# 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 toghether 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

# 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 indepentently 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 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 whereto 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.

** AANTAL NODES IN FIGURE IS FOUT

The performance of a forwarding algorithms 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 is 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 techniques 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

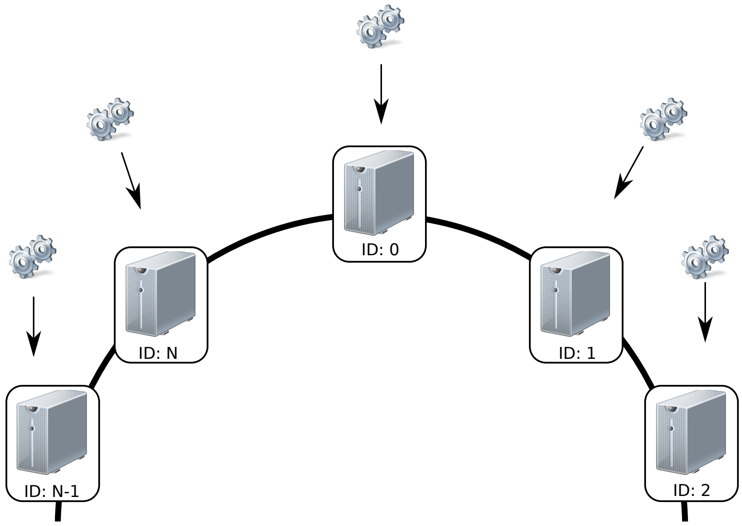


Figure 1: A ring structured network

average number of hops. One should note that the loss rate of jobs in the system is the same a 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 algorithms 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 $N - 1$, neigbouring 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 strucure 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 choosen 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 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 the 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 than generated: $1, 3, 7, 9$. Assume a job arrives at node 3 and the last time node 3 forwarded a job it was given offet 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 wich 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. The source code of the simulator can be found in appenix A or on `http://code.google.com/p/powerofpaths/`.

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

```
Usage: -r -s long -j double -a double -n long -p long -l long -t
    long -h type
        -r      Random seed
        -s      Set seed                        (default: 0)
        -j      Job length                      (default: 1.0)
        -a      Interarrival time               (default: 1.0)
        -n      Ring size                       (default: 100)
        -p      Print progress interval (default: -1 - disabled)
        -l      Simulation length               (default: 3600)
        -t      Repetition              (default: 1)
        -h      Print this help
         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.

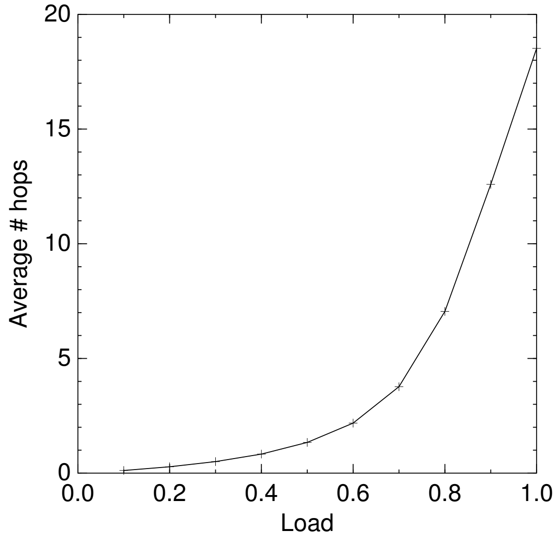


Figure 2: The Forward Right baseline result

## 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.

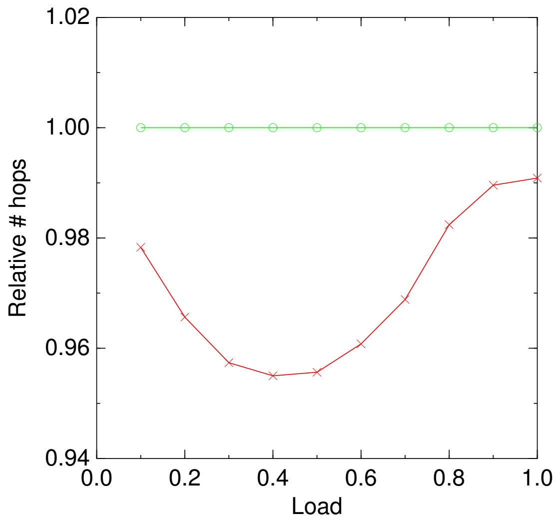


Figure 3: Left/Right

### 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.

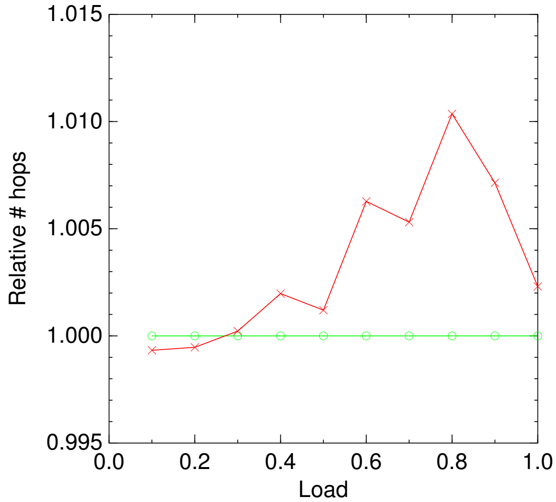


Figure 4: Random Left/Right forward with parameter 0.5

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 is increases a little until arriving at 0.5.

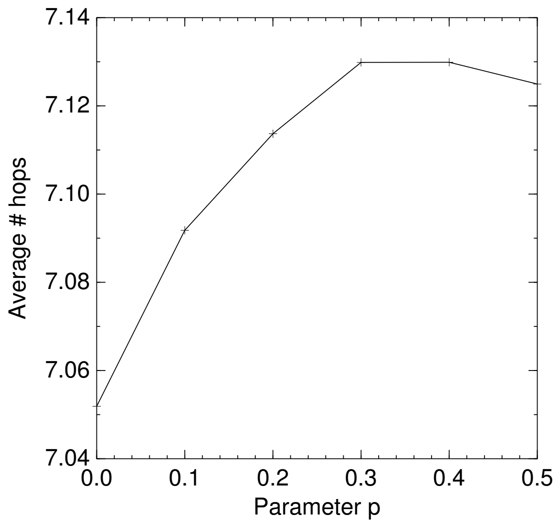


Figure 5: Random Left/Right forward with load 0.8

### 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.

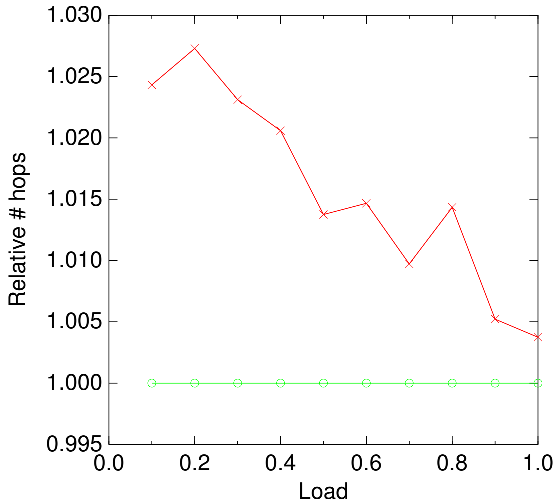


Figure 6: Position-dependant forwarding

### Random unvisited

This algorithm in 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.

## 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.

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

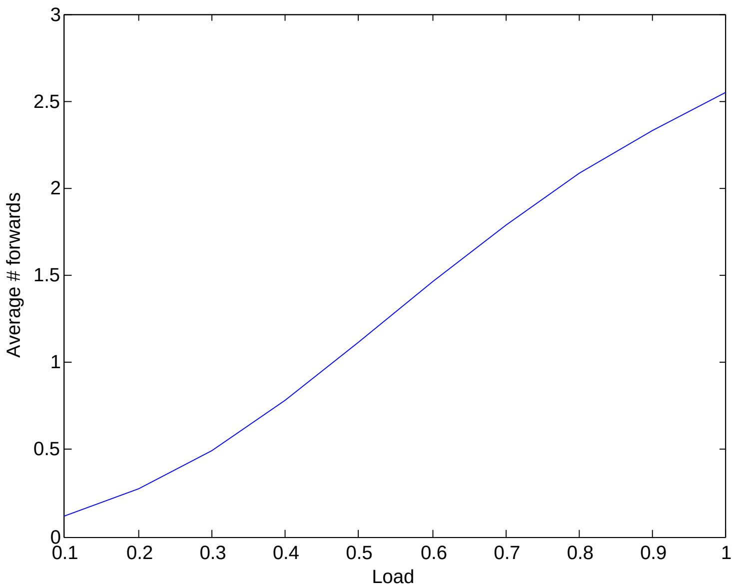


Figure 7: Validation of Forward right

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.

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

The lumped matrix (section 13) 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.

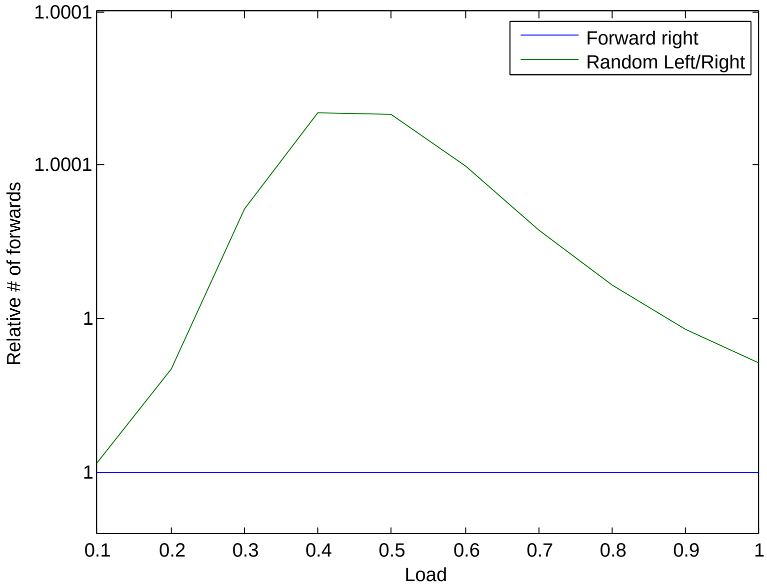


Figure 8: Validation of Random Left/Right with p=0.5

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 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.

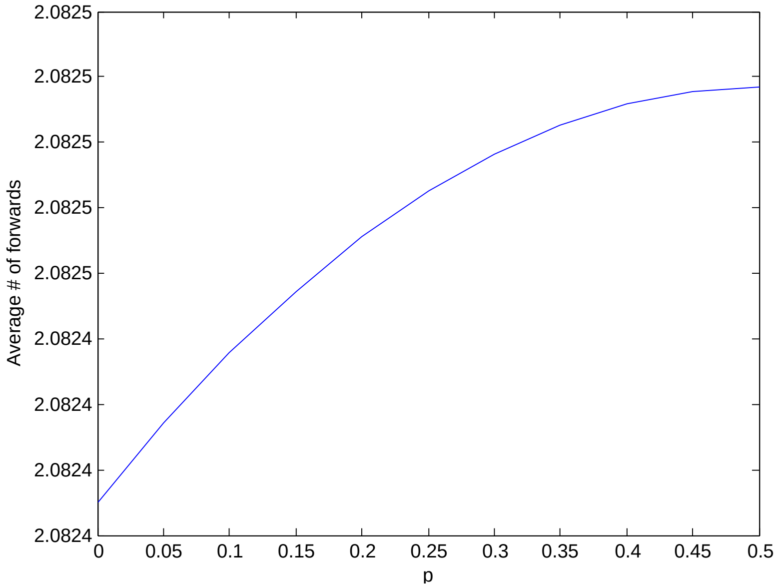


Figure 9: Performance of Random Left/Right with load= 0.8

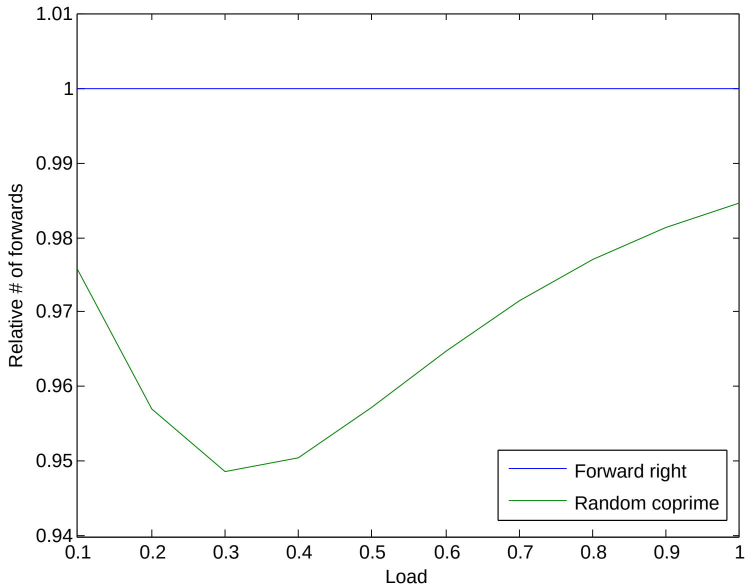


Figure 10: Validation of Coprime algorithm for $N = 10$

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 gain for $N = 12$.

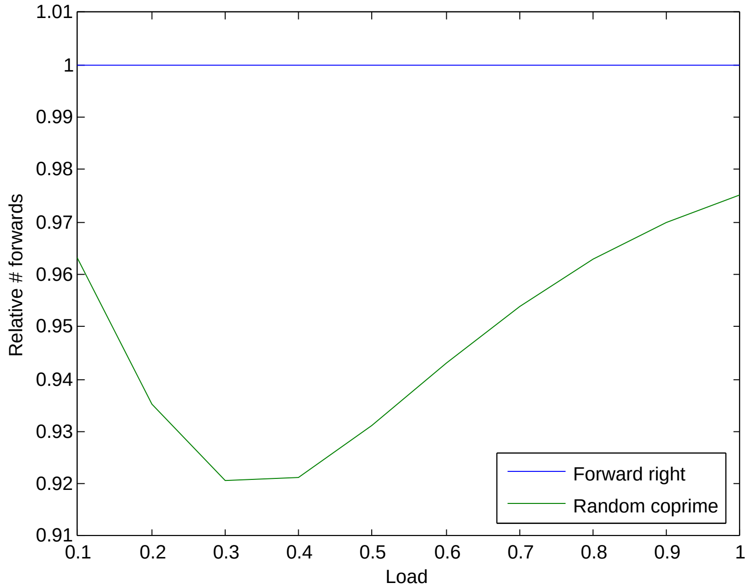


Figure 11: Validation of Coprime algorithm for $N = 11$

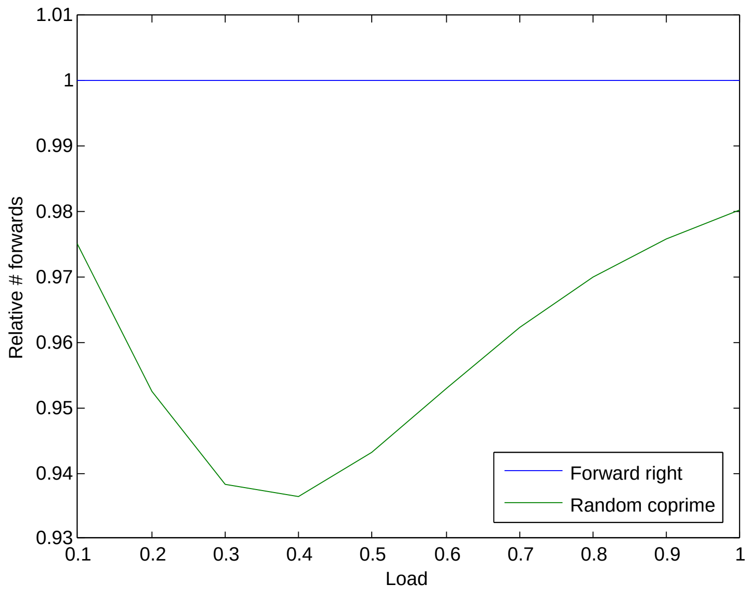


Figure 12: Validation of Coprime algorithm for $N = 12$

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.

### 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.

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

### 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:

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

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: $\frac{1}{N}\sum_{d|N}(2^{N/d} \cdot phi(d))$ with $phi(d) = d \cdot \prod_{p|d,p \text{ is prime}}(1 - \frac{1}{p})$ [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.

It seems lumping is a good technique to push the bounderies of the validation by reducing two important factors of the compution time. However, it is no

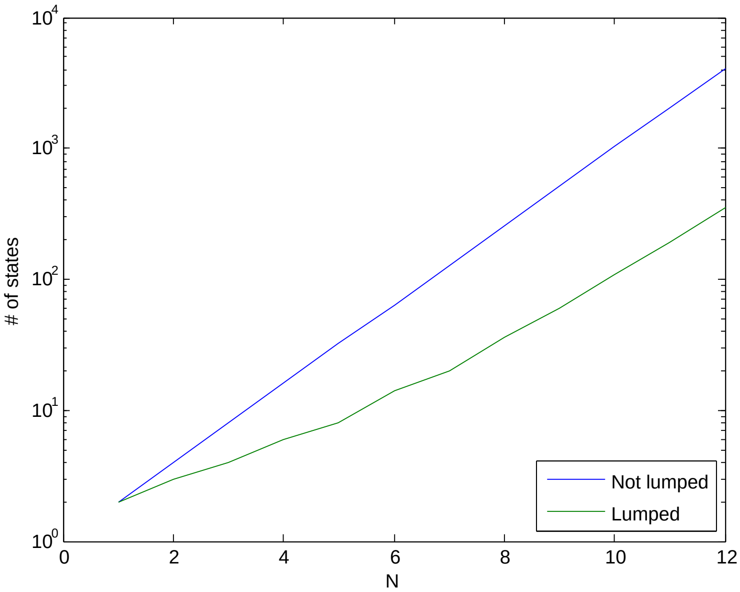


Figure 13: A

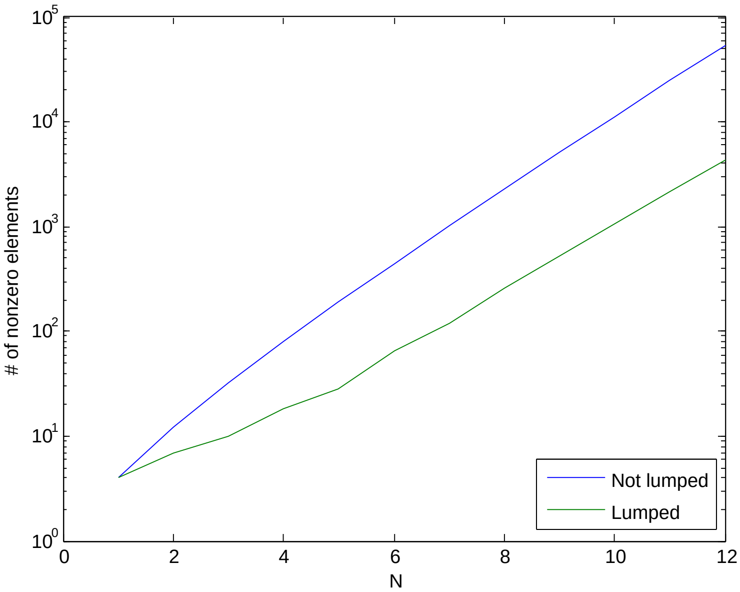


Figure 14: B

silver bullet: both the number of states and the number of nonzero elements are nonpolynomial after lumping the matrices.

### 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`.


# A Simulator source code

dit is de inhoud

# B MATLAB Numerical evaluation code

```
function [Q] = rightchain(size, rate)
%RIGHTCHAIN Generate a Markov Chain that always forwards right
%Parameters:
%       size    The size of the ring
%       rate    The rate of arrivals

    totalsize = 2^size;
    Q = sparse(totalsize, totalsize);

    BITS = zeros(1, size);

    for i=1:size
        BITS(i) = 2^(i-1);
    end

    for i=0:(totalsize-1)
        t=0;
        for b=1:size
            j=bitxor(i, BITS(b));
            if bitand(i, BITS(b))
                Q(i+1, j+1)=1;
            else
                r=rate;
                bt=b+1;
                while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (
                    b))
                    bt=bt+1;
                    r = r + rate;
                end
                Q(i+1, j+1)=r;
            end
            t=t + Q(i+1, j+1);
```

```
        end
        Q(i+1, i+1) = -t;
    end

end
```

Listing 2: rightchain.m

```
function [Q] = randswitchchain(size, rate, p)
%RANDSWITCHCHAIN Generates a Markov Chain that randomly forward left or right
%Parameters:
%       size    The size of the Markov Chain
%       rate    The rate of arrivals
%       p               The probability a job is forwarded right
%                       (Default: 0.5)

        if nargin < 3
                p=0.5;
        end

    totalsize = 2^size;
    Q = sparse(totalsize, totalsize);

    BITS = zeros(1, size);

    for i=1:size
        BITS(i) = 2^(i-1);
    end

    for i=0:(totalsize-1)
        t=0;
        for b=1:size
            j=bitxor(i, BITS(b));
            if bitand(i, BITS(b))
                Q(i+1, j+1)=1;
            else
                r=rate;
                bt=b+1;
                while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (b))
                    bt=bt+1;
                    r = r + rate*p;
                end
                bt=b-1;
                while bitand(i, BITS(mod(bt-1, size)+1)) & (bt ~= (b))
                    bt=bt-1;
                    r = r + rate*(1-p);
                end
                Q(i+1, j+1)=r;
            end
            t=t + Q(i+1, j+1);
        end
        Q(i+1, i+1) = -t;
    end

end
```

Listing 3: randswitchchain.m

```
function [ Q ] = rprimechain( size, rate )
```

```
%RPRIMECHAIN Generate a Markov Chain that chooses a random coprime and uses this as forwarding offset
%Parameters:
%       size    The size of the ring
%       rate    The arrival rate

        totalsize=2^size;
        rprimes=[];

        for i=1:(size-1)
                if gcd(size, i) == 1
                        rprimes=[rprimes i];
                end
        end

        rpcount = length(rprimes);

        %Q=zeros(totalsize);
        Q=sparse(totalsize, totalsize);

        for i=0:totalsize-1
                tot=0;
                for j=0:size-1
                        k=2^j;
                        if bitand(i,k)
                                Q(i+1, i-k+1) = 1.0;
                                tot=tot+1.0;
                        else
                                c=0;
                                for p=rprimes
                                        current=mod(j-p, size);
                                        while (bitand(i,2^current))
                                                current=mod(current-p, size);
                                                c=c+1;
                                        end
                                end
                                Q(i+1, i+k+1) = rate + c*rate/rpcount;
                                tot=tot+Q(i+1, i+k+1);
                        end
                end
                Q(i+1, i+1) = -tot;
        end

end
```

Listing 4: rprimechain.m

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

        rate = rate*size;

        Q = sparse(size+1, size+1);

        Q(1,2) = rate;
        Q(1,1) = -rate;
```

```

        Q(size+1, size) = size;
        Q(size+1, size+1) = -size;

        for i=2:size
                Q(i,i-1) = i-1;
                Q(i,i+1) = rate;
                Q(i,i) = -Q(i,i+1)-Q(i,i-1);
        end

end
```

Listing 5: runvisitedchain.m

```
function [ avg ] = avghops(Q, d)
%AVGHOPS Calculate average number of times a job is forwarded
%Parameters:
%        Q        The matrix representing a markov chain
%Optional:
%        d        Debug mode, default=1, disable debug output=0

        if nargin < 2
                d=1;
        end

        steady=full(ctmcsteadystate(Q));

        len=length(Q);
        states=log2(len);
        avg=0;
        total=0;

        for i=0:(states-1)
                c=0;
                prefix=((2^i)-1) * 2^(states-i);
                for j=0:(2^(states-i-1))-1
                        c=c+steady(prefix + j + 1);
                end
                total=total+c;
                if d
                        fprintf('%d hops:\t%f\n', i, c);
                end
                avg=avg+(c*i);
        end

        loss=steady(len);
        avg=avg/(1-loss);
        if d
                fprintf('Loss:\t%f\nTotal:\t%f\nAverage #hops:\t%f\n', loss, total + loss, avg);
        end
end
```

Listing 6: avghops.m

```
function [ avg ] = ruavghops( Q, d )
%RUAVGHOPS Calculate average number of times a job is forwarded for
    the random unvisited chain
%Parameters:
%        Q        The matrix representing a markov chain using the
    random unvisited forwarding algorithm
%Optional:
```

```
%        d        Debug mode, default=1, disable debug output=0

         if nargin < 2
                 d=1;
         end

         steady=ctmcsteadystate(Q);

         len=length(Q);

         avg = 0;
         avgp = zeros(1, len);

         for i=0:len-2
                 tmpavg = 0;
                 for h=0:i
                         c = prod(i-h+1:i) * (len-1-i) / prod(len-1-h:len-1);
                         tmpavg = tmpavg + (c * h);
                         avgp(h+1) = avgp(h+1) + (c*steady(i+1));
                 end
                 avg=avg + steady(i+1) * tmpavg;
         end

         avgp(len) = steady(len);

         loss=steady(len);
         avg=avg/(1-loss);
         if d
                 avgp
                 fprintf('Loss:\t%f\nAverage #hops:\t%f\n', loss, avg);
         end

end
```

Listing 7: ruavghops.m

```
function [ pi ] = ctmcsteadystate( Q )
%CTMCSTEADYSTATE Steady state distribution of a continious time markov chain
%Parameters:
%        Q        Matrix representing a Markov Chain
%Source: http://speed.cis.nctu.edu.tw/~ydlin/course/cn/nsd2009/Markov-chain.pdf (slide 10)

         T=Q;
         len=length(Q);
         T(:,len)=ones(len, 1);
         e=zeros(1, len);
         e(len)=1;
         pi=e*inv(T);
end
```

Listing 8: ctmcsteadystate.m

```
function [ avg ] = lumpavghops(Q)
%LUMPAVGHOPS Get the average number of times a job is forwarded when the state matrix is lumped
%Parameters:
%        Q        A lumped matrix representation of a markov Chain

```

```
        fullsize=length(Q);
        [Q S]=lump(Q);
        lumpsize=length(S);
        nodes=log2(fullsize);
        hops=zeros(1,nodes+1);

        steady=ctmcsteadystate(Q);

        hops(1)=steady(1); %zero hops
        hops(nodes+1)=steady(lumpsize); %loss
        for i=2:lumpsize-1;
                bits=ceil(log2(S(i)+1));
                hops(1)=hops(1)+(nodes-bits)/nodes*steady(i);

                for j=bits:-1:1
                        c=0;
                        while c<j && bitand(S(i), 2^(j-c-1))
                                c=c+1;
                        end
                        hops(c+1)=hops(c+1) + steady(i)/nodes;
                end
        end

        %fprintf('Sum:\t%f\n',sum(hops));

        avg=(hops(1:nodes)*[0:nodes-1]')/(1-steady(lumpsize));

end
```

Listing 9: lumpavghops.m

```
function [Ql S] = lump(Q)
%LUMP Lump a matrix representing a Markov Chain
%Parameters:
%       Q       The matrix that should be lumped
%Return:
%       Ql      The lumped matrix representation
%       S       The states that are used in the lumped matrix
%The states of the matrix Q must represent the availability of the
    the servers

        [S R C] = makestates(log2(length(Q)));

        Ql=sparse(length(S), length(S));

        [i j s] = find(Q);

        for x=1:length(i)
                Ql(R(i(x)),R(j(x)))=Ql(R(i(x)),R(j(x)))+s(x);
        end

        for x=1:length(S)
                Ql(x,:)=Ql(x,:)/C(x);
        end

end
```

Listing 10: lump.m

```
function [r, refindex, coverage] = makestates(rsize)
%Generate lumped states
%Parameters:
```

```
%       rsize           Size of the ring (or log2 of the number of states of the matrix)
%Return:
%       r               Vector of the remaining states, ordered
%       refindex        Reference index, each old state points to the new lumped state
%       coverage        How many states the lumped state with the same index represents

        powers = 2.^[0:rsize-1];

        function [v] = rotate(a, size)
                p=2^(size-1);
                v = a*2 + floor(a/p) - 2*p*floor(a/p);
        end

        function [r] = makesmallest(a)
                r=a;
                for i=1:(rsize-1)
                        a=rotate(a,rsize);
                        if a<r
                                r=a;
                        end
                end
        end

        refindex = [];
        for i=0:(2^rsize)-1
                refindex = [refindex makesmallest(i)];
        end

        function [c] = cover(a, size)
                c=1;
                a=makesmallest(a);
                b=rotate(a, size);
                while a ~= b
                        b=rotate(b, size);
                        c=c+1;
                end
        end

        function [r] = smallest(a, size)
                r=a;
                for i=1:(size-1)
                        a=rotate(a,size);
                        if a<r
                                r=a;
                        end
                end
        end

        function [v] = f(word, bits, place, size)
                if place > size
                        v = word;
                elseif bits == 0
                        v = f(word, bits, place+1, size);
                elseif place + bits > size
                        v = f(word + powers(place), bits-1, place+1, size);
                else
                        v = [f(word + powers(place), bits-1, place+1, size) f(word, bits, place+1, size)
```

```
                                    ];
                    end
          end


          function [r] = makecombs(k, n)
                    leadzeros = ceil(n/k)-1;
                    fullsize = n - leadzeros - 1;
                    r = f(0,k-1,1,fullsize)*2 + 1;
          end

          r=[0 2^(rsize)-1];
          for i=1:rsize-1
                    r = [r makecombs(i, rsize)];
          end

          s=[];
          for i=r
                    s=[s refindex(i+1)];
          end

          r=unique(s);
          refindex = arrayfun(@(x) find(r == x), refindex);

          coverage=[];
          for w=r
                    coverage=[coverage cover(w, rsize)];
          end

end
```

Listing 11: makestates.m