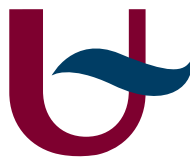


Masterthesis:
The power of two paths in grid computing
networks

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 focuses on the algorithms for choosing a successor node for a job. ...

Acknowledgments

This thesis is not just the work of one person, I could never accomplish this without the people around me. I would like to take this opportunity to thank some of them in particular.

First and foremost, my supervisor Professor dr. Benny Van Houdt. First of all, 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 got stuck. I was always welcome to drop by and reflect thoughts.

I would also like to thank my family, especially my parents. It is because of their financial support and trust I could complete my education. They gave me the freedom of making my own choices and motivated me when I needed it. My sisters, Anneleen and Evelien have been a great help and source of moral support these five years. Anneleen deserves a special note for proofreading this thesis and other assignments I had to write in English.

Next, my friends should be thanked. Playing games, eating in group and going out together is what made the past five years without a 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 choices when easy ones were available. She is a girl who understands me and never fails to cheer me up.

Contents

| | | |
|----------|---|-----------|
| 1 | Setup | 5 |
| 1.1 | Forwarding algorithms | 6 |
| 1.2 | Forward to neighbor | 6 |
| 1.3 | Forward anywhere | 7 |
| 1.4 | Properties of Algorithms | 8 |
| 2 | Simulation | 9 |
| 2.1 | Measure | 9 |
| 2.2 | Results | 10 |
| 2.3 | Multiple execution units | 14 |
| 3 | Numerical Validation | 16 |
| 3.1 | Comparison | 17 |
| 3.2 | Lumped states | 22 |
| 3.3 | Equivalent algorithms | 23 |
| 4 | Conclusion | 26 |
| A | Simulator source code | 29 |
| B | MATLAB Numerical evaluation code | 47 |

Introduction

This thesis researches the behavior of different 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. We specify how each algorithm works and their properties are summed up.

Next, section 2 provides a short introduction about the simulator we wrote and how to use it. Further, it contains the results of the simulations we have performed. Curiosities encountered in the results are simulated using other parameters. The section finishes with the attempt to transform the results of single CPU servers into the results of multiple CPU servers.

The following part, section 3, is used to validate the results of the simulation using numerical algorithms. Some of the algorithms are modeled into Markov Chains to get numerically exact results. Afterwards, the structure of the Markov Chains are reviewed and optimized. At the end of this section we will try to find equivalencies between the algorithms under certain circumstances.

Finally, in section 4, we will give our thoughts on the results obtained in this thesis.

1 Setup

We are using a ring-structured distributed system of N nodes. Each node is connected to two neighbors, its 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 the incoming job to another node. When a job has visited all nodes and none of them was found idle, the job is dropped.

Jobs have an arrival time, a length, the ID of the first node and optional metadata. They arrive at each node independently as a Poisson process at rate λ . Their length is exponentially distributed with mean μ (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, some algorithms require the node to keep state information. The performance of the algorithms is the main focus of this thesis. Different techniques will be discussed and simulated. Afterwards, some results of the simulation will be validated using numerical techniques. Note that the cost of forwarding a job is ignored. 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 the forwarding algorithms is measured by the average number of nodes 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.

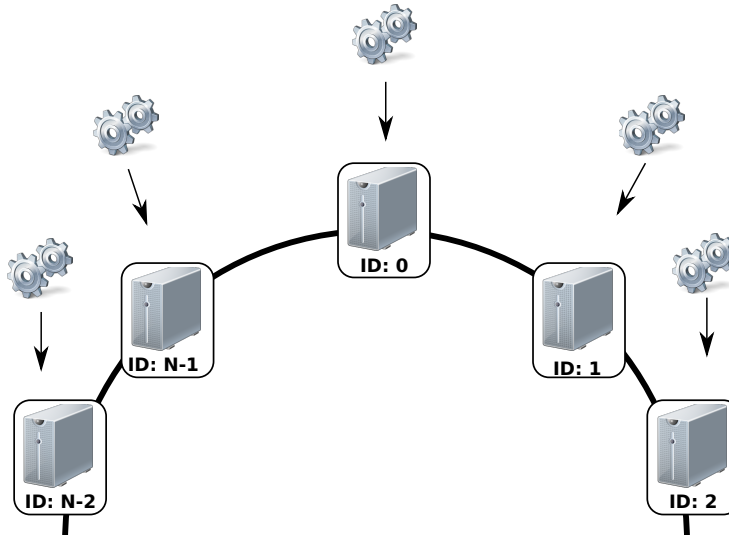


Figure 1: A ring structured network

1.1 Forwarding algorithms

Nodes that must forward a job will choose another node of the ring. Nodes have no information about other nodes, so they have no idea which nodes are idle or busy. The algorithms are grouped into two categories: forward to neighbor and forward anywhere. The first group allows a busy node to forward an incoming job to either its left or its right neighbor, where the latter may forward these jobs to any node in the ring. Because the amount of dropped jobs is equal for each forwarding algorithm, these jobs will be ignored when computing the average number of forwards. 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 neighbor

Forward right

A busy node using this technique will forward a job to its right neighbor. The job will keep traveling in this direction until an idle node is found, where it will be processed, or until it has visited all nodes when it will be dropped. Unless otherwise noted, this algorithm is used as a baseline in all further tests. This algorithm does not use any space in the job's metadata, nor must nodes save states.

Left/right forward

A variant to the previous algorithm is the *left/right forward* technique. Instead of forwarding each job to its right neighbor, a busy node will alternate the direction after forwarding an incoming job. To avoid a job coming back, this initial direction is saved in the job's metadata. Busy nodes receiving a job from another neighbor must forward it in the same direction as specified in the job's metadata. This algorithm requires the job to keep 1 bit representing the

direction in the job's metadata, it also needs to save 1 bit state information in each node to keep track of the last direction a new job was forwarded to.

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 $1 - p$. Like the previous technique, the direction is saved in the job's metadata and subsequent nodes must maintain this direction when forwarding. Unlike *left/right forward*, nodes do not need to save state information.

Position-dependent forwarding

As shown in figure 1, each node in the ring has a unique ID. The IDs are ordered clockwise on the ring. Nodes using this algorithm will always forward an incoming job into the same direction: to the right when the node's id is even, to the left otherwise. Just like the previous algorithm, the direction is saved in the job's metadata and this direction must be used when other nodes have to forward the job. The initial direction of incoming jobs can be derived from the node's ID, therefore the node needs no state information.

1.3 Forward anywhere

The ring structure can be used in real networks, however in many cases the ring is just a virtual overlay over another structure (e.g. the internet). In these networks each node is able to connect each other node directly and more sophisticated forwarding algorithms can be used.

Random unvisited

The Random unvisited algorithm is the most basic algorithm in this category. Every time a job must be forwarded, a list of unvisited nodes is generated and a random node is chosen from that list. The current node is added to the list of visited nodes, which is found in the job's metadata. This list can be saved using $N - 1$ bits (-1 because the initial node is saved separately). Each unvisited node must have an equal probability of being picked.

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 forwarding offset and it is saved in the job's metadata. When a job is forwarded, it is sent exactly this many nodes 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: $[1, 3, 7, 9]$. Let's assume a job arrives at node 3 and the last time node 3 forwarded a job, the job 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 all 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.

Because the list of coprimes can be regenerated each time a job arrives, it must not be saved. Both the job and the node must keep an index to a coprime. The size of this list $\varphi(N)$, ergo an index to an element in this list requires $\lceil \log_2 \varphi(N) \rceil$ bits (see section 1.4 for the definition of $\varphi(N)$).

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. As Coprime offset, the offset must be saved in the job's metadata. Since the offset for incoming jobs is chosen at random, the nodes do not need to keep state information.

1.4 Properties of Algorithms

We have reviewed some of the basic properties for each algorithm. They are summarized in table 1. The function $\varphi(N)$ is Euler's totient function and counts the coprimes of N up to N [1].

$$\begin{aligned}\varphi(N) &= \sum_{0 < p < N, p \perp N} 1 \\ &= N \cdot \prod_{p|N, p \text{ is prime}} \left(1 - \frac{1}{p}\right)\end{aligned}$$

Note that this value is upper bound by $N - 1$ and equals $N - 1$ when N is prime.

| | Size known | First forward | Possible paths | Space requirement (job) | Space requirement (node) |
|-------------------------------|------------|---------------|------------------|-----------------------------------|-----------------------------------|
| Forward right | N | 1 | 1 | 0 | 0 |
| Left/right forward | N | 2 | 2 | 1 | 1 |
| Random left/right forward (p) | N | 2 | 2 ⁽¹⁾ | 1 | 0 |
| Position dependent forward | N | 1 | 1 | 1 | 0 |
| Random unvisited | Y | $N - 1$ | $(N - 1)!$ | $N - 1$ | 0 |
| Coprime offset | Y | $\varphi(N)$ | $\varphi(N)$ | $\lceil \log_2 \varphi(N) \rceil$ | $\lceil \log_2 \varphi(N) \rceil$ |
| Random coprime offset | Y | $\varphi(N)$ | $\varphi(N)$ | $\lceil \log_2 \varphi(N) \rceil$ | 0 |

Table 1: Properties of forwarding algorithms

Size known Represents whether the size of the ring must be known to the nodes in order for the algorithm to function. This implies that a change in the ring (node joins or leaves) must be propagated over the ring.

¹For Random Left/Right forward where $p \neq 0.5$, the probability of each path is different.

First forward When the node on which the incoming job arrives is busy, this number represents the number of candidates to which the node can forward the job.

Possible paths Assuming the system is saturated, this number represents the number of possible paths a job can travel until being dropped. The probability of each path is the same (unless otherwise noted).

Space requirement (job) This value represents the number of bits required in the job's metadata.

Space requirement (node) This value represents the number of bits required to save the node's state information.

2 Simulation

To evaluate the different algorithms discussed in the previous section, 2 methods will be used. Our first method is using a simulator. The second method is the numerical evaluation of the simulation results 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 [3]. The source code of the simulator can be found in appendix A or at <http://code.google.com/p/powerofpaths/>.

The simulator can be controlled using a command line interface of which the 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 that the number of nodes a job must visit should be low. As a measure for our experiments, we will be using the average number of times a job is forwarded before it is executed. Since the number of forwards of a job that could not be executed is the same for each algorithm (i.e. $N - 1$: traversing each link but one), and the loss rate of each algorithm is the same (because

forwards are instantaneous), we will not take these dropped 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 ∞ , 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. For the purpose of this thesis, only loads up to 1 are discussed.

We will compare each algorithm to a baseline result. The baseline used in this thesis is the *forward right* algorithm. This means 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.

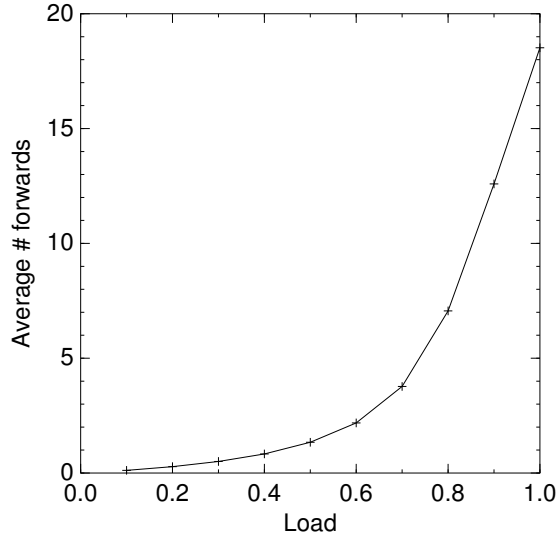


Figure 2: The forward right baseline result

2.2 Results

Left/right forward

It is intuitively clear that alternating the forwarding direction of incoming jobs should distribute the load better than keeping the same direction, certainly under temporary local heavy load. 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.

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.

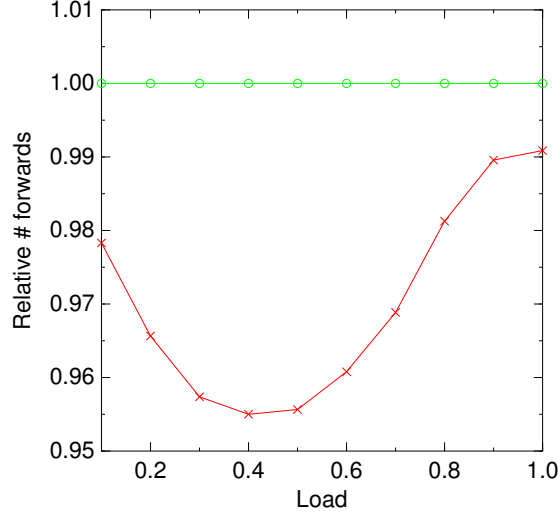


Figure 3: Left/right forward

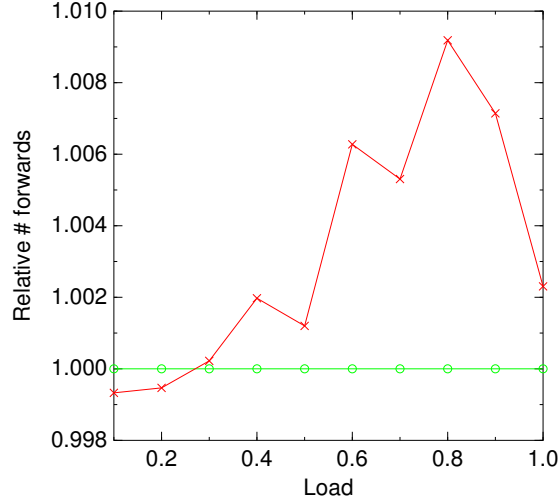


Figure 4: Random Left/Right forward with parameter 0.5

Figure 4 shows the results of this algorithm for $p = 0.5$. For $p = 0$, the algorithm is equal to the *forward right* algorithm. Hence, we should investigate how different values of p influence the final results. Figure 5 shows the results obtained for $0 \leq p \leq 0.5$. We find no value of p for which this algorithm performs better than *forward right*.

Position-dependant forwarding

This technique groups every 2 nodes into small virtual clusters. When a job arrives at a busy node, the job will be forwarded to the other node in this cluster. Jobs leaving a cluster will seem to do this in a random direction ($p = 0.5$). Since

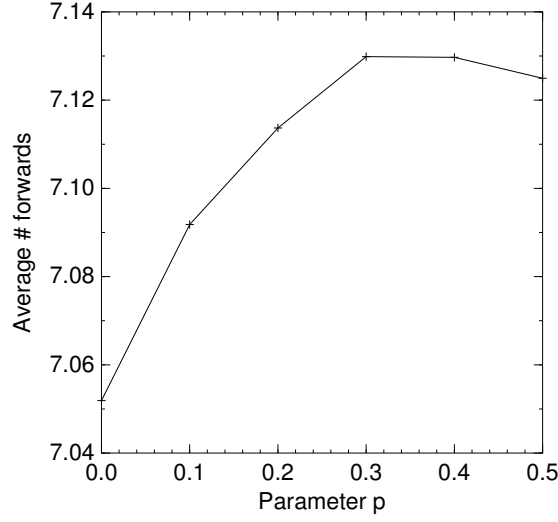


Figure 5: Random Left/Right forward with load 0.8

the load is concentrated per cluster instead of being distributed over the whole system, this technique performs worse than other techniques. The results are shown in figure 6.

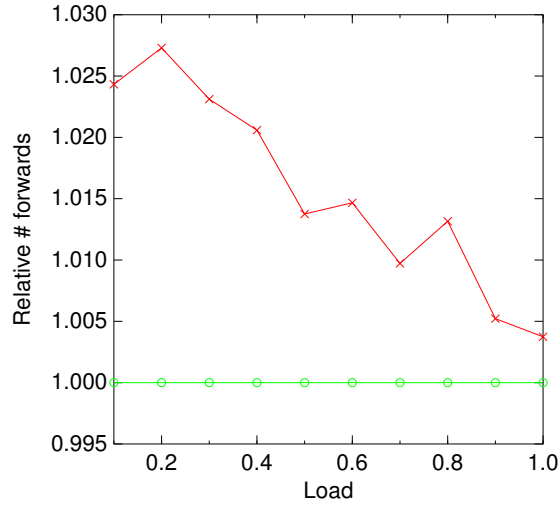


Figure 6: Position-dependant forwarding

Random unvisited

Unlike the previous algorithms, this algorithm is not bound to its neighbors when forwarding a job. Lifting this constraint allows a serious performance boost.

Figure 7 shows the results of this algorithm. The performance benefit is at least 10% in the range 0.20 – 1.00, and even up to 30% under medium load.

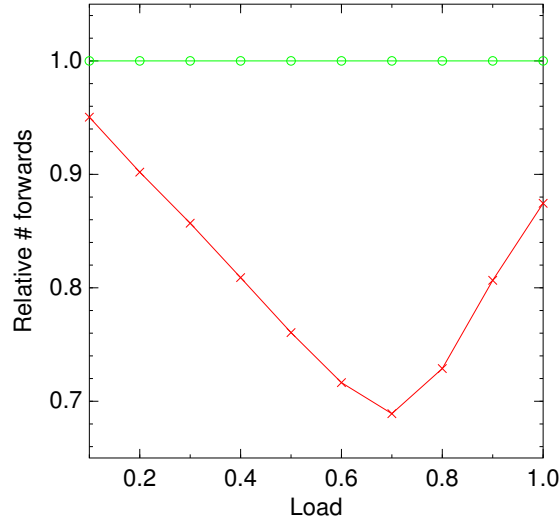


Figure 7: Random unvisited forwarding

(Random) Coprime offset

Two other algorithms that are not restricted to forwarding to a neighbor are Coprime offset and Random coprime offset. Figure 8 shows the results of both these algorithms. The difference of these algorithms with themselves and the random unvisited algorithms is not clearly visible when comparing both to the *forward right* algorithm. To put these results in perspective, we included figure 9 where Coprime offset and Random Coprime offset are depicted relative to Random unvisited. Although the difference is very small, it seems the Random unvisited algorithm shows a better performance than the other two. However, Coprime offset and Random Coprime offset require $\lceil \log_2 \varphi(N) \rceil$ bits in the job's metadata instead of N for Random unvisited.

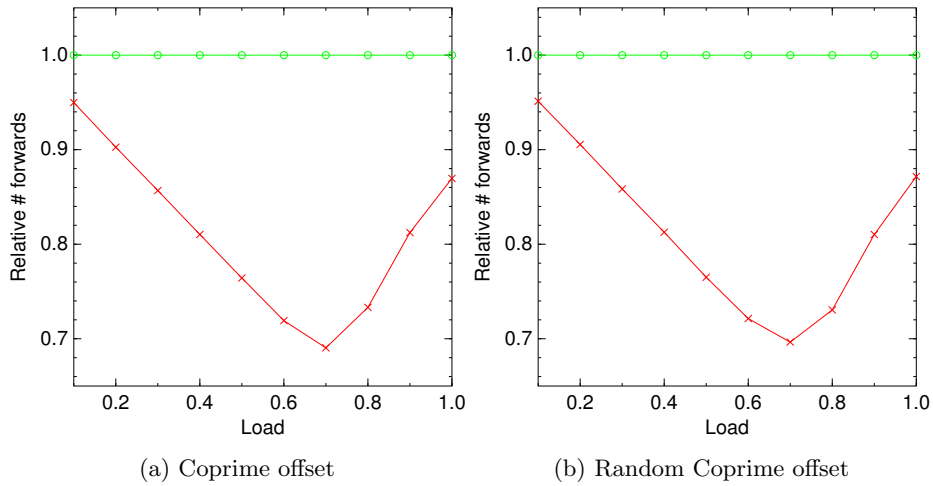


Figure 8: (Random) Coprime offset

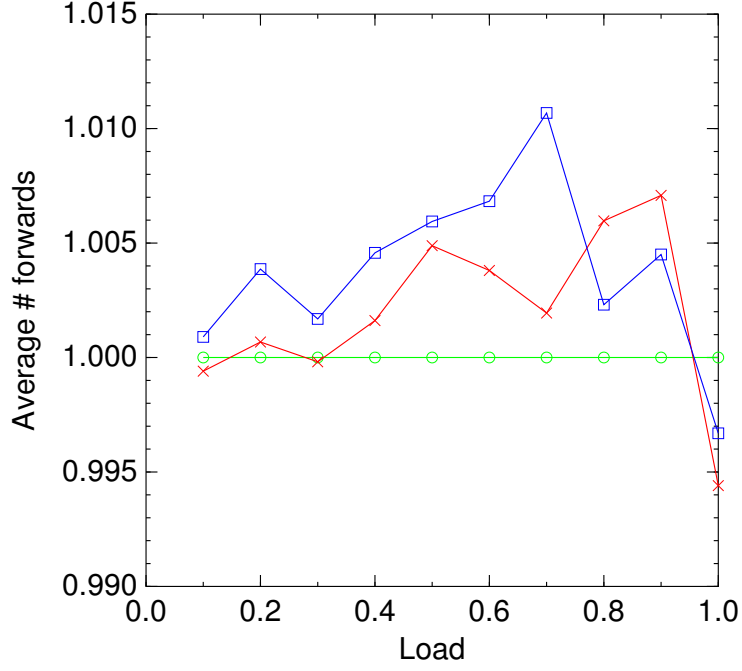


Figure 9: Random unvisited (green), Coprime offset (blue), Random Coprime offset (Red)

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 behavior of the algorithms when using multiple CPUs, and comparing the results to a system with 1 CPU per server, and to each other.

Per algorithm, two simulations were performed and compared. One using 1 CPU per server and one using 4 CPUs per server. See table 2 for more information about the tests.

| | | |
|----------------------|---|----|
| Algorithms | Forward right, Left/right forward, Random left/right (0.5), Random unvisited, Random coprime offset | |
| Ring size | 100 | 25 |
| CPUs per node | 1 | 4 |
| Total CPUs in system | 100 | |
| Load | 0.1 – 1.0 | |

Table 2: Comparison of algorithms using multiple CPUs 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 CPUs. The baseline for the tests is the result of the algorithm using 1 CPU (found in section 2.2), where the results of the simulation using multiple CPUs is drawn relative to the baseline result.

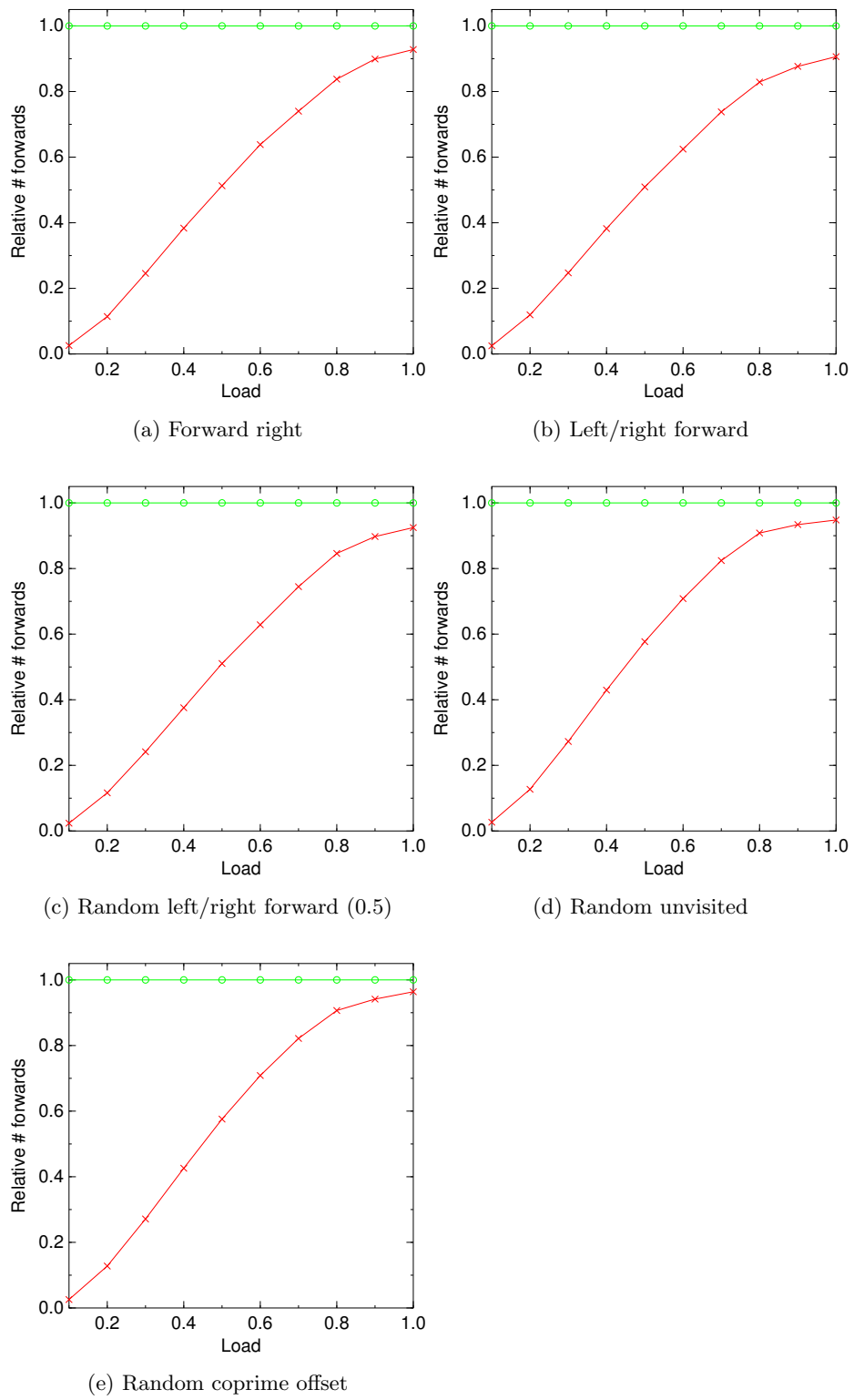


Figure 10: 4 CPUs per server versus 1

| # hops | Distribution | M | Result |
|-------------------|--------------------------|---|--------|
| 0 | 0.5092 | 0 | 0 |
| 1 | 0.2118 | 0 | 0 |
| 2 | 0.1082 | 1 | 0.1082 |
| 3 | 0.0609 | 1 | 0.0609 |
| 4 | 0.0363 | 2 | 0.0726 |
| 5 | 0.0224 | 2 | 0.0449 |
| 6 | 0.0142 | 3 | 0.0426 |
| 7 | 0.0091 | 3 | 0.0272 |
| 8 | 0.0058 | 4 | 0.0233 |
| 9 | 0.0037 | 4 | 0.0147 |
| Total | 0.9816 | | 0.3944 |
| Weighted total | $0.3944/0.9816 = 0.4018$ | | |
| Simulation result | 0.4171 | | |

Table 3: Comparison when load=0.5

Figure 10 shows the performance of 4 CPUs versus 1 CPU for different algorithms. For each tested algorithm, we obtain the same result. This means the results of each algorithm are influenced in the same way when using multiple CPUs per server. We can use this knowledge to generalize the results of section 2.2.

Since we assume each algorithm is influenced the same way, we will investigate one of them in depth. We will try to transform the results of an algorithm with ring size N and 1 CPU per server into the results of the same algorithm with ring size N/c and c CPUs per server. Every other parameter should be unchanged.

Let d be the distribution of the number of forwards, stored as a row vector. We will build a transformation vector M in the form $[[0/c], [1/c], \dots, [(N-1)/c]]'$. This vector represents the number of times a job would be forwarded if there would be c CPUs per server. $d * M$ equals the average number of forwards when using such a system. To ignore the dropped jobs, the results must be weighted for only the completed jobs. In our example, we will transform the results of the *forward right* algorithm using ring size $N = 10$ and 1 CPU into the results of a ring with size $N = 5$ and $c = 2$ CPUs per server, see table 3 for a worked out example and figure 11 for a comparison of the transformation and a simulation result.

While deriving the transformation, we did not take into account that for each choice the algorithm makes, c CPUs are bound to this choice. The real simulation chooses a node and must first test all those CPUs where the transformation only counts the $n \cdot c$ forwards using a better choice for each forward. This makes the transformation nothing more than an approximation and an upper bound of the real results.

3 Numerical Validation

To validate the results obtained in the previous section, we modeled the scheduling algorithms into Markov Chains. Using the steady state distribution

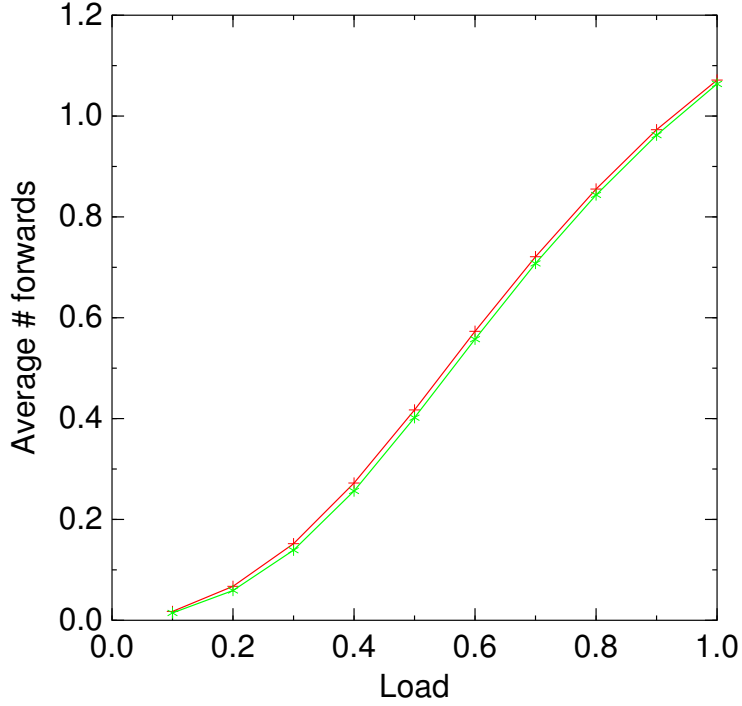


Figure 11: Multiple CPUs result derived of the result for 1 CPU (green) versus an actual simulation result using 4 CPUs (red)

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 matrices 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 are still relevant.

Not every algorithm can be modeled efficiently into a Markov Chain. Computing the results of algorithms requiring state information in the nodes is infeasible for a decent ring size, therefore we did not model *left/right forward* and the Coprime offset algorithms. We also ignored the Position dependant forwarding algorithm since its results seemed logical and further investigation would not reveal useful information.

3.1 Comparison

Forward right

Modeling 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 in their binary form.

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

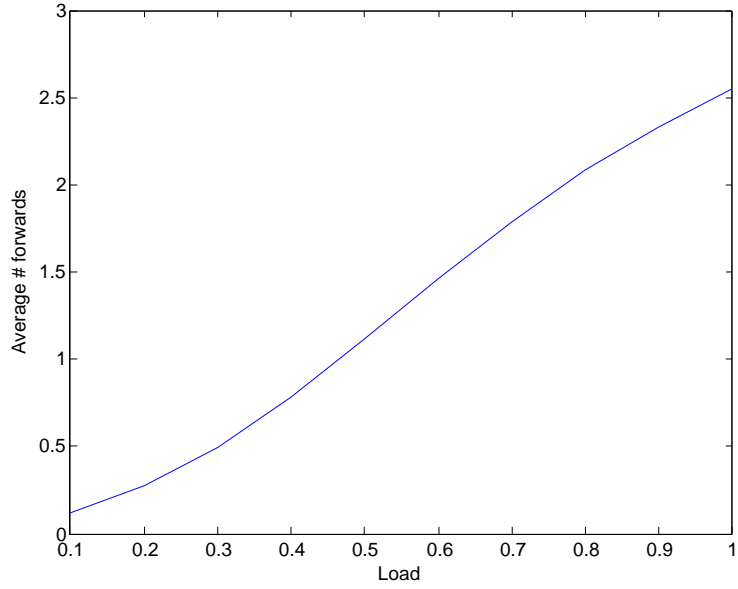


Figure 12: Validation of forward right

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

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.

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

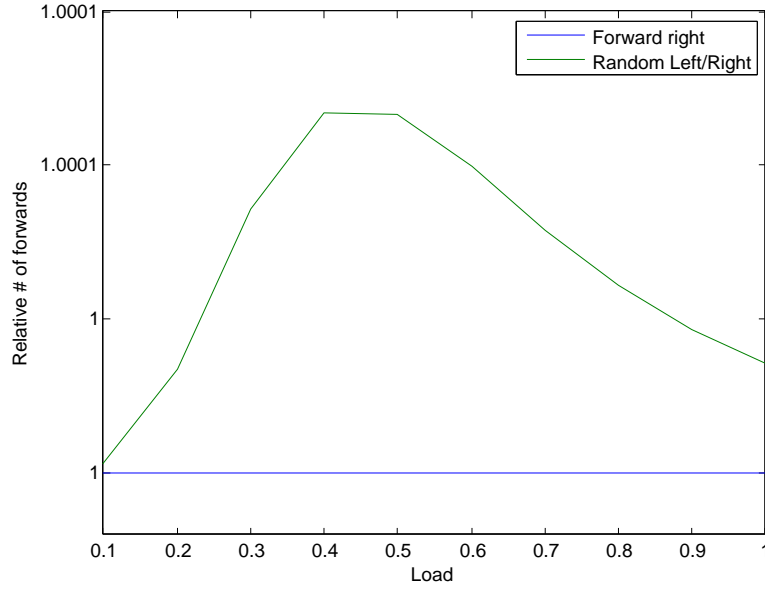


Figure 13: Validation of Random left/right with $p = 0.5$

It seems the results of the simulation fit. Although the results in 13 are much smoother than those in 4, we must take into account that these results are computed exact and using a smaller ring size.

As in the simulation section, we have validated the results for different values of p . These results are shown in figure 14.

The random left/right forward algorithm does not seem a viable choice for real implementations. It has a small performance drawback compared to *forward right* and has no other benefits.

Random Unvisited

This problem can be modeled much more efficiently than the techniques. Since the next node is chosen at random, it is not necessary to store the position of the nodes. The information we need to save consists only of the number of servers that are currently busy. This problem is similar to modeling an Erlang-B loss

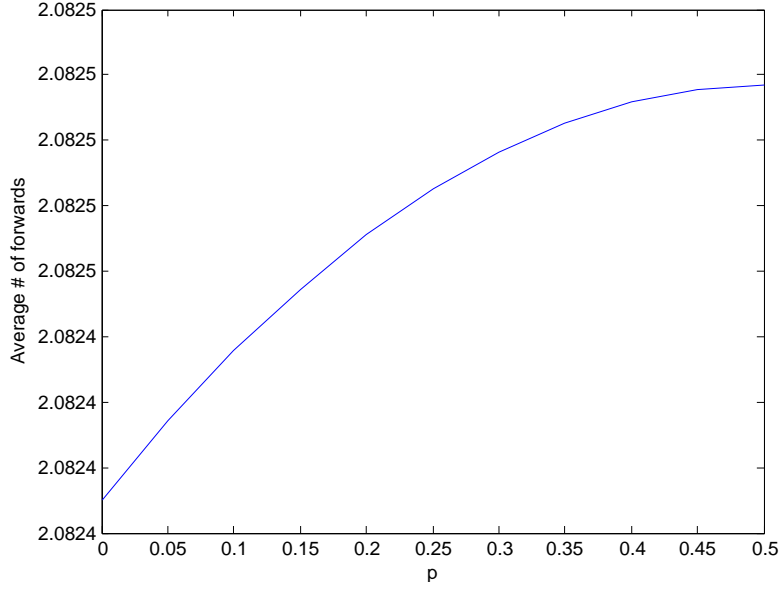


Figure 14: Performance of random left/right with load= 0.8

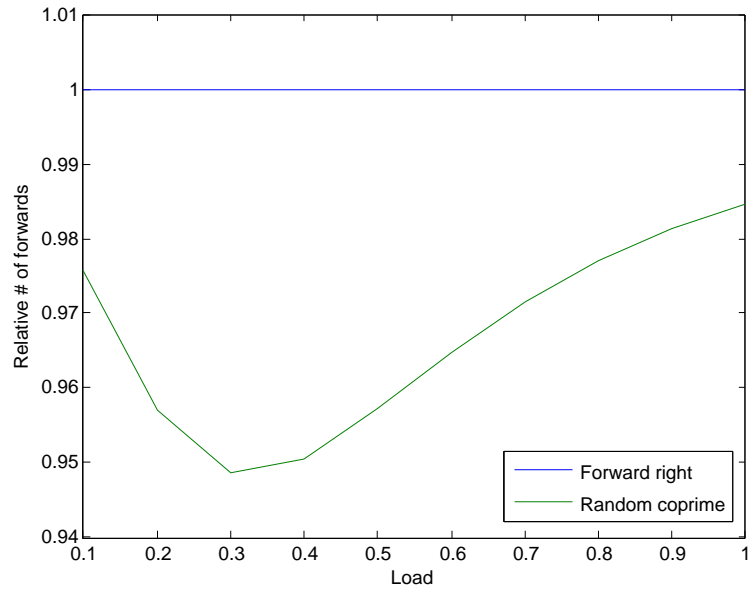
system. The number of states in this Markov Chain is linear to N , which is much more dense than the previous given models. For $N = 3$, the matrix is given below.

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

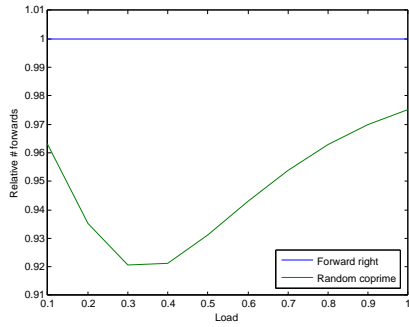
Random Coprime offset

Modeling this technique yields different results for various ring sizes. The performance of this algorithm depends on $\varphi(N)$. The results in figure 15 are very similar to those found in the simulation (figure 8b).

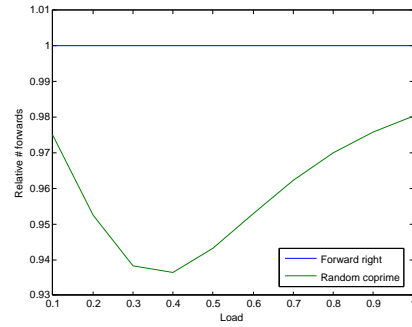
The results show a performance gain up to 5% for a ring size of $N = 10$. The list of coprimes in that scenario is 1, 3, 7, 9, so 4 possible choices. When increasing the ring size to 11, a prime number, the list of coprimes expands to 1..10 (because 11 is prime), so 10 possible choices. This increases the relative performance gain up to 8% (figure 15b). To make clear the increased performance is not due to the increase of N , figure 15c shows the performance gain for $N = 12$. For $N = 10$ and $N = 12$, a job can follow 4 possible routes, for $N = 11$, 10 different routes can be chosen.



(a) $N = 10$



(b) $N = 11$



(c) $N = 12$

Figure 15: Validation of the Random Coprime offset algorithm

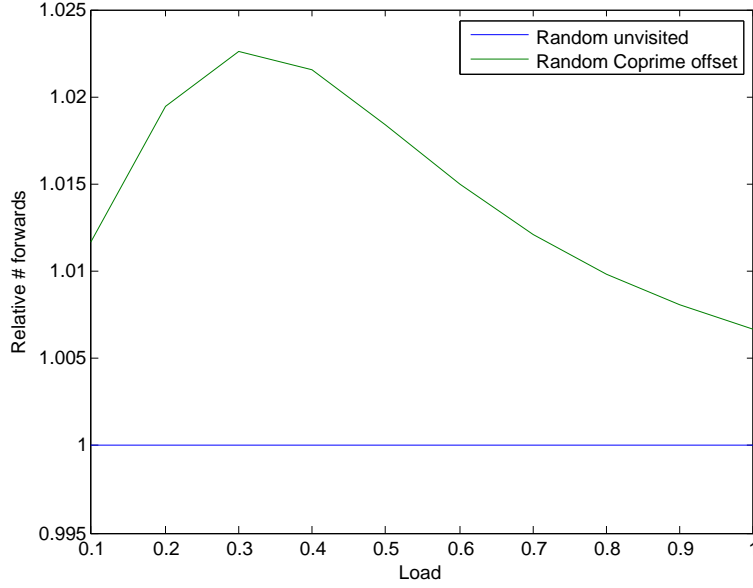


Figure 16: Performance of Random Coprime offset relative to Random unvisited

3.2 Lumped states

Except for Random Unvisited, each described technique is modeled 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 a state where one of the nodes is busy. For states representing multiple busy nodes, the space between these servers is critical information. Redundant states can be generated from one state: bitrotate the state by 1 until you get the original state. Each encountered state is redundant. Example: the states below are redundant and can therefore be lumped into one state:

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

The example model of the *forward right* algorithm can be lumped into the following Markov Chain:

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

Computing the steady state distribution of a Markov Chain is subject to both time and memory constraints. Using sparse matrices for our algorithms reduces the memory constraint so much it is no longer a bottleneck. 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 by lumping the matrices.

For unlumped Markov Chains modeling 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 $\frac{1}{N} \sum_{d|N} (2^{N/d} \cdot \varphi(d))$ [2]. 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 18 shows a clear reduction as well. Yet, this result does not seem polynomial either.

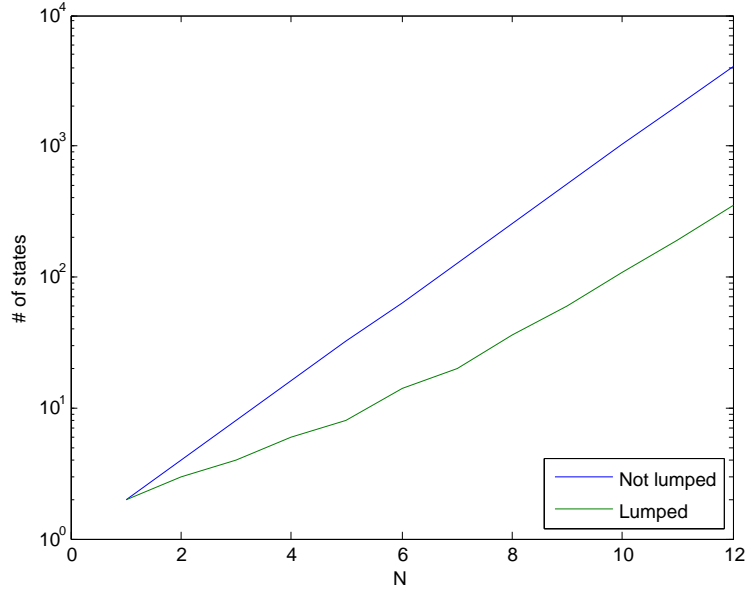


Figure 17: Number of states

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

3.3 Equivalent algorithms

Various algorithms are equivalent or very similar under special conditions. We will discuss some of these equivalencies.

Low N

For $N \leq 3$ all algorithms are equivalent. This is obvious for $N = 2$. For $N = 3$, there are 4 possible states with the same behavior for each algorithm:

All nodes idle All 3 nodes are idle, an incoming job will be executed by first node.

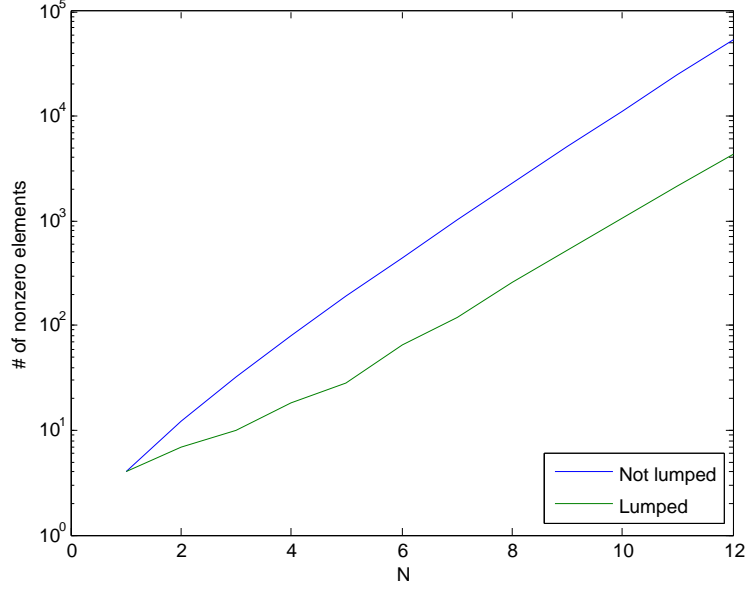


Figure 18: Number of nonzero elements

1 node busy Each node is as likely to be the next node receiving a job. For an idle node, the job is executed by that node. For the busy node, the job will need to make exactly 1 hop.

2 node busy Since each node is as likely to be the next node receiving a job, the job will be forwarded 0,1 or 2 times, all with the same probability.

Saturated The job will be dropped.

For $N > 3$, other equivalencies exist. These are described below.

High load

When the load becomes sufficiently large, all algorithms will yield the same results. In a ring under heavy load, each node will be busy nearly all the time. As long all nodes are busy, incoming jobs will be dropped. When a node becomes idle, the first event will be an incoming job. The first incoming job will claim this node. Each node has the same probability of receiving the next incoming job, so the expected value for the number of forwards is:

$$\begin{aligned} \frac{1}{N} \cdot \sum_{i=0}^{N-1} i &= \frac{1}{N} \cdot \frac{N(N-1)}{2} \\ &= \frac{N-1}{2} \end{aligned}$$

Random left/right with parameter p versus forward right

This algorithm is always equivalent to itself using parameter $1 - p$. This is intuitively clear because the probabilities of right and left are swapped, like

looking to the ring in a mirror. It is clear that for $p = 0$ (and so for $p = 1$), this algorithm is equivalent to the *forward right* algorithm. Furthermore, for arbitrary values of p , this algorithm is still equivalent to *forward right* as long as the ring size $N \leq 5$. This can be proven by lumping the Markov Chains for both techniques.

For $N = 4$, define the lumped Markov Chain of the *forward right* algorithm.

$$Q_1 = \begin{array}{c} \begin{array}{cccccc} & 0000 & 0001 & 0011 & 0101 & 0111 & 1111 \end{array} \\ \begin{array}{l} 0000 \\ 0001 \\ 0011 \\ 0101 \\ 0111 \\ 1111 \end{array} \end{array} \begin{bmatrix} -4\lambda & 4\lambda & 0 & 0 & 0 & 0 \\ \mu & -4\lambda - \mu & 3\lambda & \lambda & 0 & 0 \\ 0 & 2\mu & -4\lambda - 2\mu & 0 & 4\lambda & 0 \\ 0 & 2\mu & 0 & -4\lambda - 2\mu & 4\lambda & 0 \\ 0 & 0 & 2\mu & \mu & -4\lambda - 3\mu & 4\lambda \\ 0 & 0 & 0 & 0 & 4\mu & -4\mu \end{bmatrix}$$

Now, define the lumped Markov Chain of the FR algorithm with parameter p and simplify.

$$Q_2 = \begin{array}{c} \begin{array}{cccccc} & 0000 & 0001 & 0011 & 0101 & 0111 & 1111 \end{array} \\ \begin{array}{l} 0000 \\ 0001 \\ 0011 \\ 0101 \\ 0111 \\ 1111 \end{array} \end{array} \begin{bmatrix} -4\lambda & 4\lambda & 0 & 0 & 0 & 0 \\ \mu & -\mu & \lambda + (1-p)\lambda + p\lambda + \lambda & \lambda & 0 & 0 \\ 0 & 2\mu & -2\mu & 0 & \lambda + 2\lambda \cdot ((1-p) + p) + \lambda & 0 \\ 0 & 2\mu & 0 & -2\mu & \lambda + p\lambda + (1-p)\lambda & 0 \\ 0 & 0 & 2\mu & \mu & -x & 4\lambda \\ 0 & 0 & 0 & 0 & 4\mu & -4\mu \end{bmatrix}$$

$$Q_2 = \begin{array}{c} \begin{array}{cccccc} & 0000 & 0001 & 0011 & 0101 & 0111 & 1111 \end{array} \\ \begin{array}{l} 0000 \\ 0001 \\ 0011 \\ 0101 \\ 0111 \\ 1111 \end{array} \end{array} \begin{bmatrix} -4\lambda & 4\lambda & 0 & 0 & 0 & 0 \\ \mu & -4\lambda - \mu & 3\lambda & \lambda & 0 & 0 \\ 0 & 2\mu & -4\lambda - 2\mu & 0 & 4\lambda & 0 \\ 0 & 2\mu & 0 & -4\lambda - 2\mu & 4\lambda & 0 \\ 0 & 0 & 2\mu & \mu & -4\lambda - 3\mu & 4\lambda \\ 0 & 0 & 0 & 0 & 4\mu & -4\mu \end{bmatrix}$$

We find $Q_1 = Q_2$ for $N = 4$. This also works for $N = 5$, but for $N \geq 6$, the matrices are different.

Random Coprime offset versus forward right

For $N = 4$, this algorithm is equivalent to the *forward right* algorithm as well. The coprimes of 4 are 1 and 3. Meaning a jump of 1 (right neighbor) or a jump of 3 (left neighbor). It is clear this is the same as *random left/right forward with parameter 0.5*, which is equivalent to the *forward right* algorithm.

Random Coprime offset versus Random unvisited

The performance of the Random Coprime offset algorithm depends on $\varphi(N)$, which is $N - 1$ when N is prime. Figure 19 shows the average ring traversal of this algorithm and the Random unvisited algorithm. It seems for prime values

of N , the graphs touch. However, when looking at the actual data we see this is only true for $N = 5$.

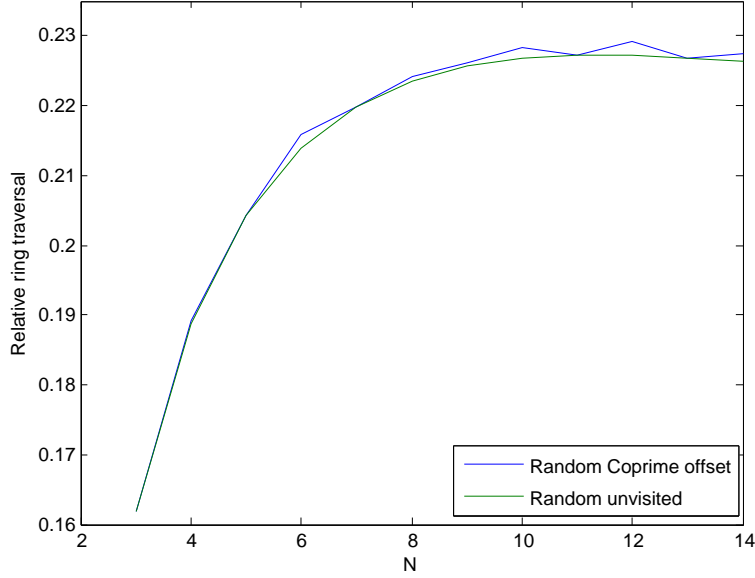


Figure 19: Ring traversal versus N for load=1.0

| N | Random unvisited | Random Coprime offset |
|----|-------------------|-----------------------|
| 3 | 0.161764705882353 | 0.161764705882353 |
| 5 | 0.204376460590610 | 0.204376460590610 |
| 7 | 0.219792926622134 | 0.219788935482327 |
| 11 | 0.227265522566042 | 0.227249844727623 |
| 13 | 0.226908433098761 | 0.226887197455899 |

Table 4: Relative ring traversal for load=1.0

4 Conclusion

We have simulated all our algorithms and validated the results of some of them. The ideal algorithm is depends on the specific requirements of the distributed system. However, it is safe to say some algorithms are not ideal for any situation. One should also note that in some configurations, different algorithms can yield the same results. In those cases, only memory requirements or the complexity to implement the algorithms should be decisive.

Forward to neighbors

When nodes are only directly connected to their neighbors, the choice depend on the memory requirements. When at least 1 bit can be saved in both the job's metadata as the node's state, *left/right forward* is an ideal choice. This

algorithm provides up to 4% better result than *forward right* under medium load ($N = 100$).

When no memory is available, the *forward right* algorithm is the only possible choice while maintaining sufficient results. Additionally, it is the easiest algorithm to implement.

Both these algorithms allow nodes to come and go. The random left/right and Position dependent forwarding techniques perform worse than *forward right* while having more requirements, therefore they should not be chosen.

Forward anywhere

If each node can reach each other node directly, the forward anywhere algorithms improve the results significantly. The best results can be achieved using the Random unvisited algorithm. This algorithm performs slightly better than coprime offset and random Coprime offset, but at a higher cost: the random unvisited algorithm requires $N - 1$ bits in the job's metadata.

The Coprime offset and Random Coprime offset algorithms require less memory but with worse performance. However, when the ring size is prime, its results are only slightly less than those of random unvisited. The memory required in the job's metadata is lowered to $\lceil \log_2 \varphi(N) \rceil$ bits ($\lceil \log_2(N - 1) \rceil$ bits when N is prime). The coprime offset algorithm yields about the same results as the random variant. In addition, it needs to save $\lceil \log_2 \varphi(N) \rceil$ bits in the state of each node. Random coprime offset and random unvisited seem two very good candidates, it depends whether the small performance benefit is worth the memory and implementation cost.

Other findings

The results of using multiple CPUs per server can be approximated using the the results of 1 CPU per server and a ring size $N_1 = N_c \cdot c$. This approximation is an upper bound to the real outcome since the derivation ignores the fact that the choice for a node is no longer a choice for a CPU, it is a choice for a group of c CPUs.

For the *random left/right forward with parameter p* , we have found out its results are worst for $p = 0.5$. We did not come up with an explanation why this algorithm is worse than *forward right*. This might be a start for further research.

We have also found a way to model some of these algorithms into Markov Chains. This allowed us to compute the actual result of a simulation. The drawback of our model is the limited ring size. We optimized it by lumping states and using sparse matrices, but because of the non-polynomial nature of the matrix size, these optimizations allow us to increase N only by 2 or 3.

Another finding is the equivalency between multiple algorithms, under certain conditions some algorithms model the exact same problem.

References

- [1] *Euler's totient function*. URL: http://en.wikipedia.org/wiki/Euler%27s_totient_function.

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

A Simulator source code

```
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(1
28                 )), r->getNode(i)));
29         }
30     }
31 }
32
33 void fillEvents(Configuration c, Ring* r){
34     double t;
35     double l;
36     Node* n;
37     for (unsigned int i = 0; i < r->getSize(); ++i){
38         n=r->getNode(i);
39         t = 0.0;
40         while (t < c.length){
41             t+=exp_distr(c.arrival);
42             l=exp_distr(c.joblength);
43             r->getSimulator()->addEvent(new ArriveEvent(t, new Job(c.makeInfoFunction(1)
44                 , n));
45         }
46     }
47 }
48
49 int main(int argc, char** argv) {
50     Configuration c(argc, argv);
51
52     double success = 0.0;
53     double avghops = 0.0;
54
55     cout.setf(ios::fixed, ios::floatfield);
56     cout.precision(12);
57
58 #pragma omp parallel for
59     for (unsigned int i = 0; i < c.repeat; ++i){
60         Ring r(c.nodes, c.nodeSize, c.makeNodeFunction());
61
62         //preload(c, &r); //disabled to compare output to more early results
63
64         fillEvents(c, &r);
65
66         if (c.progressinterval <= 0){
67             r.getSimulator()->run();
68         } else{
69             r.getSimulator()->run(c.progressinterval);
70         }
71     }
```

```

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() << endl;
77     cout << "Discarded_jobs:\t\t" << r.getDiscardedJobs() << endl;
78     cout << "Total_hops_(finished):\t" << r.getFinishedJobTotalHops() << endl;
79     long totalhops = r.getFinishedJobTotalHops() + (c.nodes-1) * r.
        getDiscardedJobs();
80     cout << "Total_hops_(all):\t" << totalhops << endl;
81     cout << "Hops/job_(finished):\t" << (double)r.getFinishedJobTotalHops()/r.
        getFinishedJobs() << endl;
82     cout << "Hops/job_(total):\t" << (double)totalhops/r.getTotalJobs() << endl;
83     cout << "Success_ratio:\t\t" << (100.0 * r.getFinishedJobs()) / r.
        getTotalJobs() << "%" << endl;
84     success+=(double)(r.getFinishedJobs()) / r.getTotalJobs();
85     avghops+=(double)r.getFinishedJobTotalHops()/r.getFinishedJobs();
86 }
87 }
88
89 if (c.repeat > 1){
90     cout << "_____ " << endl;
91     cout << "Avg._hops/job_(finished):\t" << avghops / c.repeat << endl;
92     cout << "Avg._success_ratio:\t\t" << 100.0 * success / c.repeat << "%" << endl;
93 }
94
95 return 0;
96 }

```

Listing 2: Main.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, unsigned int size);
17 typedef pop::JobInfo* (*makeInfoType)(double length);
18 struct Configuration{
19     unsigned int seed;
20     double joblength;
21     double arrival;
22     long nodes;
23     unsigned int nodeSize;
24     long progressinterval;
25     long length;
26     long repeat;
27     makeNodeType makeNodeFunction;
28     makeInfoType makeInfoFunction;
29
30     Configuration(int argc, char** argv);
31 };
32
33 #endif /* CONFIGURATION_H_ */

```

Listing 3: configuration.h

```

1 #include <iostream>
2 #include <stdlib.h>
3 #include "configuration.h"
4 #include "nodes.h"
5
6 using namespace std;

```



```

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 4: configuration.cpp

```

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 int size):
54         RightNode(id, ring, size), last(1)
55     {}
56
57     bool pushJob(pop::Job* j);
58
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, unsigned int size):
72         RightNode(id, ring, size)
73     {}
74
75     bool pushJob(pop::Job* j);
76
77 private:
78     static double v;
79 };
80
81 class EvenSwitchNode: public RightNode{
82 public:
83     typedef DirectionInfo info_type;
84
85     inline EvenSwitchNode(unsigned int id, pop::Ring* ring, unsigned int size):
86         RightNode(id, ring, size)
87     {}
88
89     bool pushJob(pop::Job* j);
90 };

```

```

91
92 class PrimeNode: public SwitchNode {
93 public:
94     typedef DirectionInfo info_type;
95
96     static void makePrimes(unsigned int size);
97
98     inline PrimeNode(unsigned int id, pop::Ring* ring, unsigned int size):
99         SwitchNode(id, ring, size)
100     {
101         if (fPrimes == 0){
102             makePrimes(ring->getSize());
103         }
104     }
105
106     bool pushJob(pop::Job* j);
107
108 protected:
109     static int* fPrimes;
110     static int fPrimesLen;
111 };
112
113 class RandPrimeNode: public PrimeNode{
114 public:
115     typedef DirectionInfo info_type;
116
117     RandPrimeNode(unsigned int id, pop::Ring* ring, unsigned int size):
118         PrimeNode(id, ring, size)
119     {
120         if (fPrimes == 0){
121             makePrimes(ring->getSize());
122         }
123     }
124
125     bool pushJob(pop::Job* j);
126 };
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 5: nodes.h

```

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     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->getSimulator()->getTime()+
41             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;

```

```

71         last*=-1;
72     }
73
74     j->forward(fRing->getNode(this->fId + ji->fDirection));
75 }
76 return true;
77 }
78
79 double RandSwitchNode::v = 0.5;
80
81 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->visited.size());
190             next = x;
191
192             for (set<unsigned int>::iterator it = ji->visited.begin(); it != ji->visited.
193                 end(); it++){
194                 if (*it <= next){
195                     ++next;
196                 }
197             }
198         } else {
199             do {
200                 next = rand() % fRing->getSize();
201             } while (ji->visited.count(next));
202         }
203
204         j->forward(fRing->getNode(next));
205     }
206     return true;
207 }
208
209 bool ToTopNode::pushJob(Job* j){
210     if (wasHereFirst(j)){
211         return false;
212     }
213
214     if (!accept(j)){
215         info_type* ji = dynamic_cast<info_type*>(j->getInfo());
216         if (ji->fDirection == 0){
217             if (this->getId() > fRing->getSize()/2){
218                 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 RRUvisited::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 6: nodes.cpp

```

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: servernode.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 8: servernode.cpp

```

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 9: ring/arriveevent.h

```

1 /*
2  * finiscevent.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 10: ring/finishevent.h

```

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 11: ring/job.h

```

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(ji)
17     {}
18
19 Job::~~Job() {
20     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: " << fStart << " / #
28     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: " << fStart << " /
36     finish time: " << fFinish << " / #hops: " << fHops << ")" << endl;
37     fCurrent->clearJob(this);
38     delete this;
39 }
40
41 void Job::forward(Node* n){
42     if (!fCurrent){
43         n->getRing()->jobCreated();
44         fStart = n->getRing()->getSimulator()->getTime();
45     }
46     ++fHops;
47     fCurrent = n;
48     if (!n->pushJob(this)){
49         discard();
50     }
51 }
52 } /* namespace pop */

```

Listing 12: ring/job.cpp

```

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 13: ring/node.h

```

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 14: ring/node.cpp

```

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)(unsigned int i, Ring*
21         r, unsigned int ns));
22     virtual ~Ring();

```

```

23     inline unsigned int getSize(){
24         return fSize;
25     }
26
27     inline Node* getNode(int id){
28         //std::cout << "id: " << id << " = " << (id + fSize) % fSize << std::endl;
29         return fRing[(id + fSize) % fSize];
30     }
31
32     inline Simulator* getSimulator(){
33         return &fSimulator;
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 15: ring/ring.h

```

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)(unsigned int i, Ring
    * r, unsigned int ns)):
13     fSize(size),
14     fRing(new Node*[size]),
15     jobsTotal(0), jobsFinished(0), jobsDiscarded(0), jobsFinishedTotalHops(0)
16 {
17     for (unsigned int i = 0; i < size; ++i){
18         fRing[i] = mkNode(i, this, nodesize);

```

```

19     }
20 }
21
22 Ring::~~Ring() {
23     for (unsigned int i = 0; i < fSize; ++i){
24         delete fRing[i];
25     }
26     delete[] fRing;
27 }
28
29 } /* namespace pop */

```

Listing 16: ring/ring.cpp

```

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 17: simulator/event.h

```

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 18: simulator/schedule.h

```

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 19: simulator/schedule.cpp

```

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 20: simulator/simulator.h

```

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 21: simulator/simulator.cpp

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

```

Listing 22: rightchain.m

```

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

```

```

33         bt=bt-1;
34         r = r + rate*(1-p);
35     end
36     Q(i+1, j+1)=r;
37 end
38     t=t + Q(i+1, j+1);
39 end
40     Q(i+1, i+1) = -t;
41 end
42
43 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-p, size);
34                        c=c+1;
35                    end
36                end
37                Q(i+1, i+k+1) = rate + c*rate/rpcount;
38                tot=tot+Q(i+1, i+k+1);
39            end
40        end
41        Q(i+1, i+1) = -tot;
42    end
43
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,distribution ] = 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 %Return:
8 %     avg      The average number of forwards
9 %     distribution  The distribution for each possible outcome
10
11     if nargin < 2
12         d=1;
13     end
14
15     steady=full(ctmcsteadystate(Q));
16     distribution=zeros(1,d);
17
18     len=length(Q);
19     states=log2(len);
20     avg=0;
21     total=0;
22
23     for i=0:(states-1)
24         c=0;
25         prefix=((2^i)-1) * 2^(states-i);
26         for j=0:(2^(states-i-1))-1
27             c=c+steady(prefix + j + 1);
28         end
29         total=total+c;
30         if d
31             fprintf(' %d_hops:\t%f\n', i, c);
32         end
33         distribution(i+1)=c;
34         avg=avg+(c*i);
35     end
36
37     loss=steady(len);
38     avg=avg/(1-loss);
39     if d
40         fprintf(' Loss:\t%f\nTotal:\t%f\nAverage_#hops:\t%f\n', loss, total + loss
41             , avg);
42     end
43 end

```

Listing 26: avghops.m

```

1 function [ avg,avgp ] = ruavghops( Q, d )
2 %RUAVGHOPS Calculate average number of times a job is forwarded for the random unvisited
3   chain
4 %Parameters:
5 %     Q        The matrix representing a markov chain using the random unvisited
6   forwarding algorithm
7 %Optional:
8 %     d        Debug mode, default=1, disable debug output=0
9 %Return:
10 %     avg      The average number of times a job is forwarded
11 %     avgp     The distribution
12
13     if nargin < 2
14         d=1;

```

```

13     end
14
15     steady=ctmcsteadystate(Q);
16
17     len=length(Q);
18
19     avg = 0;
20     avgp = zeros(len,1);
21
22     for i=0:len-2
23         tmpavg = 0;
24         for h=0:i
25             c = prod(i-h+1:i) * (len-1-i) / prod(len-1-h:len-1);
26             tmpavg = tmpavg + (c * h);
27             avgp(h+1) = avgp(h+1) + (c*steady(i+1));
28         end
29         avg=avg + steady(i+1) * tmpavg;
30     end
31
32     avgp(len) = steady(len);
33
34     loss=steady(len);
35     avg=avg/(1-loss);
36     if d
37         avgp
38         fprintf('Loss:\t%f\nAverage #hops:\t%f\n', loss, avg);
39     end
40
41 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
6         10)
7
8     T=Q;
9     len=length(Q);
10    T(:,len)=ones(len, 1);
11    e=zeros(1, len);
12    e(len)=1;
13    pi=e*inv(T);
14 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
3   lumped
4 %Parameters:
5 %     Q      A matrix representation of a markov Chain
6
7     fullsize=length(Q);
8     [Q S]=lump(Q);
9     lumpsize=length(S);
10    nodes=log2(fullsize);
11    hops=zeros(1,nodes+1);
12
13    steady=ctmcsteadystate(Q);
14
15    hops(1)=steady(1); %zero hops
16    hops(nodes+1)=steady(lumpsize); %loss
17    for i=2:lumpsize-1;
18        bits=ceil(log2(S(i)+1));
19        hops(1)=hops(1)+(nodes-bits)/nodes*steady(i);
20
21        for j=bits:-1:1
22            c=0;
23            while c<j && bitand(S(i), 2^(j-c-1))
24                c=c+1;
25            end
26        end
27    end
28 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+1, size);
59         else
60             v = [f(word + powers(place), bits-1, place+1, size) f(word, bits,
61                 place+1, size)];
62         end
63     end
64
65     function [r] = makecombs(k, n)
66         leadzeros = ceil(n/k)-1;
67         fullsize = n - leadzeros - 1;
68         r = f(0,k-1,1,fullsize)*2 + 1;
69     end
70
71     r=[0 2^(rsize)-1];
72     for i=1:rsize-1
73         r = [r makecombs(i, rsize)];
74     end
75
76     s=[];
77     for i=r
78         s=[s reindex(i+1)];
79     end
80
81     r=unique(s);
82     reindex = arrayfun(@(x) find(r == x), reindex);
83
84     coverage=[];
85     for w=r
86         coverage=[coverage cover(w, rsize)];
87     end
88
89 end

```

Listing 31: makestates.m