNode(unsigned int id, pop::Ring* ring, unsigned
54         int size):
55         RightNode(id, ring, size), last(1)
56     {}
57
58     bool pushJob(pop::Job* j);
59 protected:
60     int last;
61 };
62
63 class RandSwitchNode: public RightNode {
64 public:
65     typedef DirectionInfo info_type;
66
67     inline static void setValue(double nv){
68         v = nv;

```

```

69     }
70
71     inline RandSwitchNode(unsigned int id, pop::Ring* ring,
72                           unsigned int size):
73         RightNode(id, ring, size)
74     {}
75
76     bool pushJob(pop::Job* j);
77
78 private:
79     static double v;
80 };
81
82 class EvenSwitchNode: public RightNode{
83 public:
84     typedef DirectionInfo info_type;
85
86     inline EvenSwitchNode(unsigned int id, pop::Ring* ring,
87                           unsigned int size):
88         RightNode(id, ring, size)
89     {}
90
91     bool pushJob(pop::Job* j);
92 };
93
94 class PrimeNode: public SwitchNode {
95 public:
96     typedef DirectionInfo info_type;
97
98     static void makePrimes(unsigned int size);
99
100     inline PrimeNode(unsigned int id, pop::Ring* ring, unsigned int
101                      size):
102         SwitchNode(id, ring, size)
103     {
104         if (fPrimes == 0){
105             makePrimes(ring->getSize());
106         }
107     }
108
109     bool pushJob(pop::Job* j);
110
111 protected:
112     static int* fPrimes;
113     static int fPrimesLen;
114 };
115
116 class RandPrimeNode: public PrimeNode{
117 public:
118     typedef DirectionInfo info_type;
119
120     RandPrimeNode(unsigned int id, pop::Ring* ring, unsigned int
121                  size):
122         PrimeNode(id, ring, size)
123     {
124         if (fPrimes == 0){
125             makePrimes(ring->getSize());
126         }
127     }
128
129     bool pushJob(pop::Job* j);
130 };

```

```

127
128 class RandUnvisited: public RightNode{
129 public:
130     typedef VisitedInfo info_type;
131
132     RandUnvisited(unsigned int id, pop::Ring* ring, unsigned int
        size):
133         RightNode(id, ring, size)
134     {}
135
136     bool pushJob(pop::Job* j);
137 };
138
139 class ToTopNode: public RightNode{
140 public:
141     typedef DirectionInfo info_type;
142
143     ToTopNode(unsigned int id, pop::Ring* ring, unsigned int size):
144         RightNode(id, ring, size)
145     {}
146
147     bool pushJob(pop::Job* j);
148 };
149
150 class RRUnvisited: public RightNode{
151 public:
152     typedef VisitedInfo info_type;
153
154     RRUnvisited(unsigned int id, pop::Ring* ring, unsigned int size
        ):
155         RightNode(id, ring, size), offset(0)
156     {}
157
158     bool pushJob(pop::Job* j);
159
160 private:
161     unsigned int offset;
162 };
163
164 #endif /* NODES_H */

```

Listing 2: nodes.h

```

1  /*
2   * servernode.cpp
3   *
4   * Created on: Sep 27, 2011
5   * Author: ibensw
6   */
7
8  #include "servernode.h"
9
10 namespace pop {
11
12     ServerNode::ServerNode(unsigned int id, Ring* ring):
13         Node(id, ring)
14     {}
15
16     ServerNode::~ServerNode() {
17         // TODO Auto-generated destructor stub
18     }
19

```

```
20 } /* namespace pop */
```

Listing 3: servernode.cpp

```
1  /*
2   * configuration.h
3   *
4   * Created on: Oct 6, 2011
5   * Author: ibensw
6   */
7
8  #ifndef CONFIGURATION_H_
9  #define CONFIGURATION_H_
10
11 #include "ring/node.h"
12 #include "ring/job.h"
13
14 void help();
15
16 typedef pop::Node* (*makeNodeType)(unsigned int i, pop::Ring* r,
17     unsigned int size);
18 typedef pop::JobInfo* (*makeInfoType)(double length);
19 struct Configuration{
20     unsigned int seed;
21     double joblength;
22     double arrival;
23     long nodes;
24     unsigned int nodeSize;
25     long progressinterval;
26     long length;
27     long repeat;
28     makeNodeType makeNodeFunction;
29     makeInfoType makeInfoFunction;
30
31     Configuration(int argc, char** argv);
32 };
33 #endif /* CONFIGURATION_H_ */
```

Listing 4: randswitchchain.m

```
1  /*
2   * nodes.cpp
3   *
4   * Created on: Sep 27, 2011
5   * Author: ibensw
6   */
7
8  #include "nodes.h"
9  #include "ring/job.h"
10 #include "ring/ring.h"
11 #include "ring/finishevent.h"
12 #include <stdlib.h>
13 #include <iostream>
14 #include <vector>
15 #include <string.h>
16
17 using namespace pop;
18 using namespace std;
19
20 RightNode::RightNode(unsigned int id, Ring* ring, unsigned int size
21     ):
```

```

21     Node(id, ring, size)
22 {}
23
24 bool RightNode::wasHereFirst(Job* j){
25     info_type* ji = dynamic_cast<info_type*>(j->getInfo());
26
27     if (ji->fFirst == 0){
28         ji->fFirst = this;
29         return false;
30     }
31
32     return (ji->fFirst == this);
33 }
34
35 bool RightNode::accept(Job* j){
36     if (!isBusy()){
37         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
38         fCurrents.insert(j);
39         double len = ji->fLength;
40         fRing->getSimulator()->addEvent(new FinishEvent(fRing->
41             getSimulator()->getTime()+len, j));
42         return true;
43     }
44     return false;
45 }
46
47 bool RightNode::pushJob(Job* j){
48     if (wasHereFirst(j)){
49         return false;
50     }
51
52     if (!accept(j)){
53         j->forward(fRing->getNode(this->fId+1));
54     }
55
56     return true;
57 }
58
59 void RightNode::clearJob(Job* j){
60     fCurrents.erase(j);
61 }
62
63 bool SwitchNode::pushJob(Job* j){
64     if (wasHereFirst(j)){
65         return false;
66     }
67
68     if (!accept(j)){
69         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
70         if (ji->fDirection == 0){
71             ji->fDirection = last;
72             last*=-1;
73         }
74
75         j->forward(fRing->getNode(this->fId + ji->fDirection));
76     }
77
78     return true;
79 }
80
81 double RandSwitchNode::v = 0.5;
82
83 bool RandSwitchNode::pushJob(Job* j){

```

```

82     if (wasHereFirst(j)){
83         return false;
84     }
85
86     if (!accept(j)){
87         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
88         if (ji->fDirection == 0){
89             double rnd = (double)rand() / (double)RAND_MAX;
90             ji->fDirection = (rnd < v ? 1 : -1);
91         }
92
93         j->forward(fRing->getNode(this->fId + ji->fDirection));
94     }
95     return true;
96 }
97
98 bool EvenSwitchNode::pushJob(Job* j){
99     if (wasHereFirst(j)){
100         return false;
101     }
102
103     if (!accept(j)){
104         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
105         if (ji->fDirection == 0){
106             ji->fDirection = ((this->getId() % 2 == 1) ? 1 : -1);
107         }
108
109         j->forward(fRing->getNode(this->fId + ji->fDirection));
110     }
111     return true;
112 }
113
114 int* PrimeNode::fPrimes = 0;
115 int PrimeNode::fPrimesLen = 0;
116
117 unsigned int gcd(unsigned int a, unsigned b){
118     unsigned int t;
119     while(b){
120         t=b;
121         b=a%b;
122         a=t;
123     }
124     return a;
125 }
126
127 void PrimeNode::makePrimes(unsigned int size){
128     vector<unsigned int> primes;
129     for (unsigned int i = 1; i < size; ++i){
130         if (gcd(size, i) == 1){
131             cout << "RelPrime:_" << i << endl;
132             primes.push_back(i);
133         }
134     }
135     fPrimesLen = primes.size();
136     fPrimes = new int[fPrimesLen];
137     memcpy(fPrimes, primes.data(), fPrimesLen * sizeof(unsigned int));
138 }
139
140 bool PrimeNode::pushJob(Job* j){
141     if (wasHereFirst(j)){
142         return false;

```



```

143     }
144
145     if (!accept(j)){
146         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
147         if (ji->fDirection == 0){
148             ji->fDirection = fPrimes[last];
149             ++last;
150             last%=fPrimesLen;
151         }
152
153         j->forward(fRing->getNode(this->fId + ji->fDirection));
154     }
155     return true;
156 }
157
158 bool RandPrimeNode::pushJob(Job* j){
159     if (wasHereFirst(j)){
160         return false;
161     }
162
163     if (!accept(j)){
164         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
165         if (ji->fDirection == 0){
166             ji->fDirection = fPrimes[rand() % fPrimesLen];
167         }
168
169         j->forward(fRing->getNode(this->fId + ji->fDirection));
170     }
171     return true;
172 }
173
174 bool RandUnvisited::pushJob(Job* j){
175     info_type* ji = dynamic_cast<info_type*>(j->getInfo());
176     if (ji->visited.count(this->getId())){
177         return false;
178     }
179
180     if (!accept(j)){
181         ji->visited.insert(this->getId());
182
183         if (fRing->getSize() == ji->visited.size()){
184             return false;
185         }
186
187         unsigned int next;
188         if (5*ji->visited.size() > 4*fRing->getSize()){
189             unsigned int x = rand() % (fRing->getSize() - ji->
190                 visited.size());
191             next = x;
192
193             for (set<unsigned int>::iterator it = ji->visited.begin
194                 (); it != ji->visited.end(); it++){
195                 if (*it <= next){
196                     ++next;
197                 }
198             }
199         } else {
200             do{
201                 next = rand() % fRing->getSize();
202                 } while (ji->visited.count(next));
203         }
204     }

```

```

203         j->forward(fRing->getNode(next));
204     }
205     return true;
206 }
207
208 bool ToTopNode::pushJob(Job* j){
209     if (wasHereFirst(j)){
210         return false;
211     }
212
213     if (!accept(j)){
214         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
215         if (ji->fDirection == 0){
216             if (this->getId() > fRing->getSize()/2){
217                 ji->fDirection = 1;
218             }else{
219                 ji->fDirection = -1;
220             }
221         }
222
223         j->forward(fRing->getNode(this->fId + ji->fDirection));
224     }
225     return true;
226 }
227
228 bool RRunvisited::pushJob(Job* j){
229     info_type* ji = dynamic_cast<info_type*>(j->getInfo());
230     if (ji->visited.count(this->getId())){
231         return false;
232     }
233
234     if (!accept(j)){
235         ji->visited.insert(this->getId());
236
237         if (fRing->getSize() == ji->visited.size()){
238             return false;
239         }
240
241         unsigned int next = this->getId() + offset + 1;
242         ++offset;
243         offset%=(fRing->getSize()-1);
244
245         next%=fRing->getSize();
246         while (ji->visited.count(next)){
247             next++;
248             next%=fRing->getSize();
249         }
250
251         j->forward(fRing->getNode(next));
252     }
253     return true;
254 }

```

Listing 5: randswitchchain.m

```

1 #include <iostream>
2 #include <math.h>
3 #include <time.h>
4 #include <stdlib.h>
5
6 #include "ring/ring.h"
7 #include "ring/job.h"

```

```

8 #include "ring/arriveevent.h"
9 #include "nodes.h"
10 #include "configuration.h"
11 using namespace pop;
12 using namespace std;
13
14 double exp_distr(double lambda){
15     double r = (double)rand() / (double)RAND.MAX;
16     return -lambda * log(r);
17 }
18
19 void preload(Configuration c, Ring* r){
20     double rnd;
21     double l;
22     double load = c.length/c.arrival;
23     for (unsigned int i = 0; i < r->getSize(); ++i){
24         rnd = (double)rand() / (double)RAND.MAX;
25         if (rnd < load){
26             l=exp_distr(c.joblength);
27             r->getSimulator()->addEvent(new ArriveEvent(0.0, new
                Job(c.makeInfoFunction(l)), r->getNode(i)));
28         }
29     }
30 }
31
32 void fillEvents(Configuration c, Ring* r){
33     double t;
34     double l;
35     Node* n;
36     for (unsigned int i = 0; i < r->getSize(); ++i){
37         n=r->getNode(i);
38         t = 0.0;
39         while (t < c.length){
40             t+=exp_distr(c.arrival);
41             l=exp_distr(c.joblength);
42             r->getSimulator()->addEvent(new ArriveEvent(t, new Job(
                c.makeInfoFunction(l)), n));
43         }
44     }
45 }
46
47
48 int main(int argc, char** argv) {
49     Configuration c(argc, argv);
50
51     double success = 0.0;
52     double avghops = 0.0;
53
54     cout.setf(ios::fixed, ios::floatfield);
55     cout.precision(12);
56
57 #pragma omp parallel for
58     for (unsigned int i = 0; i < c.repeat; ++i){
59         Ring r(c.nodes, c.nodeSize, c.makeNodeFunction);
60
61         //preload(c, &r); //disabled to compare output to more
            early results
62
63         fillEvents(c, &r);
64
65         if (c.progressinterval <= 0){
66             r.getSimulator()->run();

```

```

67         }else{
68             r.getSimulator()->run(c.progressinterval);
69         }
70
71 #pragma omp critical
72 {
73     cout << "_____ " << endl;
74     cout << "Run:_" << i << endl;
75     cout << "Total_jobs:\t\t" << r.getTotalJobs() << endl;
76     cout << "Finished_jobs:\t\t" << r.getFinishedJobs() <<
77         endl;
78     cout << "Discarded_jobs:\t\t" << r.getDiscardedJobs()
79         << endl;
80     cout << "Total_hops_(finished):\t" << r.
81         getFinishedJobTotalHops() << endl;
82     long totalhops = r.getFinishedJobTotalHops() + (c.nodes
83         -1) * r.getDiscardedJobs();
84     cout << "Total_hops_(all):\t" << totalhops << endl;
85     cout << "Hops/job_(finished):\t" << (double)r.
86         getFinishedJobTotalHops()/r.getFinishedJobs() <<
87         endl;
88     cout << "Hops/job_(total):\t" << (double)totalhops/r.
89         getTotalJobs() << endl;
90     cout << "Success_ratio:\t\t" << (100.0 * r.
91         getFinishedJobs()) / r.getTotalJobs() << "%" <<
92         endl;
93     success+=(double)(r.getFinishedJobs()) / r.getTotalJobs
94         ();
95     avghops+=(double)r.getFinishedJobTotalHops()/r.
96         getFinishedJobs();
97 }
98
99 if (c.repeat > 1){
100     cout << "_____ " << endl;
101     cout << "Avg._hops/job_(finished):\t" << avghops / c.repeat
102         << endl;
103     cout << "Avg._success_ratio:\t\t" << 100.0 * success / c.
104         repeat << "%" << endl;
105 }
106
107 return 0;
108 }

```

Listing 6: randswitchchain.m

```

1  /*
2   * servernode.h
3   *
4   * Created on: Sep 27, 2011
5   * Author: ibensw
6   */
7
8 #ifndef SERVERNODE_H
9 #define SERVERNODE_H
10
11 #include "ring/node.h"
12 #include "ring/ring.h"
13
14 namespace pop {
15
16 class ServerNode: public Node {

```

```

17 public:
18     ServerNode(unsigned int id, Ring* ring);
19     virtual ~ServerNode();
20 };
21
22 } /* namespace pop */
23 #endif /* SERVERNODE.H */

```

Listing 7: randswitchchain.m

```

1  /*
2   * ring.cpp
3   *
4   *   Created on: Sep 27, 2011
5   *   Author: ibensw
6   */
7
8  #include "ring.h"
9
10 namespace pop {
11
12 Ring::Ring(unsigned int size, unsigned int nodesize, Node* (*mkNode)
13             )(unsigned int i, Ring* r, unsigned int ns):
14     fSize(size),
15     fRing(new Node*[size]),
16     jobsTotal(0), jobsFinished(0), jobsDiscarded(0),
17     jobsFinishedTotalHops(0)
18 {
19     for (unsigned int i = 0; i < size; ++i){
20         fRing[i] = mkNode(i, this, nodesize);
21     }
22 }
23
24 Ring::~~Ring() {
25     for (unsigned int i = 0; i < fSize; ++i){
26         delete fRing[i];
27     }
28     delete[] fRing;
29 }
30
31 } /* namespace pop */

```

Listing 8: randswitchchain.m

```

1  /*
2   * job.h
3   *
4   *   Created on: Sep 27, 2011
5   *   Author: ibensw
6   */
7
8  #ifndef JOB_H_
9  #define JOB_H_
10
11 #include "node.h"
12
13 namespace pop {
14
15 class JobInfo {
16 public:
17     inline JobInfo() {}
18

```

```

19     virtual ~JobInfo() {}
20 };
21
22 class Job {
23 public:
24     Job(JobInfo* ji);
25     virtual ~Job();
26
27     inline Node* getCurrentNode() {
28         return fCurrent;
29     }
30
31     inline JobInfo* getInfo() {
32         return fJobInfo;
33     }
34
35     void forward(Node* n);
36     void finish(double time);
37     void discard();
38
39 private:
40     double fStart;
41     double fFinish;
42     Node* fCurrent;
43     unsigned int fHops;
44     JobInfo* fJobInfo;
45 };
46
47 } /* namespace pop */
48 #endif /* JOB_H_ */

```

Listing 9: randswitchchain.m

```

1  /*
2   * node.h
3   *
4   * Created on: Sep 27, 2011
5   * Author: ibensw
6   */
7
8  #ifndef NODE_H_
9  #define NODE_H_
10
11  #include <set>
12
13  namespace pop {
14  class Ring;
15  class Job;
16
17  class Node {
18  public:
19      Node(unsigned int id, Ring* ring, unsigned int size = 1);
20      virtual ~Node();
21
22      inline unsigned int getId() const {
23          return fId;
24      }
25
26      inline Ring* getRing() const {
27          return fRing;
28      }
29

```

```

30     inline unsigned int getTotalSize() const{
31         return fSize;
32     }
33
34     inline unsigned int getSize() const{
35         return fCurrents.size();
36     }
37
38     inline bool isBusy() const{
39         return fCurrents.size() == fSize;
40     }
41
42     virtual bool pushJob(Job* j) = 0;
43     virtual void clearJob(Job* j) = 0;
44
45 protected:
46     unsigned int fId;
47     Ring* fRing;
48     std::set<Job*> fCurrents;
49     unsigned int fSize;
50 };
51
52 } /* namespace pop */
53 #endif /* NODE_H_ */

```

Listing 10: randswitchchain.m

```

1  /*
2   * ring.h
3   *
4   * Created on: Sep 27, 2011
5   * Author: ibensw
6   */
7
8 #ifndef RING_H_
9 #define RING_H_
10
11 #include "node.h"
12 #include "../simulator/simulator.h"
13
14 // #include <iostream>
15
16 namespace pop {
17
18 class Ring {
19 public:
20     Ring(unsigned int size, unsigned int nodesize, Node* (*mkNode)(
21         unsigned int i, Ring* r, unsigned int ns));
22     virtual ~Ring();
23
24     inline unsigned int getSize(){
25         return fSize;
26     }
27
28     inline Node* getNode(int id){
29         //std::cout << "id: " << id << " = " << (id + fSize) %
30             fSize << std::endl;
31         return fRing[(id + fSize) % fSize];
32     }
33
34     inline Simulator* getSimulator(){
35         return &fSimulator;
36     }
37
38 };
39
40 }

```

```

34     }
35
36     inline unsigned int getTotalJobs(){
37         return jobsTotal;
38     }
39     inline unsigned int getDiscardedJobs(){
40         return jobsDiscarded;
41     }
42     inline unsigned int getFinishedJobs(){
43         return jobsFinished;
44     }
45     inline unsigned int getFinishedJobTotalHops(){
46         return jobsFinishedTotalHops;
47     }
48
49     inline void jobCreated(){
50         ++jobsTotal;
51     }
52
53     inline void jobFinished(unsigned int hops){
54         ++jobsFinished;
55         jobsFinishedTotalHops+=hops;
56     }
57
58     inline void jobDiscarded(){
59         ++jobsDiscarded;
60     }
61
62 private:
63     unsigned int fSize;
64     Node** fRing;
65     Simulator fSimulator;
66
67     unsigned int jobsTotal;
68     unsigned int jobsFinished;
69     unsigned int jobsDiscarded;
70     unsigned int jobsFinishedTotalHops;
71 };
72
73 } /* namespace pop */
74 #endif /* RING_H_ */

```

Listing 11: randswitchchain.m

```

1  /*
2   * node.cpp
3   *
4   * Created on: Sep 27, 2011
5   * Author: ibensw
6   */
7
8  #include "node.h"
9
10 namespace pop {
11
12 Node::Node(unsigned int id, Ring* ring, unsigned int size):
13     fId(id), fRing(ring), fSize(size)
14     {}
15
16 Node::~~Node() {
17     // TODO Auto-generated destructor stub
18 }

```



```

19 }
20 } /* namespace pop */

```

Listing 12: randswitchchain.m

```

1  /*
2  *  events.h
3  *
4  *   Created on: Sep 27, 2011
5  *   Author: ibensw
6  */
7
8  #ifndef EVENTS_H_
9  #define EVENTS_H_
10
11 #include "../simulator/event.h"
12 #include "job.h"
13
14 namespace pop {
15
16 class ArriveEvent : public Event {
17 public:
18     inline ArriveEvent(double scheduled, Job* job, Node* n):
19         Event(scheduled), j(job), first(n)
20     {}
21
22     inline void run(Simulator* simulator){
23         j->forward(first);
24         delete this;
25     }
26
27 private:
28     Job* j;
29     Node* first;
30 };
31
32 } /* namespace pop */
33 #endif /* EVENTS_H_ */

```

Listing 13: randswitchchain.m

```

1  /*
2  *  job.cpp
3  *
4  *   Created on: Sep 27, 2011
5  *   Author: ibensw
6  */
7
8  #include "job.h"
9  #include "ring.h"
10 #include <iostream>
11 using namespace std;
12
13 namespace pop {
14
15 Job::Job(JobInfo* ji):
16     fStart(-1.0), fFinish(-1.0), fCurrent(0), fHops(-1), fJobInfo(
17         ji)
18     {}
19
20 Job::~~Job() {
21     if (fJobInfo){

```

```

21         delete fJobInfo;
22     }
23 }
24
25 void Job::discard(){
26     fCurrent->getRing()->jobDiscarded();
27     //cout << fCurrent->getId() << "\tJob discarded\t(arrival time:
28     " << fStart << " / #hops: " << fHops << ")" << endl;
29     delete this;
30 }
31
32 void Job::finish(double time){
33     fCurrent->getRing()->jobFinished(fHops);
34     fFinish = time;
35     //cout << fCurrent->getId() << "\tJob finished\t(arrival time:
36     " << fStart << " / finish time: " << fFinish << " / #hops:
37     " << fHops << ")" << endl;
38     fCurrent->clearJob(this);
39     delete this;
40 }
41
42 void Job::forward(Node* n){
43     if (!fCurrent){
44         n->getRing()->jobCreated();
45         fStart = n->getRing()->getSimulator()->getTime();
46     }
47     ++fHops;
48     fCurrent = n;
49     if (!n->pushJob(this)){
50         discard();
51     }
52 }
53
54 } /* namespace pop */

```

Listing 14: randswitchchain.m

```

1  /*
2   * finishevent.h
3   *
4   * Created on: Sep 27, 2011
5   * Author: ibensw
6   */
7
8  #ifndef FINISHEVENT_H_
9  #define FINISHEVENT_H_
10
11  #include "../simulator/simulator.h"
12  #include "../simulator/event.h"
13  #include "job.h"
14
15  namespace pop {
16
17  class FinishEvent: public Event {
18  public:
19      inline FinishEvent(double scheduled, Job* job):
20          Event(scheduled), j(job)
21      {}
22
23      inline void run(Simulator* simulator){
24          j->finish(simulator->getTime());
25          delete this;

```

```

26         }
27
28     private:
29         Job* j;
30     };
31
32 } /* namespace pop */
33 #endif /* FINISHEVENT_H */

```

Listing 15: randswitchchain.m

[illegible]

```

45         break;
46     case 's':
47         seed = atol(optarg);
48         break;
49     case 'j':
50         joblength = atof(optarg);
51         break;
52     case 'a':
53         load = atof(optarg);
54         break;
55     case 'n':
56         nodes = atol(optarg);
57         break;
58     case 'c':
59         nodeSize = atol(optarg);
60         break;
61     case 'p':
62         progressinterval = atoi(optarg);
63         break;
64     case 'l':
65         length = atol(optarg);
66         break;
67     case 't':
68         repeat = atol(optarg);
69         break;
70     case 'h':
71         help();
72         exit(0);
73         break;
74     case 'v':
75         RandSwitchNode::setValue(atof(optarg));
76         break;
77     default:
78         cout << "Unknown_option:_" << optopt << endl;
79         break;
80     }
81 }
82
83 arrival = joblength/load/nodeSize;
84
85 srand(seed);
86 cout << "Seed:_" << seed << endl
87      << "Interarrival_time:_" << arrival << endl;
88
89 for (index = optind; index < argc; index++){
90     string arg = argv[index];
91     if (arg == "right"){
92         makeNodeFunction = createN<RightNode>;
93         makeInfoFunction = createJI<RightNode::info_type>;
94     }
95     if (arg == "switch"){
96         makeNodeFunction = createN<SwitchNode>;
97         makeInfoFunction = createJI<SwitchNode::info_type>;
98     }
99     if (arg == "randswitch"){
100         makeNodeFunction = createN<RandSwitchNode>;
101         makeInfoFunction = createJI<RandSwitchNode::info_type>;
102     }
103     if (arg == "evenswitch"){
104         makeNodeFunction = createN<EvenSwitchNode>;
105         makeInfoFunction = createJI<EvenSwitchNode::info_type>;
106     }

```

```

107         if (arg == "prime"){
108             makeNodeFunction = createN<PrimeNode>;
109             makeInfoFunction = createJI<PrimeNode::info_type>;
110         }
111         if (arg == "randprime"){
112             makeNodeFunction = createN<RandPrimeNode>;
113             makeInfoFunction = createJI<RandPrimeNode::info_type>;
114         }
115         if (arg == "randunvisited"){
116             makeNodeFunction = createN<RandUnvisited>;
117             makeInfoFunction = createJI<RandUnvisited::info_type>;
118         }
119         if (arg == "totop"){
120             makeNodeFunction = createN<ToTopNode>;
121             makeInfoFunction = createJI<ToTopNode::info_type>;
122         }
123         if (arg == "rrunvisited"){
124             makeNodeFunction = createN<RRUnvisited>;
125             makeInfoFunction = createJI<RRUnvisited::info_type>;
126         }
127     }
128
129     if (!makeNodeFunction){
130         cerr << "No_type_given" << endl;
131         exit(1);
132     }
133 }

```

Listing 16: randswitchchain.m

```

1  /*
2   * schedule.h
3   *
4   * Created on: Sep 26, 2011
5   * Author: ibensw
6   */
7
8  #ifndef SCHEDULE_H_
9  #define SCHEDULE_H_
10
11  #include "event.h"
12
13  namespace pop {
14
15  class Schedule {
16  public:
17      inline Schedule(Event* e):
18          fE(e)
19      {}
20
21      inline bool operator<(const Schedule& s) const{
22          return fE->getScheduleTime() > s.fE->getScheduleTime();
23      }
24
25      inline Event* getEvent(){
26          return fE;
27      }
28
29  private:
30      Event* fE;
31  };
32

```

```

33 } /* namespace pop */
34 #endif /* SCHEDULE_H_ */

```

Listing 17: randswitchchain.m

```

1  /*
2   * event.h
3   *
4   * Created on: Sep 26, 2011
5   * Author: ibensw
6   */
7
8  #ifndef EVENT_H_
9  #define EVENT_H_
10
11  // #include "simulator.h"
12
13  namespace pop {
14  class Simulator;
15
16  class Event {
17  public:
18      inline Event(double scheduled):
19          fScheduled(scheduled)
20          {}
21
22      inline virtual ~Event() {
23
24      }
25
26      inline double getScheduleTime() {
27          return fScheduled;
28      }
29
30      virtual void run(Simulator* simulator) = 0;
31
32  protected:
33      double fScheduled;
34  };
35
36 } /* namespace pop */
37 #endif /* EVENT_H_ */

```

Listing 18: randswitchchain.m

```

1  /*
2   * simulator.h
3   *
4   * Created on: Sep 26, 2011
5   * Author: ibensw
6   */
7
8  #ifndef SIMULATOR_H_
9  #define SIMULATOR_H_
10
11  #include <queue>
12  #include "event.h"
13  #include "schedule.h"
14
15  namespace pop {
16
17  class Simulator {

```

```

18 public:
19     Simulator();
20     virtual ~Simulator();
21
22     void run();
23     void run(int infointerval);
24
25     inline double getTime(){
26         return fNow;
27     }
28
29     inline unsigned int getPendingEvents(){
30         return fPending.size();
31     }
32
33     void addEvent(Event* e);
34
35 private:
36     std::priority_queue<Schedule> fPending;
37     double fNow;
38 };
39
40 } /* namespace pop */
41 #endif /* SIMULATOR_H_ */

```

Listing 19: randswitchchain.m

```

1  /*
2   * simulator.cpp
3   *
4   * Created on: Sep 26, 2011
5   * Author: ibensw
6   */
7
8  #include "simulator.h"
9  #include <iostream>
10 using namespace std;
11
12 namespace pop {
13
14 Simulator::Simulator():
15     fNow(0.0)
16 {}
17
18 Simulator::~~Simulator() {}
19
20
21 void Simulator::run(){
22     while(!fPending.empty()){
23         Schedule x = fPending.top();
24         fPending.pop();
25         fNow = x.getEvent()->getScheduleTime();
26         x.getEvent()->run(this);
27     }
28 }
29
30 void Simulator::run(int interval){
31     int next = interval;
32     while(!fPending.empty()){
33         Schedule x = fPending.top();
34         fPending.pop();
35         fNow = x.getEvent()->getScheduleTime();

```

```

36         if (fNow > next){
37             cout << "Time:_" << fNow << "\tPending_events:_" <<
                 fPending.size() << endl;
38             next+=interval;
39         }
40         x.getEvent()->run(this);
41     }
42 }
43
44 void Simulator::addEvent(Event* e){
45     fPending.push(Schedule(e));
46 }
47
48 }

```

Listing 20: randswitchchain.m

```

1  /*
2   * schedule.cpp
3   *
4   * Created on: Sep 26, 2011
5   * Author: ibensw
6   */
7
8  #include "schedule.h"
9
10 namespace pop {
11
12
13 } /* namespace pop */

```

Listing 21: randswitchchain.m

## B MATLAB Numerical evaluation code

```

1 function [Q] = rightchain(size, rate)
2 %RIGHTCHAIN Generate a Markov Chain that always forwards right
3 %Parameters:
4 %     size    The size of the ring
5 %     rate    The rate of arrivals
6
7     totalsize = 2^size;
8     Q = sparse(totalsize, totalsize);
9
10    BITS = zeros(1, size);
11
12    for i=1:size
13        BITS(i) = 2^(i-1);
14    end
15
16    for i=0:(totalsize-1)
17        t=0;
18        for b=1:size
19            j=bitxor(i, BITS(b));
20            if bitand(i, BITS(b))
21                Q(i+1, j+1)=1;
22            else
23                r=rate;
24                bt=b+1;

```



```

25         while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (
26             b))
27             bt=bt+1;
28             r = r + rate;
29         end
30         Q(i+1, j+1)=r;
31     end
32     t=t + Q(i+1, j+1);
33 end
34 Q(i+1, i+1) = -t;
35 end
36 end

```

Listing 22: rightchain.m

```

1 function [Q] = randswitchchain(size, rate, p)
2 %RANDSWITCHCHAIN Generates a Markov Chain that randomly forward
3   left or right
4 %Parameters:
5 %   size      The size of the Markov Chain
6 %   rate      The rate of arrivals
7 %   p          The probability a job is forwarded right
8 %              (Default: 0.5)
9
10     if nargin < 3
11         p=0.5;
12     end
13
14     totalsize = 2^size;
15     Q = sparse(totalsize, totalsize);
16
17     BITS = zeros(1, size);
18
19     for i=1:size
20         BITS(i) = 2^(i-1);
21     end
22
23     for i=0:(totalsize-1)
24         t=0;
25         for b=1:size
26             j=bitxor(i, BITS(b));
27             if bitand(i, BITS(b))
28                 Q(i+1, j+1)=1;
29             else
30                 r=rate;
31                 bt=b+1;
32                 while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (
33                     b))
34                     bt=bt+1;
35                     r = r + rate*p;
36                 end
37                 bt=b-1;
38                 while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (
39                     b))
40                     bt=bt-1;
41                     r = r + rate*(1-p);
42                 end
43                 Q(i+1, j+1)=r;
44             end
45         end
46         t=t + Q(i+1, j+1);
47     end
48 end

```

```

44     Q(i+1, i+1) = -t;
45     end
46
47 end

```

Listing 23: randswitchchain.m

```

1 function [ Q ] = rprimechain( size , rate )
2 %RPRIMECHAIN Generate a Markov Chain that chooses a random coprime
   and uses this as forwarding offset
3 %Parameters:
4 %     size      The size of the ring
5 %     rate      The arrival rate
6
7     totalsize=2^size;
8     rprimes=[];
9
10    for i=1:(size-1)
11        if gcd(size, i) == 1
12            rprimes=[rprimes i];
13        end
14    end
15
16    rpcount = length(rprimes);
17
18    %Q=zeros(totalsize);
19    Q=sparse(totalsize, totalsize);
20
21    for i=0:totalsize-1
22        tot=0;
23        for j=0:size-1
24            k=2^j;
25            if bitand(i,k)
26                Q(i+1, i-k+1) = 1.0;
27                tot=tot+1.0;
28            else
29                c=0;
30                for p=rprimes
31                    current=mod(j-p, size);
32                    while (bitand(i,2^current))
33                        current=mod(current
34                            -p, size);
35                        c=c+1;
36                    end
37                    Q(i+1, i+k+1) = rate + c*rate/
38                        rpcount;
39                    tot=tot+Q(i+1, i+k+1);
40                end
41            end
42            Q(i+1, i+1) = -tot;
43        end
44    end

```

Listing 24: rprimechain.m

```

1 function [ Q ] = runvisitedchain( size , rate )
2 %RUNVISITEDCHAIN Generate a Markov Chain that forwards to an
   unvisited node
3 %Parameters:
4 %     size      The size of the ring

```

```

5 %      rate      The arrival rate
6
7      rate = rate*size;
8
9      Q = sparse(size+1, size+1);
10
11      Q(1,2) = rate;
12      Q(1,1) = -rate;
13
14      Q(size+1, size) = size;
15      Q(size+1, size+1) = -size;
16
17      for i=2:size
18          Q(i,i-1) = i-1;
19          Q(i,i+1) = rate;
20          Q(i,i) = -Q(i,i+1)-Q(i,i-1);
21      end
22
23 end

```

Listing 25: runvisitedchain.m

```

1 function [ avg ] = avghops(Q, d)
2 %AVGHOPS Calculate average number of times a job is forwarded
3 %Parameters:
4 %      Q      The matrix representing a markov chain
5 %Optional:
6 %      d      Debug mode, default=1, disable debug output=0
7
8      if nargin < 2
9          d=1;
10      end
11
12      steady=full(ctmcsteadystate(Q));
13
14      len=length(Q);
15      states=log2(len);
16      avg=0;
17      total=0;
18
19      for i=0:(states-1)
20          c=0;
21          prefix=((2^i)-1) * 2^(states-i);
22          for j=0:(2^(states-i-1))-1
23              c=c+steady(prefix + j + 1);
24          end
25          total=total+c;
26          if d
27              fprintf('%d_hops:\t%f\n', i, c);
28          end
29          avg=avg+(c*i);
30      end
31
32      loss=steady(len);
33      avg=avg/(1-loss);
34      if d
35          fprintf('Loss:\t%f\nTotal:\t%f\nAverage_#hops:\t%f\n', loss, total + loss, avg);
36      end
37 end

```

Listing 26: avghops.m

```

1 function [ avg ] = ruavghops( Q, d )
2 %RUAVGHOPS Calculate average number of times a job is forwarded for
   the random unvisited chain
3 %Parameters:
4 %     Q       The matrix representing a markov chain using the
   random unvisited forwarding algorithm
5 %Optional:
6 %     d       Debug mode, default=1, disable debug output=0
7
8     if nargin < 2
9         d=1;
10    end
11
12    steady=ctmcsteadystate(Q);
13
14    len=length(Q);
15
16    avg = 0;
17    avgp = zeros(1, len);
18
19    for i=0:len-2
20        tmpavg = 0;
21        for h=0:i
22            c = prod(i-h+1:i) * (len-1-i) / prod(len-1-
                h:len-1);
23            tmpavg = tmpavg + (c * h);
24            avgp(h+1) = avgp(h+1) + (c*steady(i+1));
25        end
26        avg=avg + steady(i+1) * tmpavg;
27    end
28
29    avgp(len) = steady(len);
30
31    loss=steady(len);
32    avg=avg/(1-loss);
33    if d
34        avgp
35        fprintf('Loss:\t%f\nAverage-#hops:\t%f\n', loss ,
                avg);
36    end
37
38 end

```

Listing 27: ruavghops.m

```

1 function [ pi ] = ctmcsteadystate( Q )
2 %CTMCSTEADYSTATE Steady state distribution of a continious time
   markov chain
3 %Parameters:
4 %     Q       Matrix representing a Markov Chain
5 %Source: http://speed.cis.nctu.edu.tw/~ydlin/course/cn/nsd2009/
   Markov-chain.pdf (slide 10)
6
7     T=Q;
8     len=length(Q);
9     T(:,len)=ones(len, 1);
10    e=zeros(1, len);
11    e(len)=1;
12    pi=e*inv(T);
13 end

```

Listing 28: ctmcsteadystate.m

```

1 function [ avg ] = lumpavghops(Q)
2 %LUMPAVGHOPS Get the average number of times a job is forwarded
   when the state matrix is lumped
3 %Parameters:
4 %      Q      A lumped matrix representation of a markov Chain
5
6     fullsize=length(Q);
7     [Q S]=lump(Q);
8     lumpsize=length(S);
9     nodes=log2( fullsize );
10    hops=zeros(1,nodes+1);
11
12    steady=ctmcsteadystate(Q);
13
14    hops(1)=steady(1); %zero hops
15    hops(nodes+1)=steady(lumpsize); %loss
16    for i=2:lumpsize-1;
17        bits=ceil(log2(S(i)+1));
18        hops(1)=hops(1)+(nodes-bits)/nodes*steady(i);
19
20        for j=bits:-1:1
21            c=0;
22            while c<j && bitand(S(i), 2^(j-c-1))
23                c=c+1;
24            end
25            hops(c+1)=hops(c+1) + steady(i)/nodes;
26        end
27    end
28
29    %fprintf('Sum:\t%f\n',sum(hops));
30
31    avg=(hops(1:nodes)*[0:nodes-1]')/(1-steady(lumpsize));
32
33 end

```

Listing 29: lumpavghops.m

```

1 function [Ql S] = lump(Q)
2 %LUMP Lump a matrix representing a Markov Chain
3 %Parameters:
4 %      Q      The matrix that should be lumped
5 %Return:
6 %      Ql     The lumped matrix representation
7 %      S      The states that are used in the lumped matrix
8 %The states of the matrix Q must represent the availability of the
   the servers
9
10    [S R C] = makestates(log2(length(Q)));
11
12    Ql=sparse(length(S), length(S));
13
14    [i j s] = find(Q);
15
16    for x=1:length(i)
17        Ql(R(i(x)),R(j(x)))=Ql(R(i(x)),R(j(x)))+s(x);
18    end
19
20    for x=1:length(S)
21        Ql(x,:)=Ql(x,:)/C(x);
22    end
23
24 end

```

Listing 30: lump.m

```

1 function [r, reindex, coverage] = makestates(rsize)
2 %Generate lumped states
3 %Parameters:
4 %     rsize           Size of the ring (or log2 of the number of
      states of the matrix)
5 %Return:
6 %     r               Vector of the remaining states, ordered
7 %     reindex         Reference index, each old state points to
      the new lumped state
8 %     coverage        How many states the lumped state with the
      same index represents
9
10     powers = 2.^[0:rsize-1];
11
12     function [v] = rotate(a, size)
13         p=2^(size-1);
14         v = a*2 + floor(a/p) - 2*p*floor(a/p);
15     end
16
17     function [r] = makesmallest(a)
18         r=a;
19         for i=1:(rsize-1)
20             a=rotate(a, rsize);
21             if a<r
22                 r=a;
23             end
24         end
25     end
26
27     reindex = [];
28     for i=0:(2^rsize)-1
29         reindex = [reindex makesmallest(i)];
30     end
31
32     function [c] = cover(a, size)
33         c=1;
34         a=makesmallest(a);
35         b=rotate(a, size);
36         while a ~= b
37             b=rotate(b, size);
38             c=c+1;
39         end
40     end
41
42     function [r] = smallest(a, size)
43         r=a;
44         for i=1:(size-1)
45             a=rotate(a, size);
46             if a<r
47                 r=a;
48             end
49         end
50     end
51
52     function [v] = f(word, bits, place, size)
53         if place > size
54             v = word;
55         elseif bits == 0
56             v = f(word, bits, place+1, size);

```

```

57         elseif place + bits > size
58             v = f(word + powers(place), bits-1, place
59                 +1, size);
60         else
61             v = [f(word + powers(place), bits-1, place
62                 +1, size) f(word, bits, place+1, size)
63                 ];
64         end
65     end
66
67     function [r] = makecombs(k, n)
68         leadzeros = ceil(n/k)-1;
69         fullsize = n - leadzeros - 1;
70         r = f(0,k-1,1,fullsize)*2 + 1;
71     end
72
73     r=[0 2^(rsize)-1];
74     for i=1:rsize-1
75         r = [r makecombs(i, rsize)];
76     end
77
78     s=[];
79     for i=r
80         s=[s reindex(i+1)];
81     end
82
83     r=unique(s);
84     reindex = arrayfun(@(x) find(r == x), reindex);
85
86     coverage=[];
87     for w=r
88         coverage=[coverage cover(w, rsize)];
89     end
90 end

```

Listing 31: makestates.m