# Large Scale Parallel Network Simulation

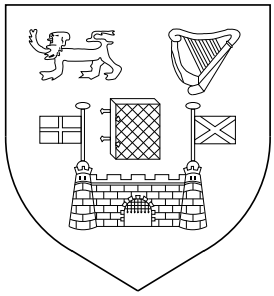
by

Eoin Lawless

A thesis submitted to
the University of Dublin
for the degree of

Doctor in Philosophy

Department of Mathematics,
University of Dublin, Trinity College



March, 2003

# Declaration

This thesis has not been submitted as an exercise for a degree at any other University. Except where otherwise stated, the work described herein has been carried out by the author alone. This thesis may be borrowed or copied upon request with the permission of the Librarian, University of Dublin, Trinity College. The copyright belongs jointly to the University of Dublin and Eoin Lawless.

Signature of Author .....

Eoin Lawless

31 March, 2003

# Acknowledgements

I would like to thank my supervisor, Professor James C. Sexton, for his inspiration and advice, and for his encouragement when it was most needed. I am grateful to Hitachi Dublin Laboratory for providing a haven of peace during the final year, thank you Martyn. My thanks also to my parents, for their confidence and support throughout my studies. Several people helped proof read sections of this thesis, I greatly appreciate their help in that onerous task. Thanks to Ken, Carlos, Carole and especially my father. My time in Trinity was greatly enriched by many friendships: thanks especially to John Loane, Kevin, Domhnall, John Mehegan and Olivia.

# Summary

Simulation is one of the primary tools used in studying computer networks. However the difficulties of simulating a network grow with its size. With the hardware resources currently available it is not feasible to simulate Internet scale networks using conventional techniques. The sheer size of the Internet prohibits its detailed simulation by, for example, discrete event simulation. The complexity of its underlying protocols has hindered the development of analytic models.

The problem has been attacked on two fronts. One approach involves increasing computational power by harnessing many processors at once, typically using the methods of parallel discrete event simulation. The other approach advocates increasing the abstraction of a model, accepting that some approximation will be introduced into the model, but seeking to minimise its effect on behaviour of interest.

Parallel discrete event simulation in network modelling has met with mixed success. The overheads of the event handling system are high, and often the method does not scale well to many processors. On the other hand, abstraction methods, if applied too freely, can reduce the accuracy of a model and even eliminate the behaviour under study.

We believe we have struck a balance in our implementation, Psim, of a network simulator. Both abstraction and parallelisation techniques are used. In particular algorithmic routing is used to avoid the cost of per node routing tables. Our parallelisation scheme eschews the complexities of a parallel discrete event handler. Instead, links that cross interprocessor boundaries manage their communication directly. This has the advantage of confining interprocessor communication to just the area in which it is needed.

The simulator models TCP traffic in high speed wired networks. We demonstrate an unrivalled degree of scalability, both in terms of number of simulated nodes, and in the number of simulating processors. A single processor is capable of simulating over one hundred thousand nodes. With sixty four processors we can simulate ten million nodes. This is an order of magnitude larger than previously recorded. Even with sixty four processors, and scaling the results to take into account worst case performance of the event handling code, the simulator exhibits a half linear speedup.

# Contents

# List of Figures

# Chapter 1

# Introduction

As computer networks have grown in size and complexity, the demand for network simulation has grown in tandem. Simulation is a tool used in diverse areas of data network research. It has been used to test protocols prior to their deployment, to help understand unexpected behaviour, to predict future requirements and to help verify theories. In recent years the Internet has become an object of interest in itself, having reached such a size and complexity that large scale phenomena, unseen in small networks, have begun to emerge. In this thesis we are concerned primarily with large scale network simulation.

Simulation is a close relative of analysis and emulation. All three are used as tools in our efforts to build and understand modern computer networks. Emulation typically involves testing a real implementation, whether a hardware device or a software protocol, in an artificial environment. Analysis attempts to describe the behaviour of a system in mathematical terms. Simulation falls between the two. It builds a model, usually simplified to a greater or lesser extent, aiming to join the accuracy of emulation with the convenience of analysis. However the boundary between these methods is often blurred, and hybrid models, combining elements of simulation and analysis have become widespread.

The explosive growth of the Internet has posed a grand challenge to network modellers; can we create analytic models or simulations that are even a fraction of the size of the Internet, while at the same time retaining enough detail to ensure their validity? The complexity of the Internet makes its analysis intractable. On one hand there has been some success in deriving expressions for aspects of its behaviour (for instance models of TCP window size [70], stationary TCP flow rates [21], or short TCP connections [69]). However no unifying description of the

complex traffic flows has been developed. But on the other hand, the sheer number of connected nodes, the profusion of protocols and number of packets transferred make a full scale, highly detailed simulation impossible.

but no unifying description of the complex traffic flows.

Although full simulation may not be feasible, we desire to push the envelope of the possible. Before describing our approach to enabling larger scale simulation, let us review some of the uses to which it might be put. One of the first applications of simulation was in the testing and prototyping of new network protocols. As the arena in which these protocols are deployed is now larger than ever, it is necessary that we model them at, or close to, this scale. A new protocol may scale well from tens to hundreds to thousands of nodes, but that is no guarantee it will scale to the millions of nodes in the Internet; large scale simulation is necessary.

The Internet is a large, complex system. The protocols that regulate its behaviour are relatively simple, but the phenomena that have appeared are altogether unexpected. These include congestion storms, route instabilities and sudden bursts of traffic. Let us consider a theory recently proposed and discuss how large scale network simulation could help verify or understand it.

In two preprints [1] [2], Abe and Suzuki propose a similarity between sudden congestion in the Internet and earthquakes. In the first preprint [2] they claim to have found an analogue of Omori's law. This is an empirical law in seismology stating that the number of aftershocks, $dN(t)$ in the period $(t, t + dt)$ after the primary earthquake at $t = 0$ are related by

$$\frac{dN(t)}{dt} \sim \frac{1}{t^p}$$

where the exponent $p$ ranges from 0.9 to 1.5. An Internet aftershock is defined to be a point in time at which the round trip time between two hosts exceeds a threshold value.

Their second preprint [1] relates the magnitude of a shock to the frequency of its occurrence, just as the Gutenberg-Richter Law does for earthquakes. This law states that the logarithm of the cumulative frequency of earthquakes with a magnitude greater than $m$ is proportional to the magnitude. The magnitude of an Internet quake is defined to be the logarithm of the round trip time between two hosts.

The authors present evidence, gathered using the *ping* utility, to support their

claims. How can large scale simulation be of use here? Abe and Suzuki make the observation that it is easier to study Internet quakes than earthquakes, and that knowledge gained from studying shocks in the Internet may be of benefit to seismology. It is easier again to measure these shocks in a simulation — not to mention less dangerous than measuring earthquakes!

In addition, a simulation allows unrivalled control over an experiment. For instance, the size and organisation of the network can be modified at will, the traffic load can be altered, shocks can be induced. This control of the parameter space could allow a researcher to determine under what conditions the proposed laws hold — how big does the network have to be for this complex behaviour to emerge, does it depend on the traffic load, or network topology? This level of experimentation is not possible in the real Internet.

## 1.1 Aims and General Approach

Our aim is simple. We wish to advance the state of the art in large scale network simulation. We are interested in:

- Networks using feedback protocols, in particular the Internet protocols, TCP/IP.
- High speed wired networks.
- Large networks — at least one hundred thousand nodes.

Our approach to the task has several distinctive features:

- Our techniques are tailored specifically for the problem. In some cases we have sacrificed generality to achieve the best performance in our area of interest.
- Our approach stresses memory efficiency and its design is tailored specifically for network simulation. To simulate a network of over a million nodes, a simulator cannot afford superfluous features that impact performance or scalability.
- We use parallel supercomputing techniques so that the largest possible models may be simulated.

We have developed a network simulator, Psim, guided by the principles above. This simulator has achieved the aims we set for it. In particular we demonstrate that it is capable of modelling networks with over ten million TCP clients. Needless to say this demands enormous resources and was made possible by using parallel computing techniques. We had access to a sixty four processor cluster for testing the code. However even on a single processor computer Psim is capable of modelling a one hundred thousand node network.

## 1.2 Contributions

Our contributions to the field of large scale network simulation are:

- Enhancements to Algorithmic Routing to increase its scalability, performance and accuracy
- A parallelisation scheme adapted to network simulation, which avoids the complexities and overhead of traditional parallel discrete event simulation.
- Offline load balancing of simulations to ensure that all processors in a cluster contribute fully to the simulation.
- A lightweight, flexible approach to simulator design.

### 1.2.1 Algorithmic Routing

Algorithmic Routing (AR) [46] [47] is a technique that approximates shortest path routing and removes the need for routing tables at each node in the network. This is an immense advantage to the network simulator, as maintaining routing tables can consume large amounts of memory. For instance a flat routing table has memory requirements that scale $\mathcal{O}(N^2)$, where $N$ is the number of network nodes. A balanced hierarchical scheme scales roughly with $\mathcal{O}(N \log N)$, where there are $\log N$ levels of hierarchy. A two level hierarchy such as the Internet scales with at best $\mathcal{O}(2N\sqrt{N})$ [58]. AR scales linearly with $\mathcal{O}(N)$.

AR maps a network graph onto a tree and uses a simple algorithm to calculate a path between two nodes. However, AR has problems that until now have restricted its use in network simulation. The worst of these issues include route lengthening and concentration of all traffic onto $N-1$ links. They arise because most network

graphs are not trees, and hence when mapped to a tree certain links are left unused. There have been some proposals to mitigate these effects, for instance by maintaining a separate tree for important traffic sources, but these are not altogether satisfactory [47].

We propose several enhancements to AR: direct AR, fixed cost routing, route improvement and multitree AR. Direct AR makes the tree mapping implicit in the network graph structure. This allows a network simulator to route packets without any additional memory usage. In some situations it may not be practical to arrange the network graph in this manner, but in this case the parent of a node can be explicitly stored. Direct AR also has the benfit of reducing the number of operations needed to perform packet routing, increasing its performance.

We propose a method for computing the routing algorithm in fixed time. Previously it had taken $\mathcal{O}(\log N)$ time to compute. This new scheme makes use of a double numbering system for nodes.

We devise a method for improving the quality of routes generated by AR. This heuristic takes a tree and iteratively modifies it so as to reduce the total distance between nodes.

Our final enhancement is perhaps the most important. We describe a procedure for generating multiple routing trees in such a way as to reduce traffic concentration at the busiest nodes, and increase the utilisation of links that are ignored in single tree AR.

These enhancements collectively make AR a viable tool for use in large scale network simulation.

### 1.2.2 Parallelisation

Parallel computing holds the promise of enabling simulations far larger than those possible on a single computer. In theory, by combining the power of $n$ processors a simulation should be able to run up to $n$ times faster. However the extra memory that a parallel computer offers is almost more significant. Many of today's computers — especially the low cost, high performance $x86$ class — are limited in the amount of memory available to a single processor. This limit is often quite low — 4 to $8GB$ for a $x86$ processor. By running a simulation in parallel one can model far larger networks than would otherwise be possible.

Parallel discrete event simulation (PDES) has been applied by several groups to the problem of network modelling. In PDES the nodes comprising the network

model are partitioned between the processors of the parallel cluster.

PDES is not easy to implement and harder again to implement efficiently. Synchronising event timelines between processors introduces a considerable overhead. The event handling code we use is entirely sequential in nature. Each processor maintains an event list but events are not synchronised in the event handling code, rather we employ a special *bridge* device to transfer packets between nodes on different processors, and provide synchronisation. This provides a clean interface for parallelisation. The extra complexity introduced by parallelisation is present only at the boundary between partitions. This also reduces the complexity of the event list handling code, which is critical to performance.

### 1.2.3 Load Balancing

In order to obtain the maximum efficiency from a parallel simulation, all processors must contribute equally to the task. This requires that the network is partitioned in such a way that each partition generates an equal number of events. It is not sufficient to divide the network so that each partition has an equal number of nodes, since a node in the core of a network will generate far more events than one on the periphery.

With increasing numbers of processors in a parallel cluster, a good partitioning scheme increases in importance. A higher number of processors means a correspondingly smaller partition size. With small partitions, any variance in the work required to simulate a partition has a far greater impact on the simulation efficiency.

We have implemented a form of offline load balancing in our code in order to ensure a good partitioning. By conducting a trial run and recording the number of events generated by each node, it is possible to perform a weighted partitioning of the network. This weighted partition can be used in a full simulation in order to achieve maximum efficiency.

## 1.3 Organisation

We continue, in Chapter 2, with an overview of the Internet structure and protocols. We describe the traditional methods of network simulation and analysis. With this foundation we can discuss new approaches to network modelling: ab-

straction, parallelisation, fluid simulation and hybrid techniques. The chapter is concluded with a survey of the current state of the art network simulators.

The following two chapters contain our contributions to large scale network simulation. The first, Chapter 3, on Algorithmic Routing, details in depth the enhancements we introduced above. We extensively test the new techniques, especially to ascertain their fidelity to shortest path routing. The final example of the chapter uses a 2.3 million node scan of the Internet as the test case for the viability of AR.

Chapter 4 describes our approach to implementing a highly parallel network simulator. We discuss the general principles that guided its design and the techniques used to parallelise it. We include detailed experimental analysis of its performance and test cases that demonstrate its scalability.

We conclude with Chapter 5 and summarise our findings.

# Chapter 2

# Literature Survey

This thesis is concerned with large scale network simulation. Interest in the subject has grown in parallel with the growth of the Internet. However, the origins of many of the simulation techniques and analytical tools used by researchers in the field predate the Internet, and even digital computers.

This chapter introduces the field of large scale network simulation. In order to give a comprehensive view of the field we discuss some material not directly connected to our own contributions. We begin by briefly describing the structure of the Internet, and the protocols that lie at its core. We move on in Section 2.2 to a discussion of the two principal methods used to understand and model the behaviour of a network: analysis and simulation. Pure analysis and pure simulation lie at opposing ends of the spectrum. Some properties of the Internet make it analytically intractable, while its sheer size makes it un-amenable to detailed, faithful, simulation.

To surmount these problems, network modellers have pursued several lines of research. These include parallel simulation, greater levels of abstraction, fluid simulation and new analytic techniques.

Parallel computing has been used to increase the speed and size of simulations by harnessing the resources of many processors linked together. Section 2.4 outlines new developments in parallelising simulations.

A network can be modelled by the loosest approximations or in the minutest detail. The level of fidelity of course depends on the type of network behaviour we wish to study. It is possible, indeed necessary, to abstract aspects of network behaviour. The aim is to reduce unnecessary complexity without sacrificing the essential properties that give rise to the richness of behaviour we observe in the In-

ternet. The tradeoff between the approximations made and the errors introduced by a particular model must be understood. Recent research to develop more abstract simulations and to understand the effects of approximation and abstraction is described in Section 2.5.

Fluid simulation, Section 2.6, treats data traffic as a continuous flow of information rather than as discrete packets. This has the potential to reduce computational and storage requirements for simulations.

Several groups have attempted to take advantage of the speed of analytic models and the accuracy of simulation by creating hybrid models combining elements of both. These are surveyed in Section 2.7.

Fluctuations in network traffic are expected. However rarely occurring events can sometimes have a disproportionate effect on the state of a network. The infrequency of their occurrence leads to difficulties in simulating their consequences. There are several techniques that seek to efficiently model these events. We briefly survey them in Section 2.8.

Although the protocols that regulate the Internet are formally specified, these alone do not provide enough information to model the Internet. They dictate standards on how computers on the Internet communicate, but do not proscribe when two computers communicate, or what is communicated. Nor is any particular physical topology imposed. We must turn to studies and surveys of the Internet to discover the types of network topology and traffic patterns to expect. Section 2.9 summarises some recent findings.

The chapter concludes with a discussion of the current state of the art in network simulators.

## 2.1 The Internet in a Nutshell

The physical structure of the Internet consists of routers, hosts, links and sundry other network devices. Hosts connected to the Internet communicate using a shared family of protocols: in particular the Internet Protocol (IP). Other protocols are layered on top of this base to add extra capabilites. The Transmission Control Protocol (TCP), for instance, adds reliability and flow control. Information is transferred between hosts by discretising it into *packets*. Packets are sent through the links connecting two hosts, and reassembled at the destination.

It is the presence of protocols such as TCP, which employ feedback to reg-

ulate their behaviour, that adds substantial challenge to the task of network modelling. Switched telephone networks, the predecessors of today's computer networks, lacked many of the complexities introduced by flow control, in particular.

### 2.1.1 The Transmission Control Protocol (TCP)

TCP is layered above IP, which provides a way for TCP to send and receive variable length data packets. However IP does not guarantee reliability, or have any notion of session or connection. Building on the lower level IP service, TCP provides basic data transfer, reliability, flow control, connections, multiplexing and security. TCP is described in [50], but has been gradually enhanced and extended since [5]. We summarise its behaviour and operation here.

TCP can transfer a continuous stream of data between two hosts by splitting the data into segments and passing them to the IP level for transmission through the network. Since IP does not guarantee the correct delivery of packets, TCP must have mechanisms for dealing with lost, duplicated, delayed or damaged packets. To this end each packet transmitted includes a sequence number and a checksum. The sequence numbers start from a randomly chosen initial sequence number (IST), and each data byte has an associated sequence number such that

$$\mathrm{SEQ}(n) = \mathrm{IST} + n$$

where $\mathrm{SEQ}(n)$ is the sequence number of the $n^{\mathrm{th}}$ data byte.

The sequence number ensures that packets can be correctly ordered and duplicates detected. In addition, the receiver must acknowledge data by sending an acknowledgement (ACK) back to the sender. At the receiver, the checksum allows damaged packets to be discarded.

In TCP the receiver controls the rate at which the sender can transmit data. With every ACK that the receiver sends back to the sender, it includes a *window* indicating a range of sequence numbers beyond that of the last data byte received. The sender may not send any packets with data whose sequence number is beyond this window. The frequency with which ACKs are generated, and the manner in which the window is calculated has evolved since the introduction of TCP. Several algorithms for controlling the TCP window size have been developed and are in common use [5].

TCP allows for multiplexing of traffic (multiple independent connections between two hosts) by the use of port numbers. The combination of a port number and a host address is known as a *socket* and a pair of sockets uniquely identifies a connection. The port number is stored in the header field of a TCP packet. For convenience, certain port numbers are commonly reserved for common applications; for example a *telnet* server usually uses port 23. A connection consists of a socket pair, window size, sequence numbers and a few other state variables (these may vary between implementations)

## 2.2 Traditional Network Analysis and Simulation

Network modelling naturally falls into two primary categories. Analysis typically builds a model based on the statistical properties of network traffic and seeks to predict the bulk behaviour of a network. It is exemplified by queueing theory. Queueing theory [12] studies the behaviour of systems of servers and queues. Jobs (or packets in network modelling) arrive at a system, wait in a queue and are eventually processed by a server. The theory has been applied to phone switches, production lines, supermarket queues and baggage handling. It attempts to predict properties such as queue size, packet delay and interarrival distributions given a description of the input to a system. Certain types of arrival and service time distributions are well described by queueing theory. If both the interarrival time of packets at a server and the service time are exponentially distributed, then it is possible to predict the statistical properties of the queue occupation and output traffic in many cases.

Unfortunately the traffic distributions observed in the Internet are not so easily characterised [77]. This has led to the use of simulation as a tool in understanding and predicting network behaviour.

### 2.2.1 Discrete Event Simulation

Discrete event simulation (DES) can be described using a mathematical formalism called Discrete Event Systems Specification (DEVS), developed by Zeigler [99]. It is a two level description, consisting of *atomic DEVS* and *coupled DEVS*. An *atomic DEVS* models a system as a sequence of transitions between states.

Furthermore it describes its reaction to external input events and how it generates output events. An *atomic DEVS* model is defined as:

$$atomicDEVS \equiv (S, ta, \delta_{\text{int}}, X, \delta_{\text{ext}}, Y, \lambda).$$

$S$ is the set of possible sequential states the system may take. $ta : S \rightarrow \Re_{0,\infty}$ is the time advance function; that is, the length of time the system remains in one state before making a transition to the next state. $\delta_{\text{int}} : S \rightarrow S$ is the internal transition function, which models the transition from one state to the next. $X$ is the set of possible input events. The system's reaction to external inputs is described by $\delta_{\text{ext}} : Q \times X \rightarrow S$, where $Q$ is the state of the system, taking into account the elapsed time, $e$, since the last transition: $Q = \{(s, e) | s \in S, 0 \leq e \leq ta(s)\}$. The set of possible outputs is given by $Y$. The output function $\lambda$ is defined as $\lambda : S \rightarrow Y \cup \{\emptyset\}$, where $\emptyset$ is the null–event. Output events are only generated at the same time as an internal transition, at which point the state before the transition is used as input. At other times the null–event $\emptyset$ is output.

Consider the following model of a network buffer in the *atomic DEVS* formalism. The buffer can hold a finite number of packets, $N$. The state of the system is represented by the set $S = \{\text{queue} = \{0, 1, \ldots, N\}, \text{status} = \{wait, send\}\}$. There are two input events, one representing a packet arrival, the other requesting that a packet be output from the buffer (if possible). The buffer remains in a constant state until either an *arrival* or *ready* event is input. On receiving an *arrival* event the queue size is increased (up to a maximum of $N$). On receiving a *ready* input the status is changed to *send*. If the status is *send* and there are one or more packets in the queue, then an internal transition occurs. This transition changes the status of the buffer to *wait* and removes a packet from the queue. At this point an output event occurs (*departure*).

More formally:

$$
\begin{aligned}
X &= \{arrival, ready\} \\
Y &= \{departure\} \\
S &= \{\{(0, 1, \ldots, N)\}, \{send, wait\}\} \\
\delta_{\text{ext}}(((n < N, s \in \{wait, send\}), e), arrival) &= ((n+1, s), e) \\
\delta_{\text{ext}}(((n = N, s \in \{wait, send\}), e), arrival) &= ((n, s), e) \\
\delta_{\text{int}}((n \neq 0, send), e) &= ((n-1, wait), e) \\
\lambda((n \neq 0, send), e) &= departure \\
ta((n \neq 0, send)) &= 0 \\
ta((0 \leq n \leq N, wait)) &= \infty.
\end{aligned}
$$

A *coupled DEVS* consists of a network of coupled components. Each component is itself an *atomic DEVS*. For example a device in a TCP/IP network might be represented by a buffer (such as the one above) and a processor that takes packets from the buffer and routes them. The processor could be an *atomic DEVS*, or might be several *atomic DEVS* coupled together.

In a practical discrete event simulation, a *timeline* of events is created. These events are sorted in time order. The computer processes these events one by one. Each event represents an occurrence in the model network. In processing one event, several others may be created. For example, an event representing a user making a web page request will generate events to send packets to a web server; each time a packet is sent over a link, an event is created to schedule the future arrival of the packet at the destination host.

This method of simulation allows an almost unlimited degree of detail in a model. It comes with a price though. The computational cost of a simulation rises with increased simulation detail.

## 2.3 Analytic Models of Network Protocols

The complexity of the protocols used to regulate network traffic has increased the difficulty of applying traditional queueing theory to network simulation [77] and driven the focus of large scale network modelling from analysis to simulation.

However, progress has been made in developing analytical models of TCP (and other) traffic.

TCP responds to network congestion by changing the rate at which it injects packets into the network. The window size reflects the amount of data that can be transmitted without waiting for acknowledgement. The original TCP specification [50] did not impose a scheme for updating the window size. Enhancements such as slow start, congestion avoidance, fast recovery and fast retransmit were later introduced [5].

Misra et al. [70] have modelled window size behaviour as a stochastic differential equation. They treat packet loss as a flow of events arriving at a source, rather than as packets leaving the source with a loss probability. The technique enables fluid analysis of TCP and TCP–like congestion control schemes. Cassetti and Meo in [21] develop a model for the stationary behaviour of TCP flows. Their method allows for the estimation of delay and packet loss for a single TCP source. They use queueing theory to analyse several superimposed sources. Mellia et al. [69] have developed a model for short TCP connections, those that remain in the slow start phase. Such connections account for the majority of Internet connections, if not for the majority of packets transferred.

The work above is a small representative example of current research into analytic models of data traffic.

## 2.4 Parallel Computing

It is not possible to simulate the Internet in detail on a single computer. This difficulty has motivated the two principle approaches to large scale network simulation: parallelisation and increased abstraction. We will discuss the latter topic in the next section.

The proposition advanced by advocates of parallel computing is simple: if one computer can complete a task in ten days, then ten computers working together should be able to complete the same task in one day, or alternatively a task ten times larger in the same time. The reality, of course, is more complex. Whether or not ten computers can simultaneously work on the same task with full efficiency depends very much on the nature of the task. Parallel efficiency, $E(N, P)$ for a

problem of size $N$ on $P$ nodes is formally defined [59] as

$$\frac{1}{P}\frac{T_{\text{seq}}}{T(N,P)}$$

where $T(N,P)$ is the runtime of the parallel algorithm, and $T_{\text{seq}}$ is the runtime of the best sequential algorithm. Typically $E(N,P)$ ranges from 0 to 1, with a high value being better. Unfortunately it is not easy to efficiently parallelise network simulation.

Before discussing the techniques for parallelising a network simulation, let us review the the types of parallel computer generally available.

A parallel computer has two or more CPUs (processors). There are two primary classes: shared memory and distributed memory parallel computers. In a shared memory computer, all processors have equal access to a central store of random access memory. Two and four processor shared memory computers are quite common. Shared memory computers with more than four processors are difficult to construct and are considerably more expensive.

In a distributed memory computer every processor has its own store of random access memory. Indeed, distributed memory computers are very often clusters of single processor computers linked by a communication backbone such as Myrinet, or by a standard network such as Ethernet.

Hybrids of the above two classes also exist. For instance some supercomputers consist of clusters of nodes. Each node is composed of several processors. Memory is distributed between the nodes, but within a node it is shared by the processors.

The first task in running a simulation in parallel is to devise a method to split the work among the available processors. One approach is Parallel Independent Replication Simulation (PIRS) [63]. Every processor runs its own copy of the simulation. This is useful if the modeller wishes to examine statistical properties. Each simulation uses a different initial seed for random variables. The parallel simulation can explore a greater proportion of the model state space in a given time than can single processor simulation.

However, those who wish to model large networks turn more often to Parallel Discrete Event Simulation (PDES) [35]. In this method the network model is partitioned between processors. The advantage is that one can model larger networks since an individual processor needs only model a small portion of the network.

In PDES, the model is divided into a number of partitions, one per processor.

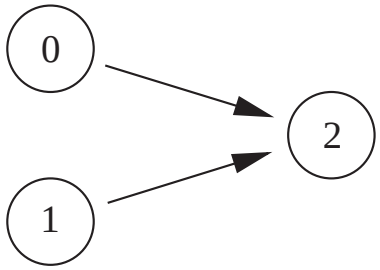



*Figure 2.1: Simple network example.*

Each partition has an associated event list, containing events for the model entities in that partition. Interactions between different partitions are managed by the exchange of messages between processors. Each processor executes events on its local eventlist, with messages from other processors periodically triggering the insertion of events representing interactions with other partitions. The difficulty of creating a PDES model arises in ensuring that these events are processed in the correct order. Approaches to parallel simulation, and PDES in particular, are surveyed in [73].

Consider the simple network of three nodes shown in Figure 2.1. Each node is modelled on a different processor. On receiving a packet arrival event a node schedules a packet departure event for one time unit later. Nodes zero and one receive arrival events from an external source. Packet arrivals on node two are triggered by departure events on nodes zero and one. Consider the scenario illustrated by Figure 2.2. Node zero receives an arrival event, $e_1$ at simulated time $t = 1$, causing it to schedule a departure event $e_2$ at simulated time $t = 2$. Node two similarly receives an arrival event $e_3$ at $t = 3$ and schedules at departure event $e_4$ at $t = 4$. On node two, event $e_2$ causes an arrival event $e_5$ at $t = 2$ and event $e_4$ causes an arrival $e_7$ at $t = 4$. These arrivals cause departures $e_6$ at $t = 3$ and $e_8$ at $t = 5$.

Next, consider the order in which the three processors generate and receive these events in real time, Figure 2.3. Suppose that processor zero is considerable slower than processors one and two. Events $e_1$ and $e_2$ are processed much later in real time than events $e_3$ and $e_4$. This means that processor two receives input arrival event $e_7$ before it receives event $e_5$, even though $e_5$ is scheduled to occur first in simulated time. If the events are to be handled by processor two in the correct order, it must wait until processor zero has handled departure event $e_2$

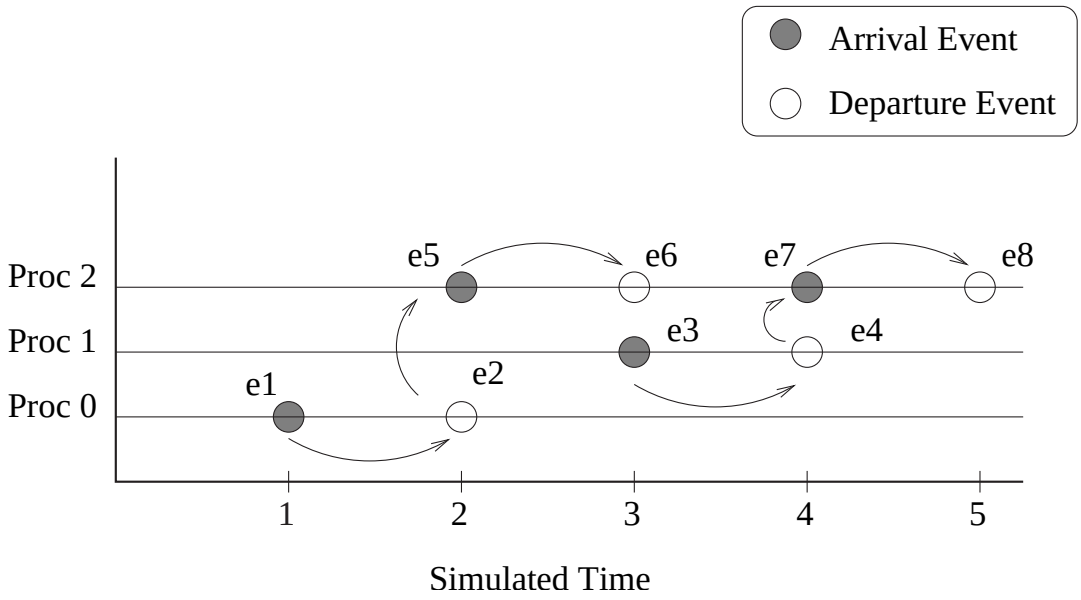



*Figure 2.2: Events in simulated time.*

causing arrival event $e_5$. How is processor two to know that it must wait?

This problem, known as the *causality constraint*, has led to two different approaches to PDES. Some method of synchronising the processors in a PDES simulation is needed. In conservative parallel simulation, a processor refrains from processing an event $e_1$ at simulated time $t_1$ until it is certain that no event $e_2$ at time $t_2 < t_1$ will arrive from another processor. This can be achieved by the exchange of *null messages*. A null message is a message with an associated timestamp sent from one processor to another saying that the sender guarantees not to send any events with a simulated time less than that of the null message timestamp. For instance at time $t = 0$ node zero has not received an arrival event. Therefore it knows that it will not schedule a departure event until $t \geq 1$. Hence it can send a null message with a timestamp of 1 to processor two. Node one can also send such a null message. Processor two will not process any events beyond $t = 0$ until it has received both of these null messages. Nodes one and two send null messages again at $t = 1$ with a timestamp of 2. However at $t = 2$, only node one sends a null message with a timestamp of 3, while node zero processes its departure event $e_2$, which causes input arrival event $e_5$ at node two. Since processor two knows that processor one will not cause any arrival events before $t = 3$, it knows that it can safely process $e_5$.

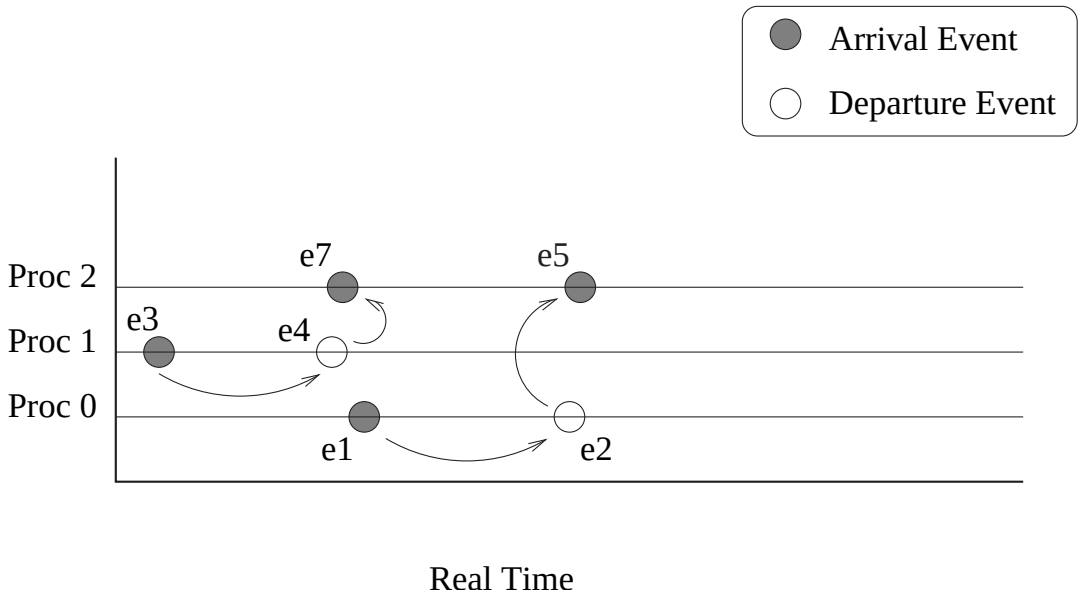



*Figure 2.3: Event generation in real time.*

In optimistic synchronisation, processors are allowed to execute events, without the guarantee that events with an earlier simulated time will not arrive later in real time. However, the simulator must be able to undo any changes made to the model state, roll back simulation time to the earlier timestamp and continue from there.

Distributed and shared memory simulators call for different approaches to partitioning a network. In a shared memory model, the entire model lies in main memory. In contrast, in a distributed memory model each processor has access to only a small part of the network, stored in its local memory. It should be noted that it is possible to run a distributed memory model on a shared memory computer and vice versa. This does not make optimal use of the hardware, but commonly occurs when a distributed memory model is run on a cluster of shared memory computers. Running a shared memory model on a distributed memory architecture is less efficient again, as access to shared memory must be simulated by message passing between processors.

Designers of distributed memory parallel simulators have run into many common problems: in particular the issue of the global name space [65] [85]. Suppose that a traffic generator wishes to send data to a random traffic sink (a network node that accepts packets without forwarding them on). In a shared memory

model, the processor responsible for the generator also has access to the entire network topology and can chose a sink at random. In a distributed model, the processor responsible for the generator may know only about the traffic sinks it models itself. How does it choose a sink at random, when it only knows about part of the network? One solution involves keeping a full copy of the network on each processor, though each processor remains responsible for only its own partition. This however negates the primary advantage of using a distributed memory computer: the ability to simulate larger models due to the partitioning of the network.

It should be noted that the techniques used to partition a model for a distributed memory computer will work on a shared memory computer, but the converse does not hold. Generally simulators are tailored for one approach or the other. This has raised problems for several groups seeking to move from shared memory computers to distributed memory computers. Model initialisation, for instance, is more difficult to program in a distributed memory environment. In a shared memory model, the network configuration is loaded once, and all processors have access to it. In a distributed model there are several possible approaches. In the simplest, each processor reads in the full model, the network is partitioned and each processor discards those parts not relevant to itself. This has the drawback that the total model size is limited to the largest model a single processor can load. Another approach might be to prepartition the network, and have each processor load its part of the prepartitioned network. A different problem arises when two network devices on different processors must be connected. How does one describe a linkage when only one end of the link is present? While most of these issues are purely technical, they are symptomatic of the greater difficulty of programming a distributed memory simulator. Liu [65], describes some of the problems encountered in porting the SSF simulator [26] from a shared to a distributed memory environment.

The attraction of distributed memory computers is that they can often be constructed using low cost, commodity parts. Therefore they frequently possess more processors and more memory than an equivalently priced shared memory computer.

One criticism levelled at parallel computing is that at best a parallel computer offers a linear speedup. This is true in most cases. (An example where this might not be true is when the parallel model resides entirely in main memory, but the

sequential model needs to use swap, with a resulting slowdown in operation). The argument is that the Internet is growing exponentially, and a linear improvement in size or speed of simulation is not sufficient — increased abstraction is called for. However the two approaches are complementary. Increased abstraction is undoubtedly needed to model large networks, but at any given time, parallel simulations — perhaps also using abstraction techniques — will be able to model the largest networks. Some conflicting viewpoints are presented in [82], [26] and [46]. Cowie et al. [26] argue that Moore's Law (a rule of thumb that states that the maximum number of transistors on a single chip doubles every eighteen months), will ensure that we will eventually be able to model Internet scale networks. Huang [46] emphasises the importance of abstraction, since parallelisation offers only a linear increase in model size. Riley et al. demonstrate that running an Internet size model in parallel using their *pdns* [85] simulator is not possible in the foreseeable future.

Section 2.10 surveys several large scale parallel network simulation packages, and briefly describes their approach to parallelising a simulation.

Chapter 4 discusses our implementation of a parallel network simulator. We chose to implement our simulator as a parallel rather than sequential simulator as we believe that the extra memory and processing power of parallel computers is necessary to simulate the very largest networks. Some level of abstraction, discussed in the next section, is also required, but not sufficient.

## 2.5 Increased Abstraction

In modelling any system we hope to capture the essential details of the system, while ignoring those that do not affect the behaviour of interest. An almost universal abstraction in network simulation, for instance, is to ignore the payload of the simulated data packets, and retain, at most, the packet headers.

Huang, in her thesis [46], studied abstraction techniques. Her work is unique in that rather than proposing single techniques on an ad hoc basis, she systematically compares existing techniques (and proposes several new ones) to determine their efficiency and range of applicability. Many of these techniques are now part of the *ns* simulator [8], and Huang's work allows a network modeller to choose an appropriate abstraction so as to combine the fastest simulation with the required level of accuracy in the areas of interest.

The abstraction techniques discussed are: end-to-end packet delivery, Finite State Automata (FSA) TCP models and Algorithmic Routing (AR) [46].

End-to-end packet delivery is a modification to a discrete event network simulation. Packets are delivered directly from source to destination, with a delay corresponding to the link delays on the intermediate links. This avoids the detailed hop by hop simulation of the packets through the network and greatly increases performance. It is suitable for networks where there is little congestion, or for simulations where accurate packet delay is not needed. In congested networks this technique does not reflect queueing times at nodes.

FSA TCP is a simplified model of the TCP protocol. A TCP state is represented by a node in a directed graph. A TCP flow in the model contains a pointer to a node in this FSA diagram which represents its current state, rather than maintaining the state variables as in a real TCP stack. States are linked by directed edges, which represent allowed transitions. A TCP flow moves from state to state as packets are acknowledged, window size is increased and packets are dropped, etc. There are slightly different state diagrams for the various flavours of TCP. The simplified FSA TCP model can be used in situations where there is a low rate of packet loss, and the connections are short. A full model must be used in other situations.

AR is a technique for eliminating full scale routing tables or protocols. It works by first building a $k$-nary tree from the network topology, breaking links if necessary to remove cycles. Given this tree, there is a simple algorithm to determine which link a packet at a router need take to reach its destination. If the network contains no cycles, the path determined by the algorithm is the shortest path. If there are cycles in the graph, then there are some paths that will be longer than necessary. This technique can be used in simulations where routing details are not vital.

*NIx–Vectors* is another approach to efficiently simulating routing [84] [86]. Rather than maintain routing tables, every packet contains an extra field, containing a Neighbour Index Vector, or *NIx–Vector*.

The basis of the *NIx–Vectors* routing technique is the observation that at any router, the act of routing simply involves choosing one element from an ordered set of directly connected neighbours. If there are $N$ neighbours, then a particular routing choice can be recorded in $\lceil \log_2 N \rceil$ bits. An entire route can be represented as the concatenation of these choices, starting at the source node and ending at

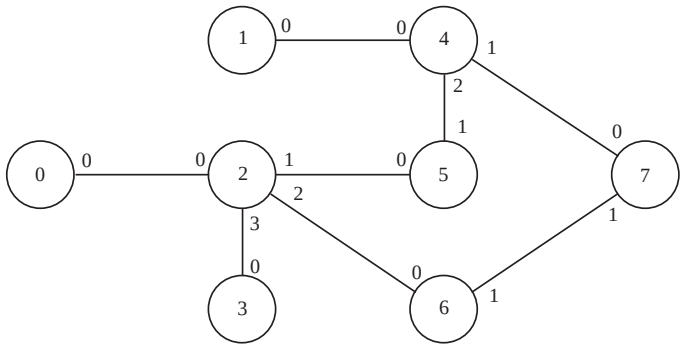



*Figure 2.4: Simple network example*

the penultimate node in the route. Riley et al. term this a *NIx–Vector*.

In order to route a packet using a *NIx–Vector*, a node simply extracts the next $\lceil \log_2 N \rceil$ bits from the vector and uses this value as an index into its ordered set of connected neighbours.

As an example consider the network shown in Figure 2.4. Each node is numbered, and each link from a node is also numbered (sequentially from zero). By inspection it is clear that the shortest path between node zero and node seven is 0 – 2 – 6 – 7. The table in Figure 2.5 shows how a *NIx–Vector* for this route can be constructed. The column labelled *Node* indicates the node at which the routing choice is made. The column *Neighbours* gives the number, $N$, of directly connected neighbours at that node, and the column *Bits* is the number of binary bits needed to record any number from 0 to $N - 1$. The column labelled *Link* indicates the neighbour index of the link to the next node in the route. The final column, NIx–Vector is the concatenation of the individual routing choices. In an actual implementation of *NIx–Vectors* in the *ns* simulator, the minimum size of a *NIx–Vector* was 96 bytes, consisting of two 32 bit units storing the current and maximum vector lengths, and a minimum of 32 bits for the vector itself [84].

| Hop | Node | Neighbours | Bits | Link | *NIx–Vector* (binary) |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 1 | 0 | 0 |
| 1 | 2 | 4 | 2 | 2 | 0 10 |
| 2 | 6 | 2 | 2 | 1 | 0 10 01 |

*Figure 2.5:* NIx–Vectors *example*

Both *NIx–Vectors* and AR address a serious problem in large scale network sim-

ulation, namely the growth of routing tables. A naive implementation of routing, for instance, where each node maintains an entry for the next hop to every other node, will use memory proportional to $N^2$ for a $N$ node network. A method such as AR or *NIx–Vectors* is essential to allow network simulation scale to Internet size models.

We believe that AR allows for greater scalability than *NIx–Vectors*. In the *NIx–Vectors* technique each packet must contain a 96 bit *NIx–Vector*. In AR, using the modifications introduced in Chapter 3, $N$ 32 bit values for a $N$ node network are sufficient. Thus, if there are more than $N$ packets active in a simulation, which is almost certainly the case in any realistic simulation, then AR is more memory efficient then *NIx–Vectors*. The lesser memory usage of AR comes at a cost in computational complexity. Each route lookup in AR requires $\mathcal{O}(\log N)$ time. However the creation of a *NIx–Vector* for a route between two nodes requires $\mathcal{O}(N + E)$ time (using a breadth first search of the network topology), where $E$ is the number of links in the network. Consider a traffic flow between two nodes that transmits $p$ packets over $n$ links. Then the time taken to route the packets using AR is $\mathcal{O}(np \log N)$, and using *NIx–Vectors* is $\mathcal{O}(N + E)$. Now, in the Internet, the vast majority of flows carry fewer than twenty packets (most of these short connections are due to webserver and domain name server requests) [24] and the default maximum path length is sixty four [81]. This indicates that for large networks AR should also be more computationally efficient than *NIx–Vectors*. While it is possible to cache *NIx–Vectors* so as to eliminate the need to calculate a *NIx–Vector* more than once, this increases its memory requirements further, especially if there are many different short lived flows.

Unfortunately, AR has one flaw that is not present in *NIx–Vectors*: the routes generated by AR all lie on a tree superimposed on the network topology graph. This means that routes are not always shortest path routes, and in addition some links are completely unused, resulting in congestion on the remaining links. However, due to the better scalability and performance offered by AR we decided to investigate methods to improve the quality of the routes it generates. This work is discussed in Chapter 3.

Another abstraction, first introduced by Jain [53] and developed by Ahn and Danzig [3] involves taking advantage of *packet trains*. Jain noticed that packets from the same flow are often clumped together in *trains* even after traversing many links in a network. Ahn and Danzig developed ways of processing an entire train

of packets in a switch, rather than processing them individually. If the trains are long enough this can result in a considerable saving of memory and computational time. Unfortunately it has been observed that in larger, modern, networks the size of packet trains has reduced due to greater mixing of flows, and so the potential reward of exploiting packet trains is correspondingly lower. The idea has been extended in fluid simulation.

## 2.6 Fluid Simulation

Fluid simulation [56] [60] represents the flow of data through a network as a fluid rather than as discrete packets. Several groups have explored the potential of this idea.

A fluid simulator keeps track of the flow rate at each point in the network. Every time the flow rate of a source changes, this change is propagated onwards to other nodes. Typically sources are represented by Markov-Modulated processes [64]. Sources are either *on* or *off* at a particular time. A source emits packets when it is *on*. The *on* and *off* periods are both Poisson distributed, as is the packet interdeparture time. In a fluid representation, fluid is emitted at a constant rate when the source is *on* and not emitted otherwise.

Every time the rate of fluid arrival at a queue changes, the departure rate must be recalculated. (The update may not be immediate, as fluid already in the queue continues to depart at the old rate).

Studies on the accuracy of the method have found that it agrees quite closely with packet level simulations [74] [97].

Fluid simulation of small networks has also generally been found to have far better performance than packet level simulations. Nicol et al., using the SSF simulation package for both fluid and packet level simulation, observed a speedup of three orders of magnitude [74].

Unfortunately, with large networks the *ripple effect* severely degrades performance. Each time the input flow rate at a queue changes (for instance when a source turns on or off), its output rate must be updated. This change in output rate causes a change of the input rate of the queues it feeds. In this way any change of a flow rate generates a *ripple* of updates through the network.

A large network has a correspondingly large number of flows. Both Nicol et al. [74] and Liu et al. [64] observed that as the size of the network increased, each

update of a flow rate became more expensive, due to the need to propagate the change. In fact with 256 switching nodes one fluid simulation executed as many events as a full packet level simulation [74]. To compound the problem, an event in a fluid simulation is more computationally expensive than a packet simulation event. This limits the application of fluid techniques in large scale simulation.

Time-stepped hybrid simulation (TSHS) [41] is an enhancement of fluid simulation. Packets are grouped into *chunks* — not dissimilar to Jain's use of packet trains [53]. All packets at a queue during a given timestep belong to the same chunk. A chunk is processed as a unit at a queue. However if the protocol simulated requires packet headers, for instance sequence numbers in TCP, these must be stored separately. TSHS replaces the buffering, dropping and processing of individual packets with a rate based calculation. If the timestep is large so that there are many packets in a chunk, this greatly reduces the number of events to be processed. Compared with a packet level simulation, TSHS is up to an order of magnitude faster. It should be noted that unlike the SSF comparison [74], two different simulators were used for the fluid and packet simulations.

Differential traffic modelling [16] [36] [15] is a new approach to the analysis of networks. It models the statistical evolution of a network by means of differential equations. It is similar to fluid simulation in that it deals with averaged quantities rather than discrete packets. The differential equations describe how the statistical values of network variables, for instance the expected queue occupancy, change with time. With traditional queueing theory it is possible to calculate the stationary state of a queue. Differential modelling extends this to allow analysis of its transient behaviour. The theory, and its hybridisation with discrete event simulation, is discussed in greater depth in Appendix A.

## 2.7 Hybrid Techniques

Hybrid simulation is the term used to describe the combination of two or more simulation methods in the same model. Many groups have applied this idea to network simulation — each with a different approach. The motivation is usually to use a fast analytic method for as much of the model as is practicable, and resort to discrete event simulation or other computationally expensive methods only when required. The hope is that the hybrid simulation will have accuracy close to that of a normal discrete event simulation, with the speed of a fast analytic model.

Schwetman [89] was among the first to apply hybrid techniques to simulation of computer systems. He modelled the competition for resources (CPU, memory, etc) among jobs in a computer system. Since the resources were modelled as a network of queues, and jobs as customers of those queues, his methods are relevant to network modelling. Schwetman divided resources into two categories — long term and short term. Long term resources included use of main memory and storage space, short term resources included the CPU and IO devices.

Discrete event simulation was used to describe the arrival of new jobs and the use of long term resources, while an analytic model was employed to describe the use of short term resources. Each job required a number of cycles through the short term resources. Rather than simulating these cycles for each job, the time interval, based on the current number of active jobs, was computed. This gave the time until the first job completed, or a new job arrived (changing the demand for resources).

This highlights one of the important issues with hybrid techniques the tighter the coupling within the problem, the harder it is to hybridise it. In this case, if jobs arrive only infrequently and persist for many cycles of short term resource usage (loose coupling) there is considerable speedup. If jobs arrive and persist only for a handful of short term resource cycles there is almost no performance gain — and decreased accuracy.

O'Reilly and Hammond [75] developed a quite different approach to hybrid simulation. They modelled a network of CSMA/CD (Ethernet) 'stations'. These stations were partitioned in two: a small number of primary stations and a larger number of background stations. The primary stations were modelled in detail by simulation. The background stations were present to provide a realistic environment for the primary stations, but were themselves simulated in a less detailed manner. In a simpler discrete event simulation the background station packets were pre-recorded and only used to create collisions with the foreground traffic. This of course meant that the primary nodes could not affect the background nodes.

In a second, time driven method, the primary stations were simulated in detail as before, but the background traffic was modelled using an algorithm.

This method was found to be more efficient than full event simulation whenever the number of stations was large (more than 1000) and where there was a high traffic intensity.

Frost et al. [34] introduced yet another hybrid simulation technique, termed conditional expectation. The simulation is of a CSMA/CD network again. Their model was based on a queue with general interarrival times and a server with a general service time distribution. They started with a known expression for the expected waiting time, $W$, of a customer in the system:

$$E[W] = \frac{E[U^2]}{-2E[U]} - \frac{E[I^2]}{2E[I]}$$

where $U = t_n - s_n$ and $t_n$, $s_n$ are the interarrival and service times, respectively, of the $n^{\text{th}}$ customer. $I$ represents the length of the idle period and cannot be found analytically. Simulation was used to estimate this parameter. With this approach they were able to model the network in the same detail as a full event simulation, but with a reduction in computational time by a factor between 9 and 16.

Huang [46], also describes hybrid combinations of abstraction techniques: in particular *mixed mode* simulation. By creating a hybrid topology of neighbouring end-to-end regions and fully simulated regions, Huang was able to take advantage of the performance gains of end-to-end packet delivery in the end-to-end regions while keeping the accuracy of simulation in congested regions where end-to-end packet delivery is inaccurate.

## 2.8 Rare Events

Many of the distributions that arise in data networks are heavy tailed, for instance the distribution of file sizes. Very rarely occurring events have an effect disproportionate to their frequency. To take an example: the 0.5% tail of file transfer protocol (FTP) bursts studied by Paxson et al. in [77] held over 40% of the data bytes transferred.

The theory of large deviations describes these events. It is one of the most active areas of probability theory. A tutorial overview using networking examples is presented in [61]. Rare events pose a challenge for network simulators as well as analysts. Large scale simulations are slow. To run a simulation for long enough for rare events to occur is not always feasible.

The goal of parallel and abstracted simulation is to model networks more quickly, so as to generate more events and explore more of the state space. A complementary approach is advocated in rare event simulation: rather than just

running faster simulations, spend more time in the areas of interest. An overview of rare event simulation is presented in [42] and a larger survey in [43].

Approaches to simulating systems with rare events include *importance sampling, split/restart* and *cloning*. Importance sampling changes a stochastic process to make rare events more frequent. Split/restart [39] halts (or splits) a simulation at a point of interest, and restarts a number of simulations from that point. This allows the state space around rare points of interest to be explored more thoroughly. Cloning, introduced by Hybinette and Fujimoto [49], extends the idea of split/restart. The approach is intended for parallel discrete event simulators. It allows for interactive injection of *decision points*. In addition rather than cloning the entire state of a model, the new state is made incrementally as it diverges from the original. However, while this approach allows for flexibility in running simulations, it does not itself enable very large simulations.

## 2.9 Network Topologies: Studies and Generators

While great attention has been paid to protocols in network simulation, far less has been paid to the effect of the topology of the network itself. Routing and traffic density are dependent on network topology. The efficiency of many network simulators may vary with the network topology, for instance if large routing tables are needed for a complex topology.

Zegura, Calvert and Donahoo [98] produced one of the first thorough comparisons of different topology models in network simulation. Although it is more applicable to the users rather than to the designers of simulators, their results are worth noting — particularly for performance evaluation.

They identified three common topology generation methods in frequent use in network simulations, and proposed their own to overcome some of the shortcomings found. The three extant methods were:

- Regular topologies — rings, trees, meshes and stars.
- Copies of existing real life networks.
- Randomly generated topologies.

There are drawbacks to all three methods. Regular topologies do not occur — large computer networks typically grow too organically for them to survive. They are more common in centrally controlled networks such as traditional telephone networks. While copies of real life networks are useful for modelling past or current networks, they are less useful for designing future networks. The authors identify several 'flat' methods for randomly generating networks, from pure random methods where the likelihood of two nodes being joined is a given fixed probability, to methods where the probability is a function of distance between nodes. They also propose a hierarchical, random, method for network generation. This *transit-stub* method consists of two types of sub-networks: stub networks, which are only sources and sinks of packets, and transit networks which can forward packets between stub networks or other transit networks.

This initial work was followed by surveys of the Internet topology and new models were developed based on the findings. Surveys of Internet topology have examined both its router level properties and Autonomous System level properties. The observation that many properties of the Internet are governed by power laws has prompted research into the extent of and the reason for this behaviour [68] [30] [13] [67].

Network topology generators fall into two broad classes: degree based and structural generators. Degree based models such as PLRG [4] and Inet [95] use the observed distributions of node degree in the Internet to generate graphs.

Structural generators such as the *transit-stub* model of GT-ITM [19] and Tiers [29] build hierarchical graphs. This is intended to emulate the division of networks into sub networks connected by backbones.

In [91] the authors systematically compare several classes of topology generators. These include the simple canonical topologies (meshes, rings, trees), several examples of degree based and structural generators as well as real networks. The results suggest that for generating large networks, the degree based approach is best. In particular, the hierarchy present in the Internet is looser than that generated by a structural model.

A further discussion of research into the properties of Internet topology and routing is presented in context in Section 3.5.

## 2.10 State of the Art

Large scale network simulation has received much attention in recent years. Many new techniques have been developed and tested. This is reflected in the large number of network simulation packages currently extant. We now survey the state of the art in the field.

- The *Scalable Simulation Framework (SSF)* [26] [27] is a an API for building discrete event simulation models. There are several implementations of the SSF, including one in Java and two in C++. The API describes five base classes and associated methods. An SSF model extends these base classes to create the specific features it requires. All details of the discrete event processing are hidden from the modeller. This allows a modeller to change SSF implementations — from a sequential to a parallel implementation for example — without modification of the code.

  SSF uses a Domain Modelling Language (DML) to allow configuration of very large networks. Definitions can be stored in a database and composed to create arbitrarily complex networks.

  DaSSF is a C++ implementation of SSF developed at Dartmouth College. The event processing subsystem uses conservative synchronisation of event queues to run on shared memory symmetric multiprocessors. All processors exchange events at set intervals. The choice of this synchronisation period is such that events within these periods can be processed without affecting causality.

  SSF has been shown to be capable of modelling networks of hundreds of thousands of UDP or TCP nodes, on shared memory symmetric multiprocessors.

  Recently, DaSSF has been extended to run on distributed memory parallel computers [65] using MPI. However the SSF API makes some assumptions that hold only in shared memory environments — in particular that the network configuration as a whole is available to each processor. In a distributed memory system the model must be preprocessed to deal with global naming issues, since the processors do not have access to the full network configuration.

SSF, because of the generality of its base classes, can be used to model more than just wired networks. In particular it is being extended to model ad hoc wireless networks [65].

Of the network simulators discussed in this section, SSF is closest in spirit to our work. The similarities include a strong emphasis on very large scale simulation of TCP networks and the use of PDES techniques. However SSF was originally designed to run on shared memory computers and has only recently been extended to run on distributed memory computers. The most important difference between SFF and our work is our greater use of abstraction. Our use of AR allows for the simulation of larger and more realistic networks. On the other hand, SSF has more detailed and realistic implementations of network protocols.

- The *ns* simulator [8] is an extremely mature and capable network simulator. It provides a rich suite of modules for simulating network devices and protocols. It is written in a mixture of TCL and C++, and is easily extensible, making it a popular platform on which to develop and test new ideas. In particular it has been used to test hybrid and highly abstracted modelling techniques [46] and has been extended to support parallel simulation [85].

- Parallel/Distributed *ns* (*pdns*) [85] is an extension of *ns* that runs on parallel computers. The network is distributed between processors. *pdns* uses the RTIKIT [44] to replace the *ns* event scheduler. RTIKIT is a conservative parallel discrete event scheduler. RTIKIT implements its own message passing system over either a TCP/IP network or Myrinet. *pdns* allows a linear scaling of simulation size, while maintaining or possibly speeding up the execution time compared to a sequential simulation of the same size.

  The latest version also includes the *NIx-vectors* stateless routing algorithm [84]. With the reductions in memory that this method makes possible, *pdns* can model networks of over 250,000 nodes.

  The extensibility of *ns* and *pdns* comes at a price. The demonstration by Riley et al. [82] that an Internet scale simulation in *pdns* is not possible due to memory and computational requirements emphasises the importance of highly efficient programming. In contrast to *ns* and *pdns*, memory efficiency is a high priority in our simulator (see Chapter 4). Likewise, we chose to

use the C programming language solely, as although a language such as TCL offers increased flexibility and extensibility, it does not have the performance of C.

- The *Ultra-large Scale Simulation Framework (USSF)* is described by Rao and Wilsey [80]. Their aim is to simulate millions of network entities using parallel techniques. In particular they seek to harness low cost commodity computer systems, rather than dedicated multiprocessing clusters. USSF isolates the simulation modules from the underlying simulation kernel. This allows USSF to use different kernels. Rao and Wilsey [80] describe the deployment of USSF on WARPED [20] and on NoTime [79]. These are an optimistic parallel discrete event simulator and an unsynchronised parallel discrete event simulator respectively.

  USSF is demonstrated to be capable of simulating networks of hundreds of thousands of nodes.

- *Parsec*(parallel simulation environment for complex systems) [7] and *GloMoSim* (Global Mobile System Simulator) [100] are related projects developed at UCLA. *Parsec* is a simulation language that provides a discrete event simulation kernel that runs both sequentially and in parallel. *GloMoSim* is a library for building models of wireless networks. It uses *Parsec* as its foundation.

  *Parsec* is based on the older *Maisie* language [6], but considerably enhanced. It is a C based library that manages the message passing in a parallel discrete event simulation. It has a large selection of synchronisation protocols — including conservative, optimistic and mixed protocols. The protocol used in a simulation can be changed without affecting the rest of the simulation. This allows the most appropriate synchronisation protocol for a given model to be easily selected.

  *GloMoSim* simulates wireless networks. This is in itself a more difficult task than simulating wired networks, since signal interference and attenuation are much more significant than in wired media. In addition, due to the broadcast nature of wireless communications, the topology is often denser. It uses *Parsec* as its event handling kernel.

  The protocol stack is broken into several layers that communicate through

a common API. It includes a TCP/IP implementation and several media access protocols as well as intermediate layers.

*GloMoSim* supports two forms of partitioning: horizontal and vertical. In the first each network layer is simulated by a different processor, in the second the nodes are partitioned and each processor simulates a given partition.

On an IBM 9076 SP *GloMoSim* is capable of simulating up to 3000 mobile nodes in 800m × 800m area. With 16 processors it achieves a speedup of between five and eight — the speedup increases with node density.

- The Dynamic Simulation Backplane [83] is not a simulator itself. Rather it is a framework for connecting other simulators together. For instance, one simulator may provide a rich set of network protocols, while another has advanced wireless simulation capabilities. The backplane provides an interface through which the two simulators can communicate. Each simulator registers the protocols which it requires or provides. The backplane can link together models that have compatible needs. This is a flexible approach that takes advantage of disparate strengths of the simulators with which it can interface. Currently it has interfaces for *ns* and *GloMoSim*, and an interface for *OpNet* is planned.

## 2.11 Summary

The goal of this thesis is to develop new techniques to enable efficient simulations of very large networks, in particular TCP/IP networks. This chapter has introduced the techniques and methods that have been developed by network simulation researchers to allow the simulation and analysis of larger and more complex telecommunications networks. In the following chapters we built on this base to create network simulations of unprecedented size and speed.

# Chapter 3

# Algorithmic Routing

Arpanet, the predecessor of the Internet, used a distributed routing algorithm. It took queueing delays at each link as a metric to decide between alternative paths from source to destination. These measurements were forwarded to all routers. However under heavy load this system was prone to routing oscillations. The hop count, or number of links a packet traversed, was later used, as it proved to be more stable.

The Internet is a collection of independent networks, including the original Arpanet, sharing common protocols. At the highest level it consists of Autonomous Systems (AS). Routing, rather than being a single flat layer, is instead a two layer hierarchy. Each Autonomous System manages its own internal routing, with one or other of the several Internal Gateway Protocols. Routing between Autonomous Systems is managed by an Exterior Gateway Protocol: typically BGP4.

The routers within an Autonomous System usually have access to a large amount of knowledge about the network topology within the Autonomous System. The most popular Internal Gateway Protocol is Open Shortest Path First, OSPF. It is a link state protocol, based on the Dijkstra shortest path algorithm [28]. In a link state protocol each router distributes the distance to its neighbours to every other router in the network. Each router can then apply Dijkstra's algorithm to calculate the shortest path to each destination in the network. In OSPF the distance metric can be chosen; it could be based on the bandwidth of links, or simply the number of hops in the path. OSPF is a memory and CPU intensive algorithm, since it creates a graph containing every node in the network, though there are various ways to make it more efficient.

The first Exterior Gateway Protocol was simply called EGP. It constructed a path between source and destination that traversed the fewest number of Autonomous Systems. As the Internet grew and commercialised, network managers demanded greater control over routing between networks, often because of contractual agreements between network service providers, load balancing, cost minimisation and other factors that EGP did not take into account. Today the most commonly used Exterior Gateway Protocol is BGP version 4. It allows providers to specify policies, for instance, on where packets enter or exit their system or whether or not to allow packets between other providers to traverse the network. If there are several possible paths from source to destination, BGP chooses the path with the best metric. The simplest metric is the AS–path length, which is the number ASs crossed on the path to the destination.

Two connected BGP routers in different Autonomous Systems exchange messages detailing the networks to which they are connected, and pass on other such messages that they have received from other routers. Business policy may also play a role. For instance a small provider might have links to two larger backbone providers. It will want to accept packets originating or terminating within its own network, but will not want to have packets between the two larger networks traversing its own small network.

As the Internet has grown, the size of the routing tables has also grown in tandem. The hierarchy imposed by Autonomous Systems helps, but routing tables may still have over 50,000 entries, taxing the computational capacity of routers. The fact that this figure is as low as it is, is due to route aggregation. Rather than routers advertising each network or host individually, they are aggregated into blocks of contiguous address space, and the route to the entire block advertised as a single path.

How do the exterior and interior protocols interact? BGP calculates the next Autonomous System in the path and OSPF provides the path from the source to a BGP connected to that Autonomous System.

A key property of Internet routing is that each router maintains its own routing table, even if this is built by communicating with its neighbours. This is necessary due to the decentralised nature of the Internet. However in simulation we are not bound by the same constraints as router designers. For instance, a simulated router has access to global network information that is not available to a real router. This allows the simulated router to more easily calculate routes. The approximation

introduced is that less routing information packets are sent through the network. Whether that is acceptable depends on the simulation goals. Unless we are interested in the details of the routing protocols it is more efficient to precalculate routes and store them, than to have each simulated router build its own routing tables by the exchange of BGP or OSPF messages. This centralised computation of routing information is a common approximation. It has several advantages: the simulation does not need to include complex protocols, computation overhead is reduced and memory requirements lowered. However some phenomena, such as route flapping (rapid, sometimes periodic, changes to routing tables), will not appear, so central computation of routes is not always desirable. See [46] for a more detailed discussion.

Unfortunately centralised computation of flat routing tables is not a panacea. It has memory requirements that scale $\mathcal{O}(N^2)$, where $N$ is the number of nodes in the network. Each router must know the path to every other router. As an example consider a network of 10000 nodes. A flat routing table with a single four byte integer for each entry would use 400 MB memory.

Kleinrock and Kamoun [58] demonstrated that the minimum routing table size in a hierarchical network is obtained when there are $\log N$ levels. The minimum table size per node is $e \log N$ for a total size of $\mathcal{O}(Ne \log N)$. For a network with $\log_k N$ levels, where $k$ is the branching factor, the total space complexity is $\mathcal{O}(kN \log_k N)$. The Internet, for example, while hierarchical in nature, has only two levels of hierarchy, and is unbalanced. Therefore

$$\log_k N = 2,$$

$$\implies k = \sqrt{N},$$

so the total space complexity is

$$\sqrt{N} N \log_{\sqrt{N}} N,$$

and

$$\log_{\sqrt{N}} N = 2,$$

so we can write the total space complexity as

$$\mathcal{O}(2N\sqrt{N}).$$

The space complexity of routing within the Internet backbone is $\mathcal{O}(N_b^2)$, where $N_b$ is the number of nodes in the backbone. The scaling properties of routing tables pose a problem for simulations. A simulation of a large network would require an inordinate amount of memory for the routing tables. In order to increase the size of simulated networks a more memory efficient method of implementing routing is needed. The two existing methods, *NIx–Vectors* and AR have been introduced in Section 2.5, and the scaling properties of each discussed. In the remainder of this Chapter we explore in detail the performance, scaling and fidelity to Internet routing of AR. AR is described in Section 3.1. Performance improvements are detailed in Section 3.2 and route quality improvement in Sections 3.3 and 3.4. The observed properties of Internet routing are discussed in Section 3.5 and contrasted with simulated properties generated using AR. We then apply the techniques developed to two large, realistic case studies.

## 3.1 Algorithmic Routing

AR was introduced by Huang [46], was further developed by Huang and Heidemann [47] and implemented in the *ns* [8] simulator. The idea was based on work for binary tree networks [78]. It trades computational complexity for a reduction in memory usage.

Huang et al. noticed that by mapping the network topology onto a $k$-nary tree, and numbering the nodes by a scheme described in Section 3.1.2, it is possible to calculate the next node the packet must visit on its path from source to destination. Before describing the algorithm, we introduce the notation we will be using.

### 3.1.1 Definitions

A network topology can be mapped to a graph $G = (V, E)$ where $V$ is a set of vertices and $E$ is a set of edges joining pairs of vertices. The number of vertices is denoted by $n = |V|$, the number of edges by $|E|$. An edge between vertices $v_i$ and $v_j$ is denoted by $(v_i, v_j)$. The degree of $G$ is the maximum number of edges connected to any node. A spanning tree of $G$ is defined as $T_s = (V, F)$ where $F \subseteq E$, $|F| = |V| - 1$ and all vertices are connected. A rooted tree of $T = (T_s, r)$ of $G$ is a spanning tree $T_s$ of $G$ and a vertex $r \in V$ designated as the *root* node. Let $p, c \in V$. We say that $p$ is the *parent* of $c$ if $(p, c) \in F$ and $p$ is closer to the

root than $c$. In this case $c$ is the *child* of $p$. A vertex $d$ is a descendent of vertex $a$ if there is a child to parent path from $d$ to $a$. We define a subtree $\mathcal{S}(a, T)$ of a tree $T$ to be the subset of $V$ consisting of vertex $a$ and all vertices that are descendants of $a$. A $k$–nary tree is a rooted tree where each vertex has at most $k$ children.

### 3.1.2 AR Setup

AR superimposes a $k$-nary tree on the original network topology. This tree can be generated by a depth first search or breadth first search (BFS) of the network. The value of $k$ is the maximum number of children possessed by any node in the tree. If there are any loops in the network, they are broken by the mapping. The choice of search algorithm has implications for the accuracy of the routing — see Section 3.1.6 and 3.3. Figure 3.1 is an example of a map from a network topology to a tree.

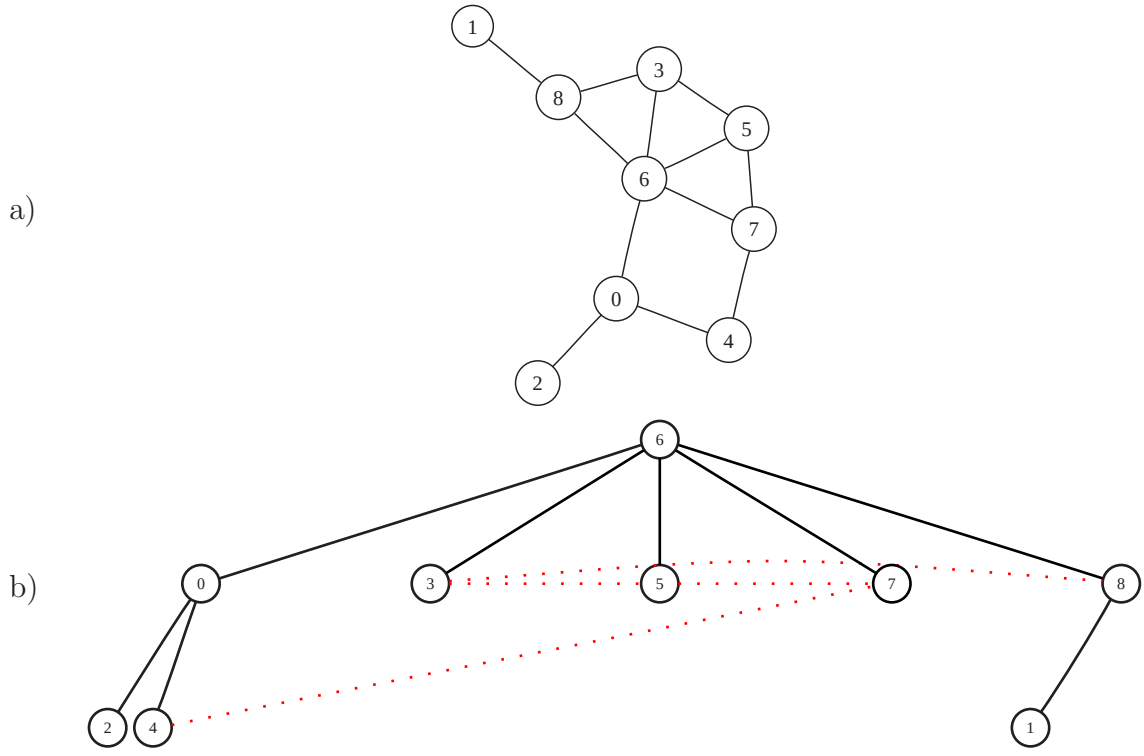



*Figure 3.1: a) A simple network. b) The same network mapped to a tree. Links that have been broken are drawn in red dots.*

The second stage of the setup phase is to assign a new numbering scheme to the nodes. The root of the tree is labelled 0. The other nodes are labelled as in

Figure 3.2, where the child of node $i$ is given by $ki + j, j \in 1 \dots k$. The parent of any node $i$ is given by

$$\lfloor \frac{i-1}{k} \rfloor. \tag{3.1}$$

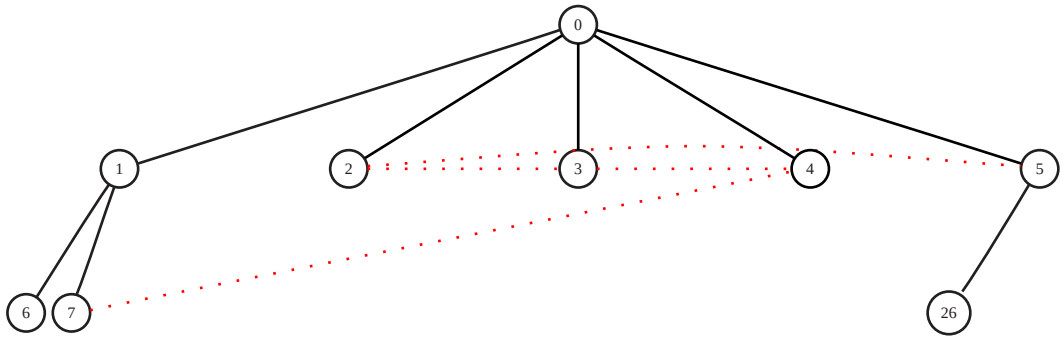



*Figure 3.2: The network from Figure 3.1 with new node addresses. Here $k = 5$. The parent of a node can be found using Equation 3.1. For example the parent of node 26 is $\frac{26-1}{5} = 5$.*

The memory needed for the new naming scheme is $\mathcal{O}(N)$. The tree mapping and name reassignment are of $\mathcal{O}(N)$ complexity.

### 3.1.3 Next hop calculations in AR

Each time a packet arrives at an intermediate router, it needs to be forwarded one step further. Suppose the packet is at the node labelled $s$ in the new $k$-nary tree scheme and its destination is node $d$, now its next hop is calculated using Algorithm 1.

In this algorithm we start at the source node $d$ and follow a child to parent path up the tree, stopping at either the root node or at the destination node $s$, whichever is is encountered first. If $s$ is encountered first, then the node just before $s$ is the next hop in the path. If the root node is encountered first, then the next hop is the parent of $s$. The packet is then forwarded along the link that connects to the next hop node.

The tree has a depth of approximately $\log N$ so the algorithm has a computational complexity of $\mathcal{O}(\log N)$.

**Input:** Routing tree of degree $k$. Source node $s$, destination node $d$.
**Output:** The address of the next node in the path from $s$ to $d$.

```
(1)  if d = 0
(2)     return (s−1)/k
(3)  y ← d
(4)  while y ≠ 0
(5)     if (y−1)/k = s
(6)        return y
(7)     y ← (y−1)/k
(8)  return (s−1)/k
```

*Algorithm 1: The basic AR algorithm.*

### 3.1.4 Lengthening of Routes

The routes generated by AR are not always the shortest possible, some distortion is introduced. Packets are only forwarded between nodes that are connected in the $k$-nary tree. In this tree, which is superimposed on the real network topology, there is only one route between any two nodes. In the real topology there may be many. This problem occurs wherever there is a cycle in the network topology (three or more nodes that are connected in a ring). Figure 3.3 illustrates the problem. To send a packet from $A$ to $C$ by shortest path routing is one hop. However mapping the network to a tree breaks the $A-C$ link, so the packet takes two hops $A-B-C$ instead.

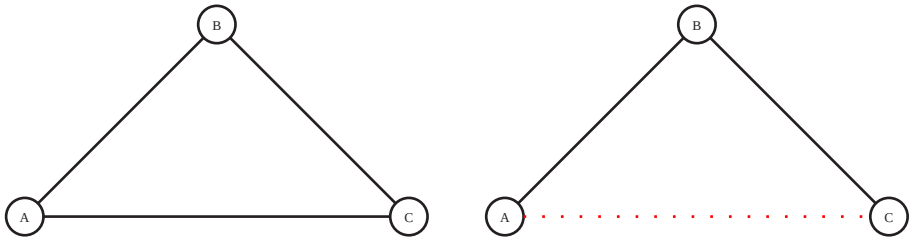



*Figure 3.3: When the network (left) is converted to a tree (right) the link $A-C$ is broken. Packets from $A$ to $C$ must now travel $A \rightarrow B \rightarrow C$ rather than going directly to $C$.*

As the size of the cycle increases, so does the number of extra hops. We define

relative difference in lengths, $r$ as:

$$r = \frac{D_a - D_s}{D_s}$$

where $D_s$ is the length of the shortest path between two nodes and $D_a$ is the length of the AR path. The average relative difference, $R = \overline{r}$, can be shown to be:

$$R = \frac{k}{2k+1}$$

where $k$ is the largest integer less than $\frac{N-1}{2}$ (see [48] for details). As $N$ increases, $R$ approaches 0.5, a 50% increase in route length.

In a network there may be many cycles. Each cycle contributes to raising the average relative difference of path lengths, $R$. The length of cycles is also a factor — though interlocking and overlapping cycles make analysis more difficult than with a simple ring network. In addition to the number and size of cycles in the network, the location of the cycles also has large affect on $R$. If the cycles are on the periphery, as leaf nodes, the cycles will only raise the route length of routes ending there. If, on the other hand, the cycles are in the backbone of the network, they will have a greater influence on $R$ since most routes need to traverse the backbone. Finally, since there is only one path between any two branches of the tree, whereas there may be many paths between the same nodes in the real network, traffic may be concentrated in certain links and routers, while others are idle.

In Section 3.2.3 several sample networks, both real and generated, are used to compare shortest path and AR. AR is an approximation, and like all approximations introduces errors. Whether these errors are important depends on the purpose of the simulation. The first factor is the network itself; if it contains large numbers of cycles or very large cycles, the node to node paths may be greatly altered. If the cycles are in the backbone the effect is amplified. Networks which already possess a treelike structure are least affected. Simulations that are only concerned with a small number of senders can have increased accuracy by using several routing trees.

### 3.1.5 Implementation Problems

There are some implementation problems with AR. The method requires mapping each node in the network to a new id number. If the maximum number of neighbours of any node in the network is $k$, then the network must be mapped to a $k$-nary tree. If the tree has depth $d$, then the largest number needed in the mapping is:

$$L = \sum_{i=0}^{d-1} k^i = \frac{k^d - 1}{k - 1} - 1 \approx k^{d-1}.$$

AR, even in a small network, will overflow a four byte integer if there is a combination of moderate depth and a single node with a large number of neighbours.

This explosion in mapped node addresses causes two problems. Firstly the memory requirements for a node address rapidly exceed the standard four byte integer. The memory needed for the node addresses scales as $\mathcal{O}(N \log_2(k^{(d-1)}))$ binary bits. Secondly the new addresses are typically very sparsely distributed in the range $0 \ldots L$. This is especially the case if the maximum number of neighbours is much greater than the average number of neighbours. With the original addresses in the range $0 \ldots (N-1)$ a simple array of length $N$ can be used to point to the memory used to store a node. With the mapped addresses an array of length $L$ would be required. This is clearly grossly inefficient. Sparse matrix techniques or a tree data structure could be used to work around the problem, but this both increases memory consumption and adds computational complexity.

As an example, using a network topology from the Internet circa November 1999 [51], the maximal observed number of neighbours is 1937, and the depth of a mapped tree is between 12 and 14 depending on the root node chosen. The largest node id in the resulting tree is approximately $1937^{13}$. This requires eighteen bytes of storage space, far exceeding the four bytes of a normal integer.

### 3.1.6 Existing Enhancements to Algorithmic Routing

Many of the problems associated with AR have already been noted by researchers. There have been several proposals to avoid the worst of the issues. These proposals can be roughly categorised into performance and route quality improvements to the technique.

### Performance Improvements

Huang et al. suggest [47] that a *Least Common Ancestor* (LCA) [94] algorithm could be used to perform fixed cost, $\mathcal{O}(1)$, routing.

The LCA of two nodes $x$ and $y$ in a tree is the node that is ancestor of both $x$ and $y$ and is furthest from the root of the tree. The LCA problem has several solutions that can answer LCA queries in constant time, although varying amounts of preprocessing are necessary. Most of these solutions are extremely complex. In fact it is said that

> ... the folk wisdom of algorithm designers holds that the LCA problem still has no implementable optimal solution. Thus, according to hearsay, it is better to have a solution to a problem that does not rely on LCA precomputation if possible [11].

However, Bender and Farach-Colton [11] present a simplification of the original parallel algorithm developed by Schieber and Viskin [94]. This algorithm requires $\mathcal{O}(N)$ time and space for preprocessing and $\mathcal{O}(1)$ time for a LCA query.

The second improvement suggested in [47] is to replace the indiscriminate BFS of the network, with a BFS that selects highly connected nodes in preference. We tested a simple form of this idea, as follows: when all nodes at depth $n$ have been identified, they are sorted in descending order according to their connectivity. When visiting the children of a given node, they are visited again in descending order of connectivity.

We define $H$ to be the sum over all pairs of the number of hops needed to connect any pair of nodes, $i$, $j$ in a $N$ node network:

$$H = \sum_{i=1}^{N} \sum_{j=i+1}^{N} \text{Dist}(i, j) \tag{3.2}$$

Then let $H_p$ be the number of hops when the BFS tree is constructed giving priority to highly connected nodes. For a random seed $s$, let $H_s$ be the $H$ that results from choosing nodes randomly during the BFS. Let $H_{\text{avg}}$ be the average of $H_s$, $s \in S$, and $H_{\text{max}}$, $H_{\text{min}}$ be the maximum and minimum respectively. Denote by $D$ the value of $H$ in shortest path routing.

Values of $\frac{H_p}{D}$, $\frac{H_{\text{avg}}}{D}$, $\frac{H_{\text{max}}}{D}$ and $\frac{H_{\text{min}}}{D}$ for example networks ranging in size from one hundred to two thousand nodes are presented in Table 3.1. In each of these examples one hundred different seeds were used.

| Nodes in Network | $\frac{H_p}{D}$ | $\frac{H_{\text{avg}}}{D}$ | $\frac{H_{\max}}{D}$ | $\frac{H_{\min}}{D}$ |
|---|---|---|---|---|
| 10 | 1.12 | 1.16 | 1.20 | 1.12 |
| 100 | 1.05 | 1.06 | 1.07 | 1.05 |
| 200 | 1.09 | 1.09 | 1.09 | 1.09 |
| 300 | 1.14 | 1.14 | 1.14 | 1.14 |
| 400 | 1.12 | 1.14 | 1.17 | 1.09 |
| 500 | 1.12 | 1.16 | 1.19 | 1.12 |
| 800 | 1.19 | 1.20 | 1.21 | 1.19 |
| 1000 | 1.18 | 1.22 | 1.24 | 1.19 |
| 2000 | 1.13 | 1.14 | 1.15 | 1.13 |

*Table 3.1: Relative path lengths for some sample networks. $H$ is the total distance between distinct, unordered, node pairs. $H_p$ is the value of $H$ when nodes are chosen by order of connectivity in the BFS. $H_{\text{avg}}$ is the average value of a large number of random orderings. $H_{\min}$ is the best value of these random orderings. $D$ is the value of $H$ using shortest path routing.*

Figure 3.4 shows the values of $H$ using three different nodes as root. Each line represents the variation in $H$ for different choices made during BFS.

It is clear from Table 3.1. that ordering the nodes is an improvement over random choice in the BFS construction of the tree. However it is a slight improvement, typically lowering the relative increase over shortest path routing by between 0.5 and 4 percentiles. The improvement over the worst case is better — between 1 and 8 percentiles. After the setup phase of AR there is no extra cost involved.

### Quality Improvements

Huang et al. [47] propose some enhancements to improve the quality of the routes generated by AR. A tree generated by BFS has the property that the path from the root node to all other nodes is a shortest path. Consider a simulation with $s$ important source nodes. If a tree is generated for each of these source nodes, then routing to and from these nodes will be shortest path. A separate tree may be used for less important background traffic. This scheme requires $s$ times the memory of a single tree. It guarantees correct routing for important traffic flows. However it does not prevent congestion on certain links due to the concentration of the background traffic on the $N-1$ links of the background traffic routing tree.

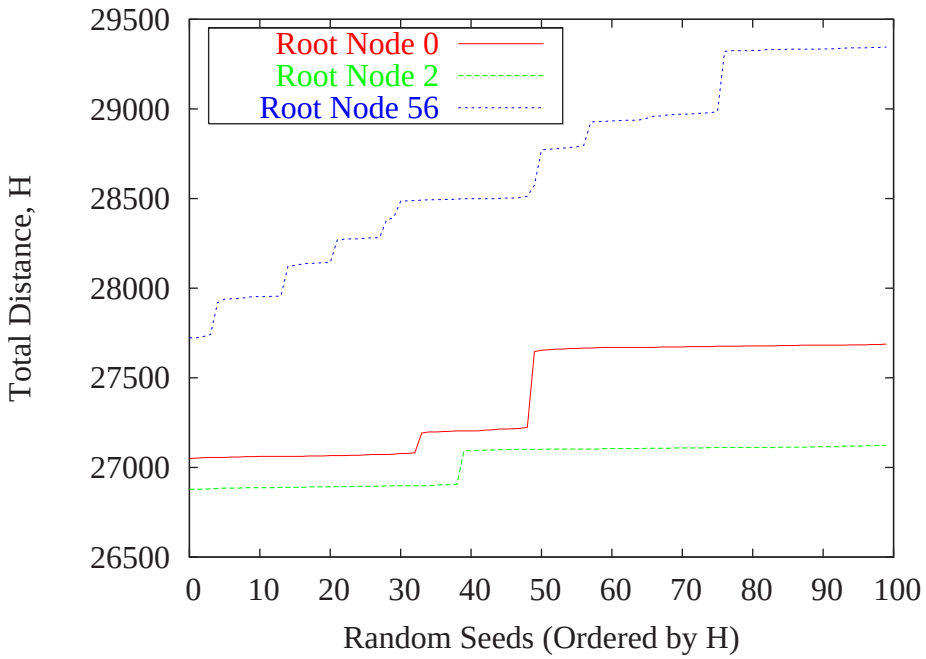



*Figure 3.4: Each line shows a range of possible values of H for a single choice of root node. The order in which nodes are visited during BFS has a large impact on H. Here the order is selected randomly and H plotted by increasing value.*

In addition, this method is only feasible for relatively small values of $s$.

A second suggestion in [47] concerns a network with a large ring component. If two trees are rooted on opposite sides of the ring, then by choosing the appropriate tree for a given source and destination pair, it is possible to get shortest path routing. In a large and complex network with many interlocking cycles it would be much more difficult and expensive to do this.

Their final proposal is to perform two passes while routing. The first pass checks to see if the destination is directly connected to the current node. If the destination is directly connected to the current node, then the packet is passed to it, even if the link joining the two nodes is not present in the routing tree. If the two nodes are not directly connected then the normal algorithm is applied to determine the next hop. This will result in shorter paths for all routes in which a node in the AR path is separated from the destination by a broken link. However this is a relatively small number of routes, and for a network with $N$ nodes and $E$ links, requires on average $\frac{E-N+1}{N}$ extra checks in each next hop calculation.

## 3.2 New Efficiency Improvements

In this section we introduce two separate improvements to AR. The first is a small implementation change to the basic method which makes the tree structure implicit in the network graph and avoids the overflow issues discussed in Section 3.1.5. It also has the advantages of not requiring any memory (depending on the implementation), and replacing several arithmetic operations by a single lookup. The second is a new algorithm which has fixed $\mathcal{O}(1)$ complexity, regardless of network size, but produces the same paths as basic AR.

### 3.2.1 Direct Algorithmic Routing

In AR each node is given a new address when the network is mapped to a tree. Given the address of a node one can calculate its parent node and any possible child nodes. This, however, is more information than is actually needed. In calculating the next hop from a node, the only requirement is the ability to determine the parent of any node.

Direct AR is the name we have given to an enhancement of AR that obviates the need for mapped node addresses. It assumes that each node stores a list of its neighbours in an array. When the network is being mapped to a tree, rather than assigning a new address to the node, this neighbour list is instead reordered so that the parent is placed at a known position in the list: first or last, for example. However, the root of the tree must be explicitly stored. Direct AR also has the advantage of replacing the $(i-1)/k$ calculation to find the parent of node $i$ with a direct memory lookup. Division is a relatively expensive operation, so this is a measurable improvement. If it is not practical to reorder the neighbour list, the parent of each node may be explictly stored, at a cost of $\mathcal{O}(N)$ memory. Even so it retains the performance improvement.

Consider the small network in Figure 3.5. It might be stored, prior to mapping, as Figure 3.6a. After the mapping to a tree, the root is noted, and the neighbour lists are reordered so that that parent node is first in each list, Figure 3.6b. We now have a system in which the tree structure is implicit in the ordinary network structure. The parent of a node can be instantly determined by looking at the first element of the neighbour list, and the children are those neighbours that have the node as their parent. Finding the children is more complicated than before, but that does not matter since they are not needed in the next hop calculation.

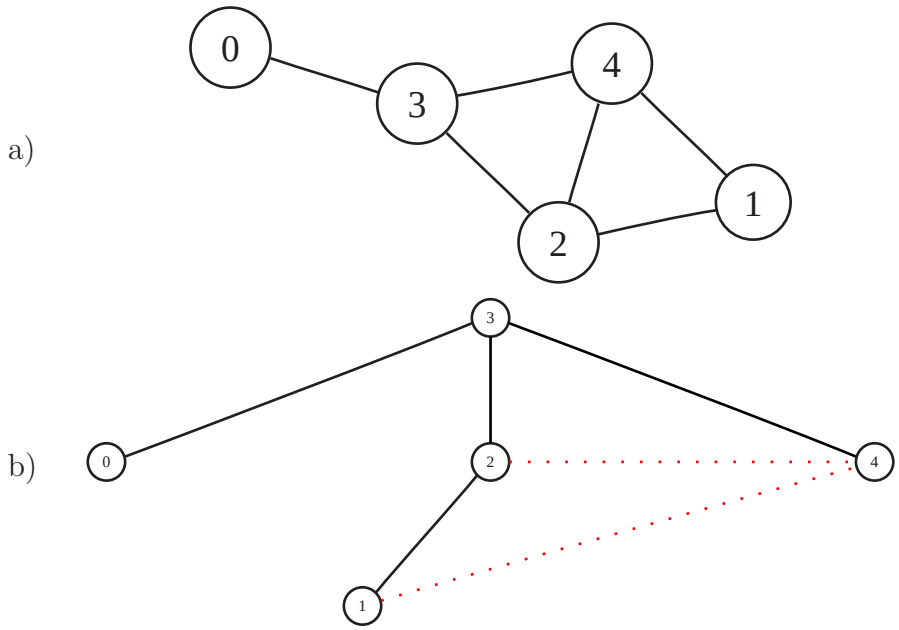



*Figure 3.5: a) A simple network. b) The same network mapped to a tree. Links that have been broken are drawn in red dots.*

The next hop algorithm itself needs to be changed slightly. As before, let $s$ and $d$ be the source and destination respectively. The modified next hop calculation is presented in Algorithm 2.

Why is this better than normal AR? There are a number of reasons. It requires no extra memory for mapping new node addresses. Since we don't use new addresses it does not suffer from the address overflow problems discussed in Section 3.1.5. Finally, finding the parent of a node by direct lookup is faster than performing the Parent$(i) = (i-1)/k$ calculation, especially since division is a relatively expensive operation. The next hop calculation still has $\mathcal{O}(\log N)$ complexity.

Section 3.2.3 compares the costs and performance of this new method with the original algorithm.

### 3.2.2 Fixed Cost Routing

AR has $\mathcal{O}(N)$ memory requirements and $\mathcal{O}(\log N)$ complexity. Direct AR has no extra memory requirements, but has $\mathcal{O}(\log N)$ computational complexity. In this section a third algorithm is introduced. This has $\mathcal{O}(N)$ memory requirements, but fixed cost, $\mathcal{O}(1)$, computational complexity. However it does suffer from the same implementation problems as AR.

a)

| Node | Neighbours |
|---|---|
| 0 | 3 |
| 1 | 2 4 |
| 2 | 1 3 4 |
| 3 | 2 4 0 |
| 4 | 1 2 3 |

b)

| Node | Neighbours |
|---|---|
| 0 | 3 |
| 1 | 2 4 |
| 2 | 3 1 4 |
| 3(root) | 2 4 0 |
| 4 | 3 1 2 |

*Figure 3.6: Two representations of the network in Figure 3.5.a) Pre-mapping representation of network. b) Post-mapping representation of network.*

Consider Figure 3.7: if the nodes are numbered using the AR numbering scheme, it should be possible to determine whether a destination node is in Region A, B or C, ie whether the next hop is to the parent node or to one of the child nodes. In this section we present an address scheme similar to that of AR that allows us to do this. We term the new numbering scheme, and associated algorithm, Fixed Cost Routing.

Rather than using one number as the address of a node, we use two numbers. The first part of the address, $x$, is unique among all other nodes at the same depth in the tree. The second part of the address is the depth of the node in the tree. The root node has an address of $(0, 0)$. As in direct AR, the list of neighbouring nodes is reordered at each node so that the parent node is at the end of the list.

Let $k$ be the maximum number of neighbours belonging to any node in the network. Let $p$ be a parent node with child node $c$. Let the address of $p$ be $(x_p, y_p)$ and the address of $c$ be $(x_c, y_c)$. Let $c$ be the $i^{\text{th}}$ node in the neighbour list of $p$, with $i \in \{0 \ldots k-1\}$. Then $(x_p, y_p)$ and $(x_c, y_c)$ are related by:

$$y_c = y_p + 1 \tag{3.3}$$

$$x_c = x_p \times k + i \tag{3.4}$$

This is illustrated by example in Figure 3.8.

**Input:** Routing tree with a named root. Function *Parent* that returns the parent node of a non-root node. Source node $s$, destination node $d$.
**Output:** The next node in the path from $s$ to $d$.

```
(1)  if d = root
(2)     return Parent(s)
(3)  y ← d
(4)  while y ≠ root
(5)     if Parent(y) = s
(6)        return y
(7)     y ← Parent(y)
(8)  return Parent(s)
```

*Algorithm 2: The implicit AR algorithm, in which parent nodes are stored rather than calculated.*

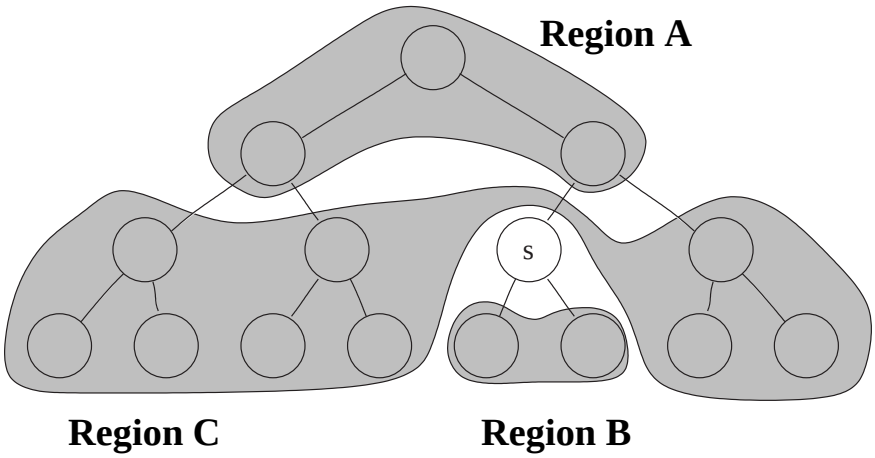



*Figure 3.7: The destination node can be categorised into one of three types, according to the region it is in.*

Let $s$ and $d$ be a source and destination pair. Initially node $d$ can be sorted into one of three regions, see Figure 3.7, on the basis of the nodes' addresses. The first area, Region A, is all nodes that are higher than $s$ in the tree. This is easy to test for: if $y_d < y_s$, then node $d$ is in region A. The second area, Region B, consists of all nodes directly below node $s$ in the tree, so $y_d > y_s$. Now, from Equation 3.4, if $d$ is in this region then:

$$x_d \in \{x_c \times k^{(y_d - y_s)}, \ldots, x_c \times k^{(y_d - y_s)} + k - 1\}$$

If $d$ belongs to neither of these first two regions, it must then belong to the third,

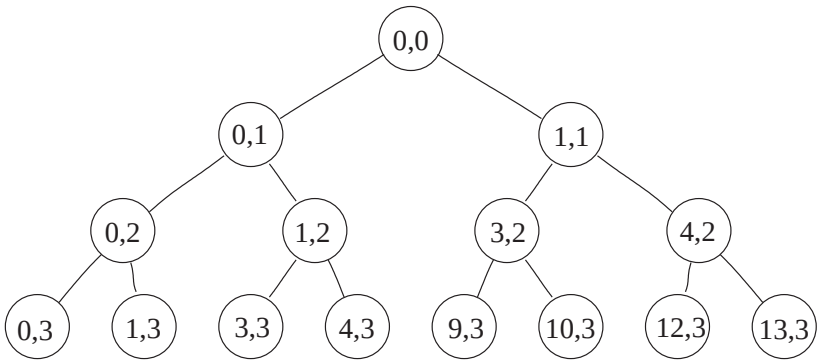



*Figure 3.8: The two number address scheme of fixed cost routing. The maximum node degree, $k = 3$.*

Region C: nodes that are below $s$ in the tree, but in a separate subtree.

If the destination node is in either Region A or Region C, then the next hop from the source $s$ is the parent of $s$. If the destination is in Region B, there may be more than one child node that could be the next hop. Let $q = x_d - x_s \times k^{(y_d - y_s)}$. Then the next hop node's position in the neighbour list of node $s$ is the integer part of:

$$\frac{q}{k^{(y_d - y_s - 1)}}.$$

Algorithm 3 shows the full procedure in practice. Below are three examples using Figure 3.8:

**Example** $(3,2) \rightarrow (0,1)$**:** $y_d < y_s$ since $1 < 2$ so the next node is the parent of $(3,2)$.

**Example** $(3,2) \rightarrow (12,3)$**:** $y_d \not< y_s$ since $3 \not< 2$ and $y_d \notin \{3 \times 3^{3-2} \ldots 3 \times 3^{3-2} + k - 1\}$ since $12 \notin \{9 \ldots 11\}$

**Example** $(1,1) \rightarrow (10,3)$**:** $y_d \not< y_s$ since $3 \not< 1$. But, $x_d \in \{1 \times 3^{3-1} \ldots 1 \times 3^{3-1} + k - 1\}$ since $10 \in \{9 \ldots 11\}$ and $q = x_d - x_s \times k^{(y_d - y_s)} = 10 - 1 \times 3^{(3-1)} = 1$ and the integer part of $\frac{1}{k} = 0$. So the next hop is the first item in the neighbour list (using a zero offset) which is $(3,2)$ as expected.

Although this method has a fixed cost for all next hop calculations, that does not by itself make it more efficient for all networks one might wish to simulate. Currently only networks with less than one million nodes can feasibly be simulated. In AR it is necessary to perform a calculation on average $\log N$ times to find the

**Input:** Routing tree with maximum degree $k$. Each node $a$ in the network has a two part address $(x_a, y_a)$. The root node is $(0, 0)$. If node $p$ is parent of node $c$ then $y_c = y_p + 1$ and $x_c = x_p \times k + i$ where $0 \leq i < k$. The function *Parent* returns the parent of a node (either by lookup or by calculation). Source node $s$, destination node $d$.
**Output:** The next node in the path from $s$ to $d$.

```
(1)   Δy ← y_d − y_s
(2)   if Δy < 0
(3)      return Parent(s)
(4)   else
(5)      q ← k^Δy
(6)      r ← x_d − x_s × q
(7)      if r ∈ {0 ... (x_s × q − 1)}
(8)         n ← Integer Part(r / k^(Δy−1))
(9)         return neighbour n of node s
(10)     else
(11)        return Parent(s)
```

*Algorithm 3: Fixed cost AR.*

next node — roughly 14 times with a million node network. We need to perform one calculation with fixed cost AR. However it is conceivable that this single calculation might be more than 14 times as expensive. In AR (Algorithm 1), the most expensive part of the calculation is the division to find the parent node. In fixed cost routing it is necessary to calculate $k^{\Delta y}$ twice and perform several multiplications, additions and a division. The power calculation is the expensive part. It is relatively simple to work around this problem: since $\Delta y$ has a small range of values — typically 1 to $\log N$ it is easy to precalculate them and perform a table lookup rather than repeat the power calculation each time. Section 3.2.3 contains systematic comparisons of the methods.

This fixed cost method suffers from the same problem with large node address values as the original method. The largest number needed is of the order of $k^d$ where $k$ is the maximum node degree, and $d$ is the depth of the tree. If addresses are restricted to standard 4 or 8 byte integers, the size of networks we can study is limited. An alternative is to use multiprecision arithmetic libraries, such as GNU MP [33]. The disadvantage is that memory use no longer scales linearly with $N$, and the cost per arithmetic operation is no longer constant. Section 3.2.3 presents experimental results using such a library for both original AR and fixed cost routing.

### 3.2.3 Scalability and Performance

In Sections 3.2.1 and 3.2.2 two new methods for implementing AR were presented. This section compares their memory and computational efficiency on networks of many scales. Since both basic AR and fixed cost AR are limited by the default size of hardware integer variables (usually 32 bit or 64 bit), we denote by MP basic AR and MP fixed cost AR, implementations of basic AR and fixed cost AR that use a multiprecision arithmetic library rather than the 32 bit or 64 bit arithmetic that can be performed directly by the computer hardware. Using a multiprecision library allows us to overcome the node address explosion discussed in Sections 3.1.5 and 3.2.2.

It should be noted that when using the multiprecision library for node addresses, the amount of memory per address is not fixed. The memory used is dependent on the maximum node degree and the tree depth. The number of machine operations involved in a multiprecision operation is dependent on the size of the operands. This makes estimating the cost of MP AR more difficult.

Table 3.2 summarises the theoretical properties of the five methods. With these theoretical figures in mind let us examine the experimental results presented in Figure 3.9. Tests were performed on networks with one hundred to one million nodes. A fixed number ($10^7$) next hop lookups were performed. This tests the methods without the need to generate traffic. A 1.0GHz Pentium running FreeBSD 4.6, with code compiled using GCC 2.95, was used to run the tests.

| Algorithm | Memory | Operations per lookup |
|---|---|---|
| Basic | $\mathcal{O}(N)$ | $\mathcal{O}(\log N)$ divisions |
| Direct | $\mathcal{O}(0)$ | $\mathcal{O}(\log N)$ lookups |
| Fixed Cost | $\mathcal{O}(N)$ | 1 division, 1 multiplication |
| MP Basic | $< \mathcal{O}(N \log k^{d-1})$ | $\mathcal{O}(\log N)$ MP divisions |
| MP Fixed | $< \mathcal{O}(N \log k^{d-1})$ | 1 mp division, 1 mp multiplication |
| $k$ is the maximum node degree and $d$ is depth of tree. | | |

*Table 3.2: Memory and computational cost for several types of AR.*

The networks were all generated with GT-ITM [19] using the transit–stub model. This mimics the structure of the Internet by dividing nodes into core routing domains and periphery stub domains. Many properties, such as connectedness, of the generated networks depend quite sensitively on the parameters chosen. However the performance of the AR lookup depends only on the depth of

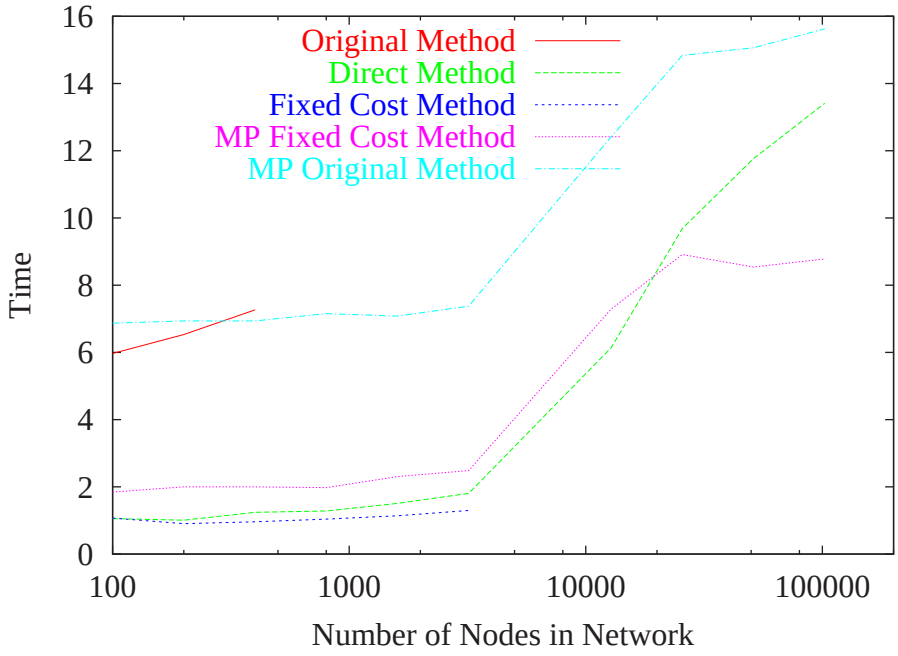



*Figure 3.9: The time taken (in seconds) to perform a fixed number (1e7) of nexthop calculations*

the routing tree, which varies less than other properties

The first item of note is that Figure 3.9 has three distinct regions: networks with up to four thousand nodes, networks with four to twenty thousand nodes and networks with more than twenty thousand nodes. For small networks fixed cost routing is fastest, but is followed very closely by direct routing. AR is several times slower. Both original and fixed cost routing are unable to deal with networks above a certain size: eight hundred nodes for the original method and three thousand for fixed cost. This is expected — the mapped node addresses overflow standard integers for these networks.

For these small networks, the original and direct methods quite clearly scale with $\mathcal{O}(\log N)$. The fixed cost method is not perfectly fixed, but the variation is quite low.

Something unexpected happens with medium sized networks. The relative performance of the three methods left remains the same, but all have a large jump in computational time. This, we believe, is due to memory cache effects. Above two thousand nodes the network cannot be held in cache and performance drops quite sharply — even for fixed cost routing.

Above twenty thousand nodes, the expected behaviour reasserts itself again as most of the network is stored outside of the memory cache. Direct and original AR scale with $\mathcal{O}(\log N)$ while fixed cost routing has roughly constant performance.

#### The best method?

At this point it is pertinent to ask which of the proposed methods is best. Unsurprisingly, there is no simple answer. Direct algorithmic routing has the advantage of having no extra memory requirements and scaling very well up to about forty thousand nodes (for this set of networks). Direct routing is also very simple. For small networks, where memory is not an issue, fixed cost routing has the best performance. For large networks the choice comes down to a speed/memory tradeoff. Multi precision fixed cost routing has a roughly constant performance, but does require extra memory, while direct AR requires no extra memory but is slower.

## 3.3 Route Length Improvement

Figure 3.4 demonstrated how the quality of routes generated by AR can depend on the order in which nodes are searched during BFS. In this section the behaviour is explored further and a method is suggested for improving the trees generated by BFS.

The choice of route node in a BFS tree influences both route quality and the work necessary for the next hop calculation. The following example demonstrates how the position of the root node affects the amount of work done. Consider the network of five nodes connected in a straight line, Figure 3.10a, with two choices, Figures 3.10b, 3.10c of root node for the BFS tree. In the next hop calculation, using the algorithm in Section 3.2.1, we ascend the tree from the destination node until we reach either the source or the root node. If the tree is shallow, Figure 3.10b, one of these nodes is encountered sooner than if the tree is deep, Figure 3.10c. For this reason it is beneficial to choose a node that will minimise the average depth, or to rebalance the tree after its creation.

The root node of the tree can be changed after creation without altering the routes generated by the tree. However, choosing a different root node at search time may create a different tree. All links emanating from the root node are present in the tree, but this is not necessarily true for other nodes.

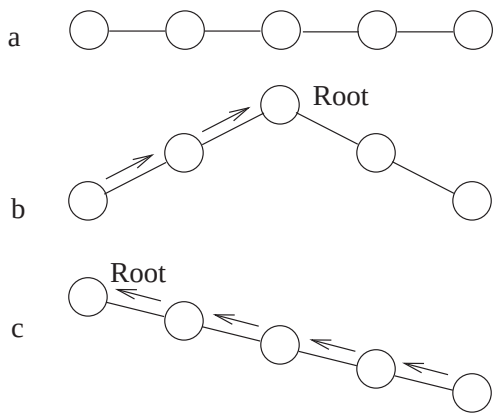



*Figure 3.10: It takes fewer hops to reach the root in a shallow tree than in a deep tree*

As before, $H$ is defined to be the sum total of the number of hops needed to join all distinct pairs of nodes in the network:

$$H = \sum_{i=1}^{N} \sum_{j=i+1}^{N} \text{Dist}(i, j). \tag{3.5}$$

Figure 3.11 shows the range that $H$ can take in a 100 node network. The nodes have been sorted in ascending order of $H$ and plotted against $H$. See Figure 3.12 for a diagram of the network. Table 3.3 lists the average and standard deviation of $\frac{H}{D}$ for several networks of different sizes. In all cases highly connected nodes were given precedence in the BFS creation of the routing tree. The relative quality of routes varies from between 1.1 and 1.27 times the shortest path lengths. Unfortunately there is no obvious way of choosing a root node and BFS order to minimise $H$.

| **Number of Nodes** | $\frac{E}{N}$ | **Average** $\frac{H}{D}$ | **Standard Deviation of** $\frac{H}{D}$ |
|---|---|---|---|
| 100 | 3.26 | 1.23 | 0.053 |
| 200 | 3.29 | 1.10 | 0.028 |
| 400 | 3.33 | 1.12 | 0.062 |
| 800 | 3.20 | 1.11 | 0.030 |
| 1600 | 3.21 | 1.16 | 0.069 |
| 3200 | 3.22 | 1.17 | 0.076 |
| 6400 | 3.25 | 1.27 | 0.092 |

*Table 3.3: A sample of $\frac{H}{D}$ values for several networks.*

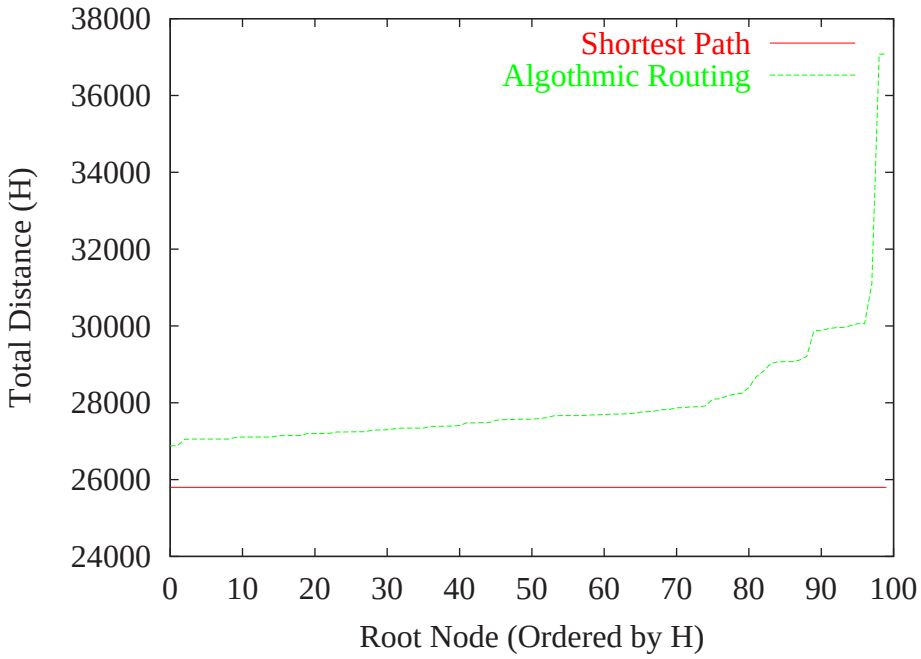



*Figure 3.11: A range of values of H. Each point represents the value of H for algorithmic routing starting at a different node and sorted in ascending order. The network is that shown in Figure 3.12*

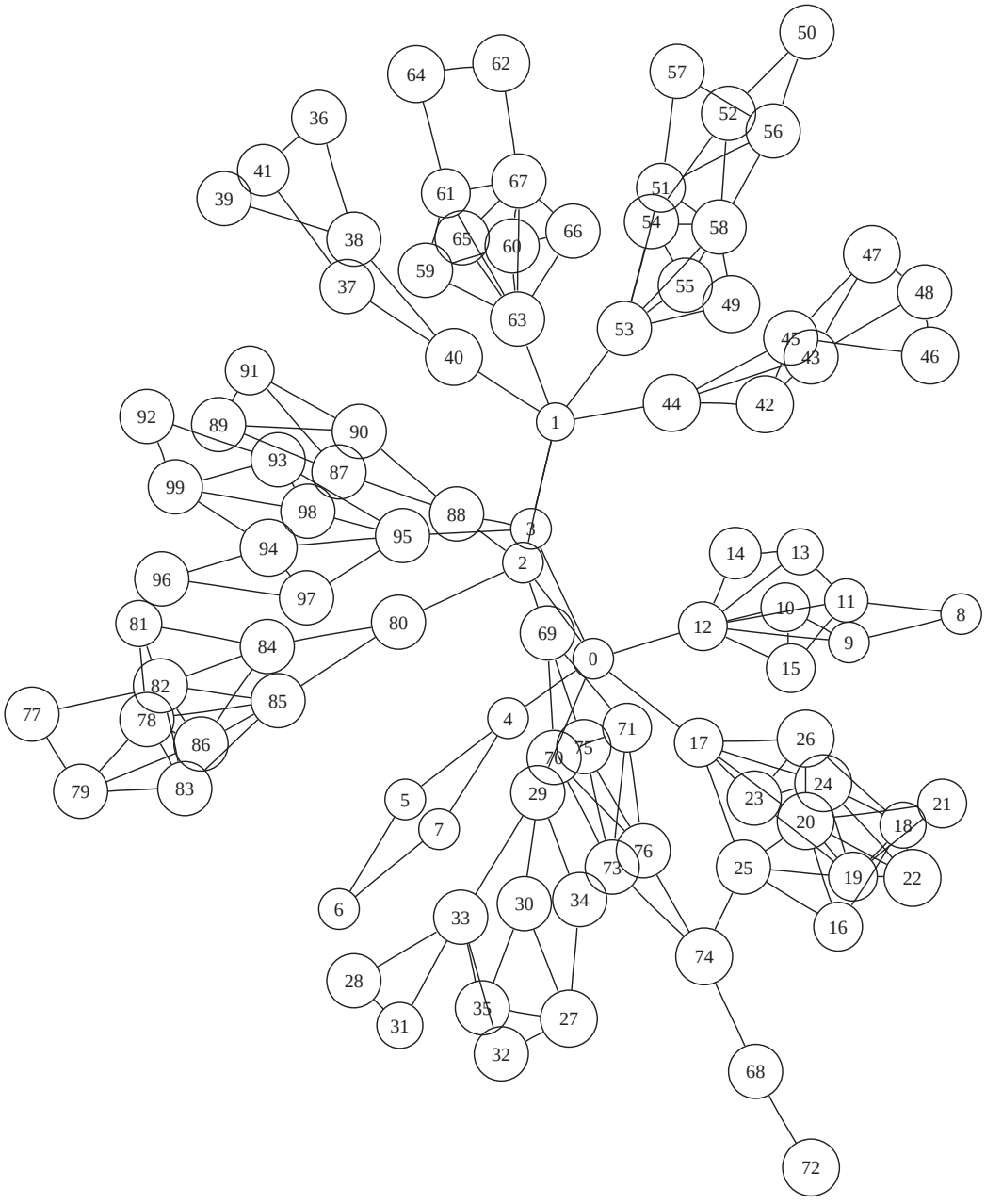



*Figure 3.12: A 100 node network generated by GT-ITM. Other similar networks, with between 10 and 100000 nodes were also generated and used for testing the accuracy and efficiency of AR.*

### Optimal Solutions of Shortest Path Spanning Tree

Creating an optimal tree for AR is a special case of a more general problem in graph theory, known as the optimal communication spanning tree (OCST) problem, proposed by Hu [45]. This can be formally defined as follows: a complete undirected graph $G$ is defined as $G = (V, E)$ where $V$ is a set of vertices and $E$ is a set of edges joining all pairs of vertices. The number of vertices is denoted by $n = |V|$. An edge between vertices $v_i$ and $v_j$ is denoted by $(v_i, v_j)$ A spanning tree of $G$ is defined as $T = (V, F)$ where $F \subseteq E$, $|F| = |V| - 1$ and all vertices are connected.

The OCST problem involves finding a spanning tree of $G$ that satisfies certain constraints. A *demand matrix* $R = (r_{ij})$ determines the amount of traffic between vertex pairs. $R$ is an $n \times n$ matrix, and $r_{ij}$ is the traffic required to flow between vertices $v_i$ and $v_j$. An $n \times n$ *distance matrix* $W = w_{ij}$ specifies the distance weight of each vertex pair. Let $P_{ij}(T) \subseteq F$ be the set of edges linking vertices $v_i$ and $v_j$ in tree $T$. The weight $w(T)$ of a spanning tree $T$ is defined as

$$w(T) = \sum_{v_i, v_j \in V} \left( r_{ij} \sum_{(v_p, v_q) \in P_{ij}(T)} w_{pq} \right).$$

A tree $T$ is a solution of the OCST problem if $w(T) \leq w(T')$ for all other spanning trees $T'$.

An optimal routing tree for AR is a solution to a special case of the OCST problem. Let $G_n$ be the graph corresponding to a network, with edges $E_n$ corresponding to the links in the network. Certain conditions are imposed on the OCST problem. These are: $|G| = |G_n|$, $r_{ij} = 1$ if $i < j$, $r_{ij} = 0$ if $i \geq j$, $w_{ij} = 1$ if $(v_i, v_j) \in E_n$ and $w_{ij} = \infty$ if $(v_i, v_j) \notin E_n$. As before $G$ is a complete undirected graph.

Finding an optimal AR tree is also a special case of the Shortest Total Path Length Spanning Tree problem. This problem has the same conditions as the optimal AR tree problem except that $w_{ij}$ is not restricted to $w_{ij} = 1$ if $(v_i, v_j) \in E_n$, but may assume any non–negative value. The problem is $\mathcal{NP}$–complete. Another related problem is $k$-source shortest paths spanning tree problem [25] [31].

The general OCST problem is $\mathcal{NP}$–complete [38]. Recent approaches to solving the OCST problem have concentrated on evolutionary algorithms [62]. However solutions to the general OCST problem have been limited to small graphs, typically

with fewer than twenty five vertices [87]. These approaches are not feasible for use with the large, but less general, graphs used in AR.

Recent work has concentrated on finding approximate algorithms for near optimal solutions. In particular Wu et al. [96], building on work of Bartal [10] [9], have demonstrated how to build a spanning tree with a cost within a factor of $(1+\epsilon)$ of the minimum, in time $\mathcal{O}(N^{\mathcal{O}(\frac{1}{\epsilon})})$. However, this approach is not suitable for large graphs as it requires the calculation of the all-pairs shortest paths of the graph. For a large graph, this is not computationally feasible.

We now propose a $\mathcal{O}(N \log N)$ method for generating a tree for AR. Unlike the method above, it does not guarantee that $H$ will be within a bound, but does work on large graphs. It is implemented as an iterative algorithm. At each step an alteration is made to the tree. $H$ is measured. If $H$ has decreased the alteration is kept, otherwise it is reversed.

### Types of modification

The network itself cannot be modified. All that is changed is the mapping to a tree. If the network contains cycles then there is more than one way to map it to a tree, and this may affect the value of $H$: consider the small network in Figure 3.13a. This can be mapped by a BFS to either Figure 3.13b or Figure 3.13c. Both are valid trees, but have very different values of $H$. However by making node $z$ rather than node $y$ the parent of node $x$ we can convert one tree to the other. These are the modifications we will test — changing which links are broken by the mapping to a tree. However the change must not introduce a cycle in the tree, or result in a tree that is not fully connected, Figure 3.14.

As before $G(V, E)$ is the network graph and $(T, r)$ is a routing tree where $T$ is a spanning tree and $r$ is the root node. $\mathcal{S}(T, a)$ is the set of nodes in the subtree of $T$ rooted at $a$. Let $x, y, z \in V$ and let $y$ be a child of $x$

We denote a modification to the tree by

$$T_{\text{new}} = T/\{(x, y)\} \cup \{(z, y)\}$$

where $z \notin \mathcal{S}(T, x)$. We write $H(T)$ or $H(T_{\text{new}})$ to specify the value of $H$ using $H(T)$ or $H(T_{\text{new}})$ respectively.

We propose two types of iterative improvement. The first approach tests each legal modification in the network and accepts or rejects each one immediately.

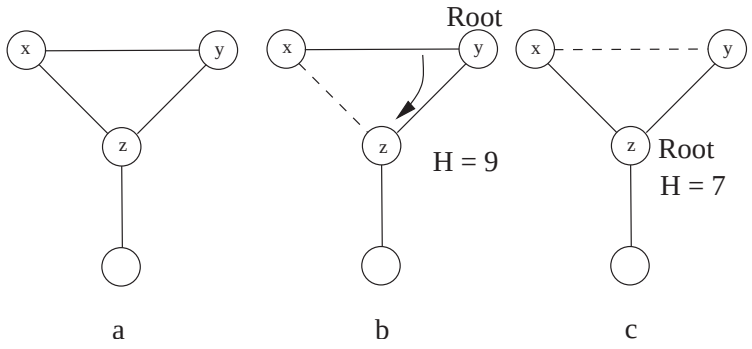



*Figure 3.13: Two different choices of root node result in trees with different values of $H$. However by making $z$ the parent of $x$ rather than $y$ in tree (b) we can convert one tree to the other.*

The second examines a node, reconnects all possible broken links and accepts or rejects the change based on the overall change in $H$, and repeats this for every node. The two methods are shown in Algorithms 4 and 5 respectively.

Results showing the improvement in route quality after applying this method are presented in Section 3.3.1.

**Input:** A graph $G = (V, E)$. A routing tree $T = (V, F, r)$ where $F \subseteq E$. The sum of distances between all unordered node pairs in tree $T$ is written $H(T)$.
**Output:** A routing tree $T_{\text{new}}$ with $H(T_{\text{new}}) \leq H(T)$.

(1) $T_{\text{test}} \leftarrow T$
(2) $T_{\text{new}} \leftarrow T$
(3) **foreach** $z \in V$
(4) **foreach** $(x, y) \in F_{\text{new}}$ s.t. $(x, z) \in E$ and $z \notin \mathcal{S}(x, T_{\text{new}})$
(5) $T_{\text{test}} \leftarrow T_{\text{new}} / \{(x, y)\} \cup \{(x, z\}$
(6) **if** $H(T_{\text{test}}) \leq H(T_{\text{new}})$
(7) $T_{\text{new}} \leftarrow T_{\text{test}}$
(8) **return** $T_{\text{new}}$

*Algorithm 4: An iterative method for modifying a routing tree so as to lower $H$, the sum of distances between node pairs. Each modification is tested immediately and either accepted or rejected.*

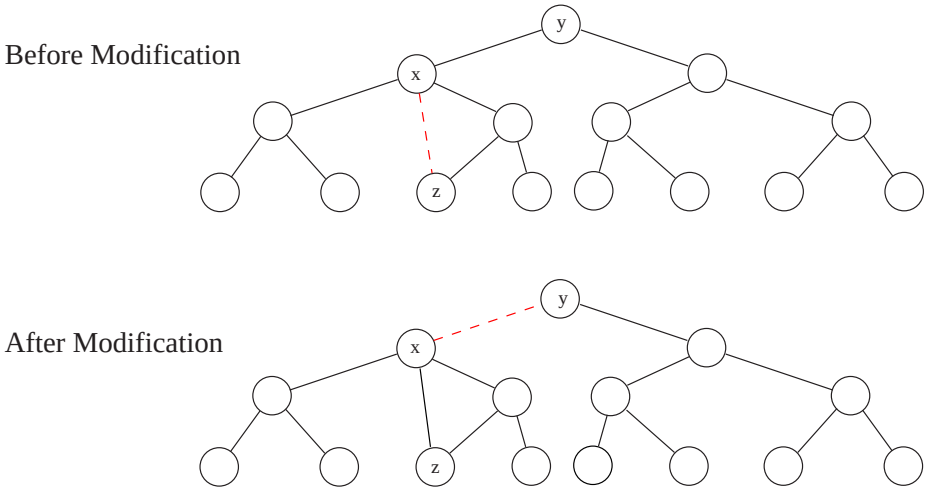



*Figure 3.14: An illegal modification to a tree. Since $z$ is a descendant of $x$, if we make $z$ the parent of $x$ as well, we end up with both a cycle in the tree, and two disjoint islands.*

**Input:** A graph $G = (V, E)$. A routing tree $T = (V, F, r)$ where $F \subseteq E$. The sum of distances between all unordered node pairs in tree $T$ is written $H(T)$.
**Output:** A routing tree $T_{\text{new}}$ with $H(T_{\text{new}}) \leq H(T)$.

(1) $T_{\text{test}} \leftarrow T$
(2) $T_{\text{new}} \leftarrow T$
(3) **foreach** $z \in V$
(4) **foreach** $(x, y) \in F_{\text{new}}$ s.t. $(x, z) \in E$ and $z \notin \mathcal{S}(x, T_{\text{new}})$
(5) $T_{\text{test}} \leftarrow T_{\text{new}} / \{(x, y)\} \cup \{(x, z\}$
(6) **if** $H(T_{\text{test}}) \leq H(T_{\text{new}})$
(7) $T_{\text{new}} \leftarrow T_{\text{test}}$
(8) **return** $T_{\text{new}}$

*Algorithm 5: An iterative method for modifying a routing tree so as to lower $H$, the sum of distances between node pairs. Several modifications are tested at a time and either accepted or rejected on the basis of the overall change in $H$.*

### 3.3.1 Measurement of $H$

Each time the tree is modified in the route improvement algorithm, it is necessary to measure the change in $H$, so as to decide whether to accept or reject the change. In a network with $l = |E| - |V| - 1$ broken links the calculation is performed $\mathcal{O}(Nl)$ times and this is a significant cost if $l$ is large. In order for the algorithm to scale with $N$, we need to be able to measure the change in $H$ in at worst $\mathcal{O}(\log(N))$ time at each step.

In this section we propose a method for performing an initial calculation of $H$ in $\mathcal{O}(N \log(N))$ time, and a method for updating $H$ in $\mathcal{O}(\log N)$ time. It has memory requirements of $\mathcal{O}(N)$.

First some definitions. Let $d(x, y)$ be the distance between nodes $x$ and $y$ along the path chosen by AR. For two sets of nodes $\mathcal{X}$ and $\mathcal{Y}$ let $D(\mathcal{X}, \mathcal{Y})$ be the sum of the lengths of paths joining each pair of nodes $x$, $y$, s.t $x \in \mathcal{X}$, $y \in \mathcal{Y}$ and $x \neq y$. More formally:

$$D(\mathcal{X}, \mathcal{Y}) = \sum_{x \in \mathcal{X}, y \in \mathcal{Y}} d(x, y) - \frac{1}{2} \sum_{x, y \in \mathcal{Y} \cap \mathcal{X}} d(x, y) \tag{3.6}$$

The second term in this expression compensates for nodes that are in both $\mathcal{X}$ and $\mathcal{Y}$, and are counted twice in the first term. Using this notation, if $V$ is the set of all vertices in the network graph then $H = D(V, V)$.

We wish to be able to quickly update $H$ when a subtree rooted at $a$ with a parent $b$ and broken link to $c$, has its link to $b$ broken and has $c$ made its parent (provided that this modification results in a fully connected tree).

Let $\mathcal{S}(a, T)$ be the set of nodes in the subtree of tree $T$ rooted at $a$. Let $\mathcal{S}'(a, T) = V \backslash \mathcal{S}(a, T)$. Let $T_1$ be a routing tree for the network. For conciseness we write $\mathcal{S}_1 = \mathcal{S}(a, T_1)$ and $\mathcal{S}'_1 = \mathcal{S}'(a, T_1)$. Let $a$ be a node with parent $b$ in $T_1$. Note that

$$(V, V) = (\mathcal{S}_1, \mathcal{S}_1) \cup (\mathcal{S}_1, \mathcal{S}'_1) \cup (\mathcal{S}'_1, \mathcal{S}_1) \cup (\mathcal{S}'_1, \mathcal{S}'_1).$$

Now

$$\begin{aligned} H(T_1) &= \frac{1}{2} \sum_{x, y \in V} d(x, y) \\ &= \frac{1}{2} \left[ \sum_{x, y \in \mathcal{S}_1} d(x, y) + 2 \sum_{x \in \mathcal{S}_1, y \in \mathcal{S}'_1} d(x, y) + \sum_{x, y \in \mathcal{S}'_1} d(x, y) \right]. \end{aligned} \tag{3.7}$$

Since $\mathcal{S}_1$ and $\mathcal{S}'_1$ are disjoint sets, if $x \in \mathcal{S}_1$ and $y \in \mathcal{S}'_1$ then

$$d(x,y) = d(x,a) + d(a,b) + d(b,y) = d(x,a) + 1 + d(b,y) \tag{3.8}$$

and

$$\begin{aligned}
\sum_{x\in\mathcal{S}_1, y\in\mathcal{S}'_1} d(x,y) &= N(\mathcal{S}_1)N(\mathcal{S}'_1) + \sum_{x\in\mathcal{S}_1, y\in\mathcal{S}'_1} (d(x,a) + d(b,x)) \\
&= N(\mathcal{S}_1)N(\mathcal{S}'_1) + N(\mathcal{S}'_1)\sum_{x\in\mathcal{S}_1} d(x,a) + N(\mathcal{S}_1)\sum_{y\in\mathcal{S}'_1} d(b,y) \\
&= N(\mathcal{S}_1)N(\mathcal{S}'_1) + N(\mathcal{S}'_1)D(\{a\},\mathcal{S}_1) + N(\mathcal{S}_1)D(\{b\},\mathcal{S}'_1) \qquad (3.9)
\end{aligned}$$

Using Equations 3.6, 3.7 and 3.9 we can write

$$\begin{aligned}
H(T_1) &= D(\mathcal{S}_1,\mathcal{S}_1) + D(\mathcal{S}'_1,\mathcal{S}'_1) + N(\mathcal{S}_1)N(\mathcal{S}'_1) \\
&\quad + N(\mathcal{S}'_1)D(\{a\},\mathcal{S}_1) + N(\mathcal{S}_1)D(\{b\},\mathcal{S}'_1). \qquad (3.10)
\end{aligned}$$

Let $T_2$ be a modification to $T_1$ such that $a$ has parent $c$, where $c \notin \mathcal{S}_1$. This implies that $\mathcal{S}_2 = \mathcal{S}(a,T_2) = \mathcal{S}_1$. Denote by $H(T_1)$ and $H(T_2)$ the distance function of the network when using trees $(T_1)$ and $T_2$ respectively. Now $H(T_2)$ can be expressed similarly to Equation 3.10

$$\begin{aligned}
H(T_2) &= D(\mathcal{S}_1,\mathcal{S}_1) + D(\mathcal{S}'_1,\mathcal{S}'_1) + N(\mathcal{S}_1)N(\mathcal{S}'_1) \\
&\quad + N(\mathcal{S}'_1)D(\{a\},\mathcal{S}_1) + N(\mathcal{S}_1)D(\{c\},\mathcal{S}'_1). \qquad (3.11)
\end{aligned}$$

and by combining Equations 3.10 and 3.11

$$\Delta H = H(T_2) - H(T_1) = N(\mathcal{S}_1)\left[D(\{c\},\mathcal{S}'_1) - D(\{b\},\mathcal{S}'_1)\right] \tag{3.12}$$

We need simple expressions for these final two terms:

$$\begin{aligned}
D(\{b\},\mathcal{S}'_1) &= D(\mathcal{S}_1 \cup \mathcal{S}'_1, \{b\}) - D(\mathcal{S}_1, a) - N(\mathcal{S}_1) \\
D(\{c\},\mathcal{S}'_1) &= D(\mathcal{S}_1 \cup \mathcal{S}'_1, \{c\}) - N(\mathcal{S}_1)d(b,c) - N(\mathcal{S}_1) - D(\mathcal{S}_1, a)
\end{aligned}$$

We get:

$$\Delta H = N(\mathcal{S}_1)\left[D(V,c) - D(V,b) - N(\mathcal{S}_1)d(b,c)\right] \tag{3.13}$$

The last step is to show how to calculate these values, and most importantly, how to efficiently update them as the graph is modified.

### Implementation

At each iteration we measure the $\Delta H$ of the proposed change to the network tree. This modification consists of moving a subtree rooted at $x$ from $y$ to $z$, where $y$ is the parent of $x$ and the link between $x$ and $z$ is broken — this change must also result in a connected tree. Figure 3.15 is an example.

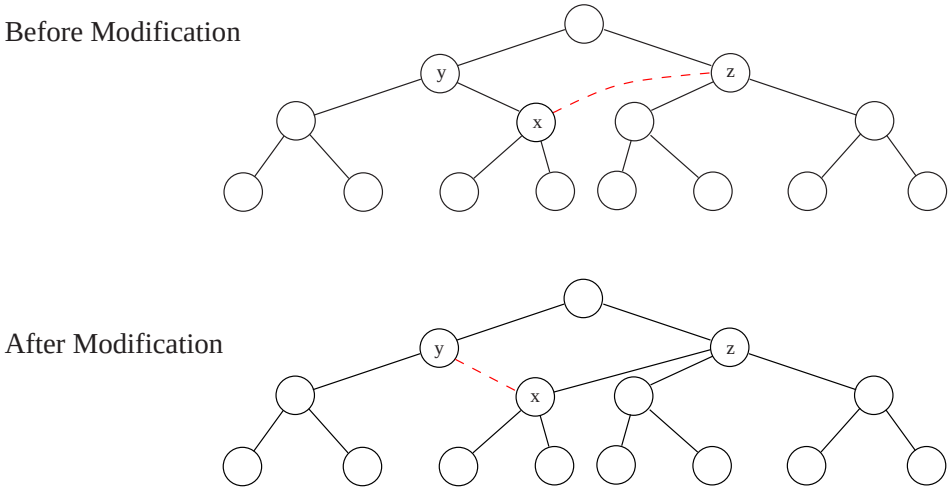



*Figure 3.15: An example of a legal modification to a tree, in which $z$ replaces $y$ as the parent of $x$. The dashed line is a broken link.*

In order to calculate both $H$ and $\Delta H$ we store two values for each node $x$: $N_x$ and $D_x$. If $\mathcal{S}(x,T) = \mathcal{S}_x$ is the set of nodes in the subtree rooted at $x$ then let $N_x = |\mathcal{S}_x|$ and $D_x = D(x, \mathcal{S}_x)$. We can build $N_x$ and $D_x$ for all $x \in V$ using Algorithm 6. The relationship between $N_x$, $D_x$ and $N_y$, $D_y$ of connected nodes $x$ and $y$ is illustrated in Figures 3.16 and 3.17.

Now recall that $D_x = D(x, \mathcal{S}_x)$. Therefore

$$D(x,V) = D_x + D(x, V \backslash \mathcal{S}_x) \tag{3.14}$$

and if $y = \text{Parent}(x)$ we can write $D(x, V\backslash\mathcal{S}_x)$ as

**Input:** A routing tree $T = (V, F)$.
**Output:** $N_x$ and $D_x$ for all nodes $x \in V$.
(1) **foreach** $x \in V$
(2) $N_x \leftarrow 0$
(3) $D_x \leftarrow 0$
(4) **foreach** $x \in V$
(5) $y \leftarrow x$
(6) $d \leftarrow 0$
(7) **repeat**
(8) $N_y \leftarrow N_y + 1$
(9) $T_y \leftarrow T_y + d$
(10) $d \leftarrow d + 1$
(11) $y \leftarrow \text{Parent}(y)$
(12) **until** $y = \text{root}$
(13) **return** $N_x$ and $D_x \; \forall \; x \in V$

*Algorithm 6: Construction of $D_x$ and $N_x$.*

$$D(x, V\backslash\mathcal{S}_x) = D(y, V\backslash\mathcal{S}_y) + N - N_x + D_y + N_x - D_x - 2N_x \quad (3.15)$$
$$= D(y, V\backslash\mathcal{S}_y) + N + D_y - D_x - 2N_x \quad (3.16)$$

This is a recursive definition for any $x \neq$ root. The recursion terminates when $y$ is the root node:

$$D(y, V\backslash\mathcal{S}_y) = D(\text{root}, V\backslash V) = 0 \quad (3.17)$$

We can generate $D(x, V\backslash\mathcal{S}_x)$ using Algorithm 7.

**Input:** A routing tree $T = (V, F)$ and a node $x \in V$.
**Output:** The value of $d = D(\{x\}, V\backslash\mathcal{S}_x)$ where $\mathcal{S}_x$ is the set of nodes in the subtree rooted at $x$.
(1) $d \leftarrow 0$
(2) **repeat**
(3) $y \leftarrow \text{Parent}(x)$
(4) $d \leftarrow d + N + D_y - D_x - 2N_x$
(5) $x \leftarrow y$
(6) **until** $y = \text{root}$
(7) **return** $d$

*Algorithm 7: Calculation of $D(\{x\}, V\backslash\mathcal{S}_x)$.*

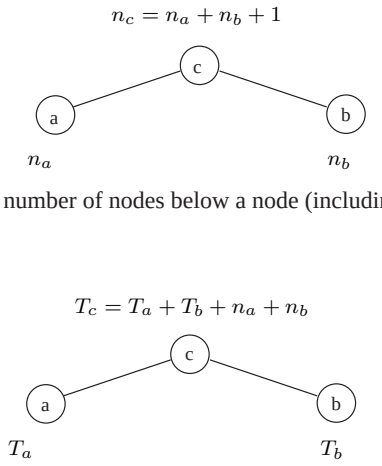



*Figure 3.16: Relationship of $D_x$ and $N_x$ between nodes in a tree.*

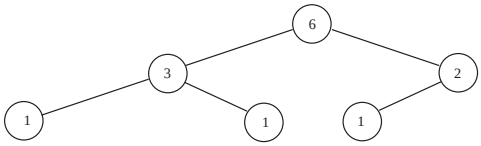
Once $D_x$ and $N_x$ for all $x \in V$ are set up we can generate $D(x, V)$ using Equations 3.16 and 3.17. With this we have all that is necessary to calculate the $\Delta H$ of a proposed modification to the tree. If the modification is accepted we then need to update $D_x$ and $N_x$ for the network. The first step is to detach the subtree at $x$ from $b$ and update the nodes above $x$ as in Algorithm 8. The second

**Input:** A routing tree $T(V, F, r)$ and a node $x$. $N_y$ and $D_y$ for all nodes $y \in V$.
**Output:** Updated values of $N_y$ and $D_y$ for all nodes $y \in V$ when $x$ is detached from its parent.
(1) $y \leftarrow \text{Parent}(x)$
(2) $d \leftarrow D_x$
(3) **repeat**
(4) $d \leftarrow d + N_x$
(5) $D_y \leftarrow D_y - d$
(6) $N_y \leftarrow N_y - N_x$
(7) $y \leftarrow \text{Parent}(y)$
(8) **until** $y = \text{root}$
(9) **return** $N_y$ and $D_y \; \forall \; y \in V$

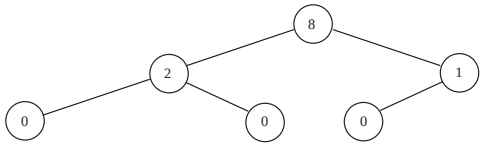
*Algorithm 8: When $x$ is detached from its parent the values of $N_y$ and $D_y \; \forall \; y \in V$ must be updated.*

step is to reattach the subtree to its new parent $p$, and update the nodes above it, Algorithm 9.

If two or more modifications are to be made simultaneously then the situation




Total number of nodes below a node (including itself)




Sum of distances from a node to all nodes below

*Figure 3.17: Example of $D_x$ and $N_x$ between nodes in a tree.*

**Input:** A routing tree $T(V, F, r)$ and a node $x$. $N_y$ and $D_y$ for all nodes $y \in V$.
**Output:** Updated values of $N_y$ and $D_y$ for all nodes $y \in V$ when $x$ is attached to a new parent.

(1) $y \leftarrow p$
(2) $d \leftarrow T_x$
(3) **repeat**
(4) $N_y \leftarrow N_y + N_x$
(5) $d \leftarrow d + N_x$
(6) $T_y \leftarrow T_y + d$
(7) $y \leftarrow \text{Parent}(y)$
(8) **until** $y \leftarrow$ root
(9) **return** $N_y$ and $D_y \; \forall \; y \in V$

*Algorithm 9: When detached subtree rooted at $x$ is reattached to a new parent the values of $N_y$ and $D_y \; \forall \; y \in V$ must be updated.*

is more complicated. As we make each individual change we calculate the $\Delta H_i$ of that change, and base our final decision on the sum of these $\Delta H_i$, Algorithm 10.

### Efficiency of Algorithm

There are several stages in determining the initial values of $H$, calculating $\Delta H$ and in updating the network after a successful modification. The initialisation of $N_x$ and $D_x$ using Algorithm 6 has complexity $\mathcal{O}(N \log N)$ and requires $\mathcal{O}(N)$ memory. The calculation of $D(x, V)$ has $\mathcal{O}(\log N)$ complexity. Finding the distance between two nodes $x$ and $y$, $d(x, y)$, has complexity $\mathcal{O}(\log N)$ (depending on the next hop algorithm used). So, overall, one modification of the network requires $\mathcal{O}(\log N)$

**Input:** A routing tree $T(V, F, r)$. A number of modifications of the form $T_{\text{new}} = T / \{(x_i, y_i)\} \cup \{(z_i, y_i\}$ where $0 < i < n$.
**Output:** A new tree $T_{\text{new}}$ with $H(T_{\text{new}}) \leq H(T)$.

```
(1)   ΔH ← 0
(2)   T_test ← T
(3)   foreach  i ∈ {1...n}
(4)      T_test ← T_test − {(x_i, y_i)} ∪ {(z_i, y_i}
(5)   ΔH ← H(T) − H(T_test
(6)   if  ΔH > 0
(7)      T_new ← T
(8)   else
(9)      T_new ← T_test
(10)  return T_new
```

*Algorithm 10: A method for making several modifications to a tree and accepting or rejecting them based on the overall change in $H$.*

time.

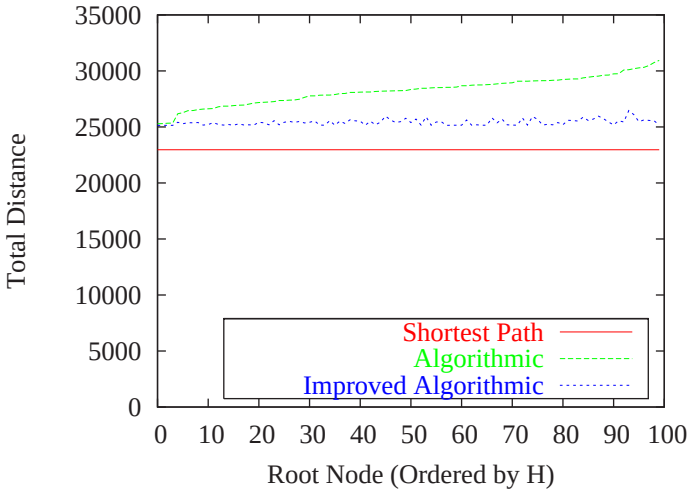
### Improvements in $H$

Figures 3.18, 3.19 and 3.20 present the results of performing the improvement algorithms on an initial BFS tree. We tested three approaches for each network. It should be noted that the improvements in $H$ discussed below will vary considerably from network to network. These results should not be taken as indicative of all networks of the given size, but rather as examples to demonstrate some typical behaviour and scaling properties. Later in the chapter we will use more realistic networks taken from Internet surveys and modern network generation tools.

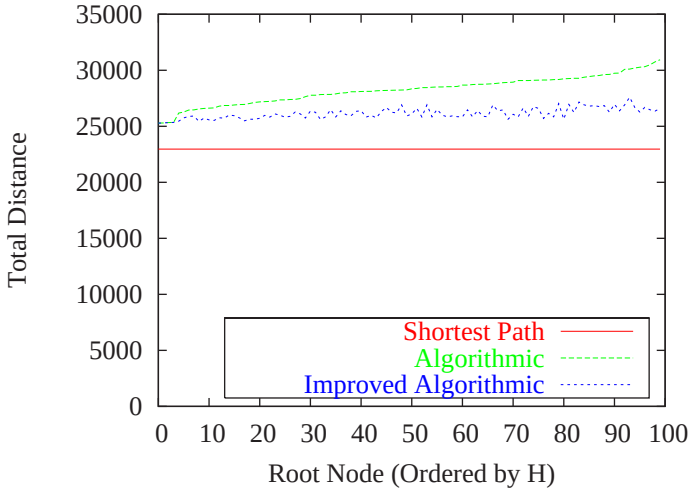
It is interesting to note that there is little variation in local minima of $H$ between different starting points, suggesting that the evolutionary approach to tree generation used in the general OCST problem would not be particularly useful in AR tree generation, even if it were possible on large networks.

Method one is that of Algorithm 4 and is shown in Figures 3.18a, 3.19a and 3.20a. We iterate through each node in the tree, make that node the parent of any neighbours to which it is connected by a broken link and accept or reject the modification immediately. The overall improvement depends on the initial tree, which in turn depends on the root node chosen. Additionally the final value of $H$ varies with the initial tree and root node. $H$ has a minimum value with occasional local maxima. These maxima occur when two or more modifications

a)




b)




c)

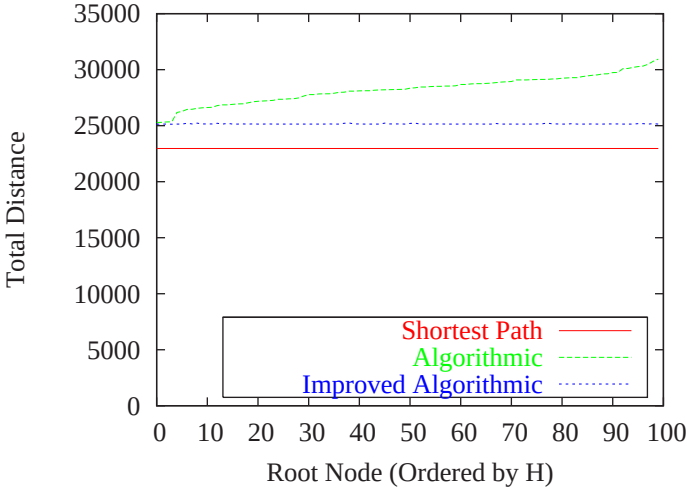
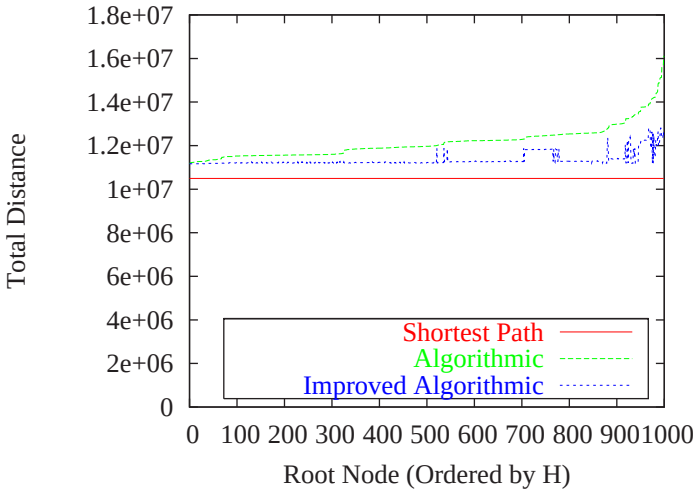
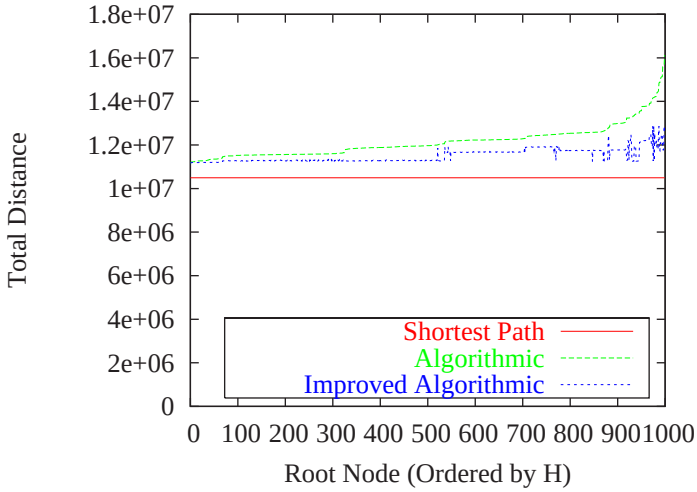



*Figure 3.18: A one hundred node network. The graphs show the improvement in $H$. Each point on the $x$ axis represents a different choice of root node. a) the tree is modified one link at a time. b) all broken links of a node are updated simultaneously and the combined change rejected or accepted. c) the above two steps are both applied.*

a)




b)




c)

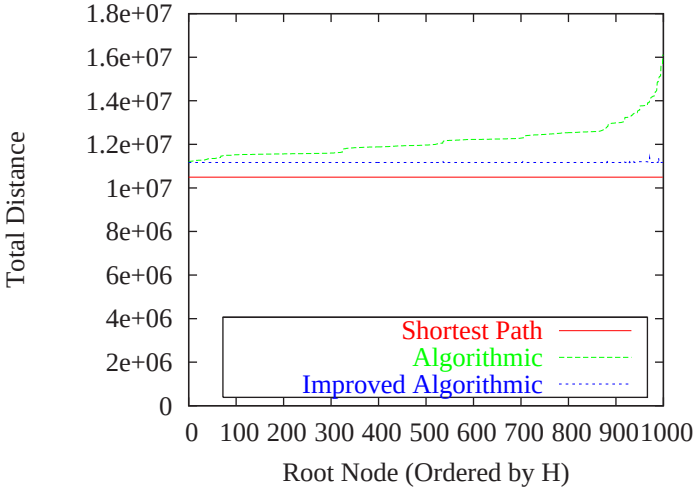
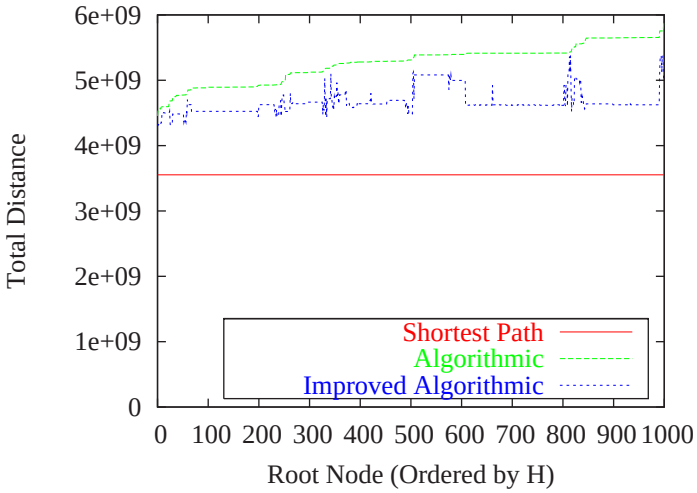
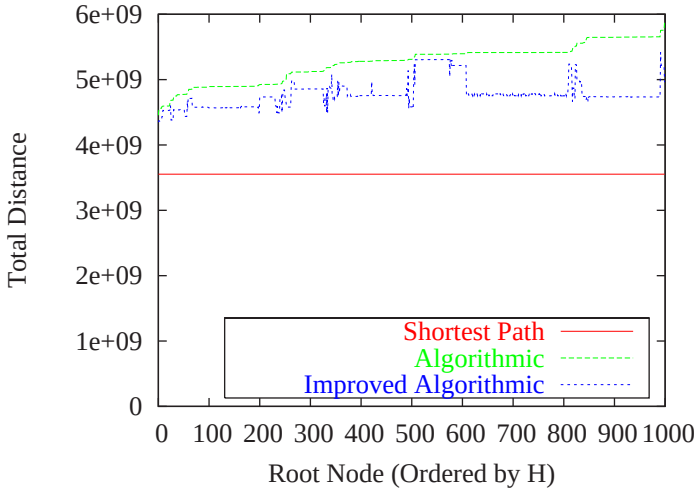



*Figure 3.19: A 1600 node network. The graphs show the improvement in $H$. Each point on the $x$ axis represents a different choice of root node. a) the tree is modified one link at a time. b) all broken links of a node are updated simultaneously and the combined change rejected or accepted. c) the above two steps are both applied. The improvement process is sometimes trapped in local minima.*

a)




b)




c)

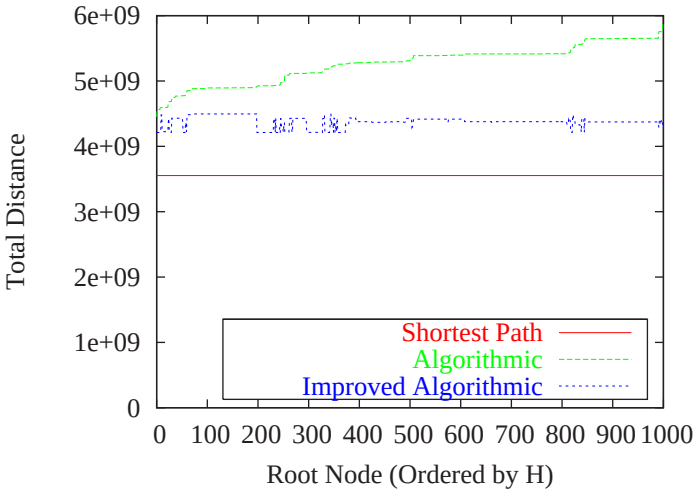



*Figure 3.20: A 25600 node network. The graphs show the improvement in $H$. Each point on the $x$ axis represents a different choice of root node. a) the tree is modified one link at a time. b) all broken links of a node are updated simultaneously and the combined change rejected or accepted. c) the above two steps are both applied. The improvement process is sometimes trapped in local minima.*

taken together cause a decrease in $H$, but individually cause an increase and so are rejected by Algorithm 4.

Method two uses Algorithm 5. The results are plotted in Figures 3.18b, 3.19b and 3.20b. In this case a node becomes parent to all possible neighbouring nodes and the overall modification to the tree is accepted or rejected. The results are not substantially different to those of the first method. Local minima of $H$ still occur.

Method three is more interesting. It is a combination of the first two methods. The first method is applied, then the second and finally the first method is applied a second time. The results, in Figures 3.18c, 3.19c and 3.20c are noticeably better. There are far fewer local minima.

On average, for these networks, the improvement algorithms reduce the value of $H$ by between 6 and 17 percent. While this is not a huge improvement, the cost is so small that it is worthwhile to apply the improvement process, at least once.

### Performance of Algorithm

The algorithm for determining and updating $H$ is fast. It has complexity of $\mathcal{O}(N \log N)$. To emphasise the importance of being able to efficiently update $H$, consider the case in which the distance between nodes is directly measured. It takes $\mathcal{O}(\log N)$ work to find the distance between two nodes. This must be calculated $N(N-1) \approx N^2$ times, for an overall cost of $\mathcal{O}(N^3 \log N)$. This clearly does not scale, even if it were to be reduced to $\mathcal{O}(N^2 \log N)$ by only updating the routes that are modified. The 12800 node network, which can be improved in 1.2 seconds with the $\mathcal{O}(N \log N)$ method would take at least four hours with a $\mathcal{O}(N^2 \log N)$ method.

The times taken for improving the tree using Algorithm 4 or Algorithm 5 show no significant differences.

Table 3.4 contains the times taken to improve the BFS tree for several different sized networks. These are taken from a single measurement and illustrate the order of magnitude time, rather than precise performance figures. A 1.0GHz Pentium running FreeBSD 4.6, with code compiled using GCC 2.95, was used to run the tests.

The times are graphed against network size in Figure 3.21 for the combined method, and a plot of $kN \log N$ is fitted. The theory and experimental results match very closely.

| Size of Network | Time for Method 1 (seconds) | Time for Method 2 (seconds) | Time for Method 3 (seconds) |
|---|---|---|---|
| 100 | 0.0021 | 0.0020 | 0.0055 |
| 200 | 0.0043 | 0.0044 | 0.013 |
| 400 | 0.010 | 0.0098 | 0.028 |
| 800 | 0.021 | 0.020 | 0.060 |
| 1600 | 0.044 | 0.044 | 0.12 |
| 3200 | 0.095 | 0.095 | 0.27 |
| 6400 | 0.20 | 0.20 | 0.57 |
| 12800 | 0.43 | 0.41 | 1.17 |
| 25600 | 0.90 | 0.84 | 2.40 |

*Table 3.4: Time taken to improve routing tree quality using the three different methods discussed in Section 3.3.1.*

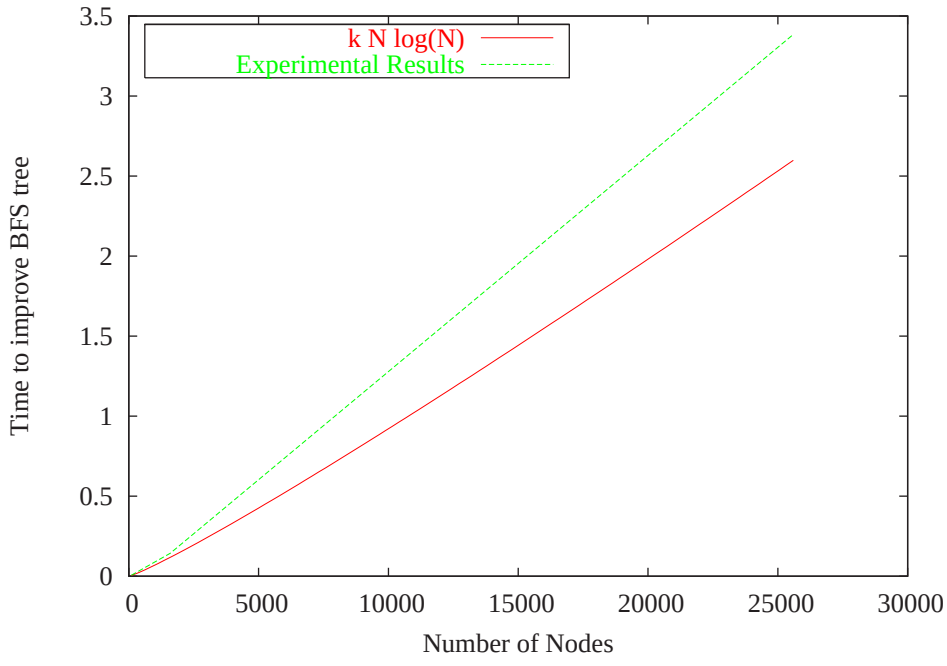



*Figure 3.21: A fit of the experimental times to improve BFS trees against the theoretical time of $\mathcal{O}(N \log N)$.*

## 3.4 Quality Improvements

Previous sections in this chapter have concentrated on two aspects of AR. The first consideration was the performance and scalability of the algorithm. It is important to note that, though new algorithms were developed for calculating the next hop in a path between two nodes, the actual paths remain the same for a given network and tree. The second consideration was the generation of the $k$-nary trees that provide the highest quality routes.

This section is concerned with improving the quality of routing beyond that which is possible by modifying the routing tree. In particular the issues of route length and link utilisation are examined.

Consider a network of $N$ nodes and $E$ links. In a connected network $E \geq N-1$. For example in the SCAN [40] [51] map of Internet routers $N = 228298$ and $E = 320203$. A tree generated from a connected $N$ node network has exactly $N-1$ links. Each link connects a child node to a parent node one level higher in the tree. Every node, apart from the root, has a parent node. Let $E_{\text{tree}}$ be the number of links in the tree generated from a network of $N$ nodes and $E$ links. Then $E_{\text{tree}} = N-1 \leq E$. This results in two differences between AR and shortest path routing. First, since only $N-1$ links are present in the tree, $E-N+1$ links will be completely unused by AR. All the traffic will be concentrated in the remaining $N-1$ links, leading to a greater utilisation of these links and possibly greater congestion. The second difference, which has been discussed in previous sections, is that the average path length may increase. This is an inaccuracy in itself, and also increases packet delay times. In addition, since packets spend longer in the system, congestion is further increased. Finally, since there is only one path between a pair of nodes, all routes are symmetric. While in real networks most routes may be symmetric, this is not necessarily true for all routes.

Huang and Heidemann [47] proposed several schemes for mitigating the worst of these effects, as described in Section 3.1.6. The new routing scheme developed in this section generalises and extends some of these ideas.

### 3.4.1 Multiple Tree Routing

AR with a single tree can only utilise $N-1$ of the $E$ links in a network and has effects on link congestion and packet delays, as discussed above. Using multiple trees may spread the utilisation of links and shorten route lengths. However the

| Number of Nodes, $N$ | Fraction of links used in 95% or more trees. | Fraction of links used in 5% or fewer trees. |
|---|---|---|
| 100 | .33 | .31 |
| 200 | .57 | .30 |
| 1000 | .62 | .28 |
| 10000 | .63 | .29 |

*Table 3.5: Certain links are highly likely to be present in a BFS routing tree, no matter which node is the root, while others are likely not to absent.*

choice of trees is vital to the success of this approach. Clearly building a tree for every source node would provide shortest path routing, and high link utilisation, but would be even more wasteful than maintaining a flat routing table. The aim is to maximise link utilisation and minimise route lengths while maintaining the lowest possible number of routing trees.

In our new approach several trees are generated, with as little overlap as possible. A packet uses the tree that provides the shortest path between source and destination. The calculation of the distance between nodes takes $\mathcal{O}(\log N)$ time. The procedure is simple. From both nodes ascend the tree to the root. Discard nodes visited twice. The distance is then the number of nodes remaining (including the source nodes) plus one. As an aside, this distance measurement could be performed in constant time with a LCA algorithm and the fixed cost routing scheme of Section 3.2.2, in which the depth in the tree is already stored.

If this calculation is performed for every packet, it would make routing expensive. However if it is done only once at the beginning of a session between two nodes the cost is relatively low — no more than the cost of a single next hop calculation.

The key to the success of this scheme is the choice of routing trees. Simply generating $t$ trees using BFS will not guarantee that significantly more links will be used, as the figures in Table 3.5 demonstrate. Given a network of $N$ nodes, a tree rooted at each node is generated. The second column in the table contains the fraction of links used in more than 95% of the $N$ routing trees. The third column contains the fraction of links used in less than 5% of the $N$ trees.

In these cases at least, tree structure does not differ considerably depending on root node, as most links remain either in or out of the tree irrespective of root node. A new method for generating trees is needed if using multiple trees is to

increase link utilisation.

The mechanics developed in Section 3.3.1 will be used again. Recall that

$$\Delta H = N(\mathcal{S}(a,T))\left[D(V,c) - D(V,b) - N(\mathcal{S}(a,T))d(b,c)\right] \tag{3.18}$$

where $V$ is the set of all nodes, $N(\mathcal{S}(a,T))$ is the number of nodes in a subtree rooted at $a$, $D(V,a)$ is the sum of the distances from every node to node $a$ and $d(b,c)$ is the distance between nodes $b$ and $c$. $\Delta H$ is the change in the value of $H$ in an algorithmic routing tree when the subtree rooted at node $a$ is disconnected from node $b$ and reconnected to node $c$.

In Section 3.3 the value of $\Delta H$ was used as a test to accept or reject the modification of a tree. The criteria are now different. The task is not to generate the single best tree, but to maximise the number of links used while maintaining trees of a reasonable quality. The first tree can be generated normally, as described in Section 3.3. Subsequent trees are initially generated by a BFS and modified according to the following criteria:

- Each tree should have a different root to maximise diversity of the tree structures.
- Modifications to the tree should result in the use of a previously unused link.
- Modifications that have a positive $\Delta H$ (decrease in quality) should be sometimes accepted in order to increase the diversity of the trees.

The first two items in this list are easily understood and implemented. The third item, the acceptance of modifications with positive $\Delta H$ requires more care. Consider again Equation 3.18. This can be rewritten as:

$$\begin{aligned}\Delta H &= N(\mathcal{S}(a,T))D(V,c) - N(\mathcal{S}(a,T))\left[D(V,b) + N(\mathcal{S}(a,T))d(b,c)\right] \\ &= \Delta H_{\text{ bad}} - \Delta H_{\text{ good}},\end{aligned}$$

where $\Delta H_{\text{ bad}}$ and $\Delta H_{\text{ good}}$ are non negative. When deciding whether to accept a change with a positive $\Delta H$ it is necessary to look at both its components. If $\Delta H_{\text{ good}}$ is very much smaller than $\Delta H_{\text{ bad}}$ then the change would result in degradation of the quality of the tree. If the $\Delta H_{\text{ good}}$ is larger than $\Delta H_{\text{ bad}}$ then the change results in an improvement. However if the ratio of $\Delta H_{\text{ good}}$ to $\Delta H_{\text{ bad}}$

is only slightly less than one, the modification should be considered if it results in the use of a previously unused link.

The choice of the exact ratio $r = \frac{\Delta H_{\text{good}}}{\Delta H_{\text{bad}}}$ to use as a cut off point is somewhat arbitrary and may require tuning. There is a balance between maximising link utilisation and maintaining route quality. If $r$ is close to one, then $H$ is lower than if $r$ is close to zero. In addition, as more trees are generated, and more links are utilised, it is necessary to lower $r$ in order to include more unused links.

### 3.4.2 Measurements of Routing Quality

In order to improve the quality of routing, a method for measuring it is required. The first metric is total path length, $H$, which was introduced in Equation 3.2. This metric and others based on path length are discussed in Section 3.4.3.

The second measurement of quality reflects the utilisation of the links. Let $u_{ij}$ be the number of times that the link between two connected nodes $i$ and $j$ appears in the set of all paths linking distinct, unordered, node pairs. Then $u_{ij} = u_{ji}$ and

$$U = \sum_{(i,j>i)} u_{ij} = H.$$

It is relatively simple to compare the value of $H$ for shortest path routing in a network and the value of $H$ for AR with a tree generated from that network. It is more difficult to compare utilisation figures. Since there may be multiple paths of the minimum length between two nodes in a network, the utilisation figures $u_{ij}$ are not fixed, even for shortest path routing. Section 3.4.4 introduces methods for comparing the utilisation patterns of different routing schemes.

The accuracy of multitree routing and AR in general, depends on the topology of the network. The closer the network is to a tree, the closer the approximation to shortest path routing. Its behaviour is illustrated Section 3.4.3 with example networks created by network topology generators. In Section 3.5 the results of Internet topology surveys are examined and the implications for the use of multitree AR in large scale simulations are discussed.

### 3.4.3 Path Lengths in Multitree Routing

In this section we examine how multitree routing affects path length. In particular three aspects of its behaviour will be discussed. The simplest task is the measure-

ment of the variation in the value of $H$ as the number of trees in multitree routing is increased, using a single tree AR as the baseline. Next is comparison of the cumulative distribution function of path lengths. Finally the ratio and difference of individual path lengths under multitree routing are compared with those of shortest path routing.

Consider Figure 3.22. Figures 3.22a, Figures 3.22c and Figures 3.22e show the variation in path length as the number of routing trees is increased, for three different size networks.

Figures 3.22b, Figures 3.22d and Figures 3.22f show the variation in path utilisation as the number of routing trees is increased, for the same three networks. The smallest of these networks is one hundred nodes, the largest ten thousand. The maximum number of routing trees is ten. It is clear that multitree routing improves the value of $H$ — in each case the ratio $\frac{H}{H_{\min}}$ is reduced by factor of at least five. Moreover, the more trees used in multitree routing the better. However, the benefit of an extra routing tree decreases with every tree added. The greatest improvement in $H$ is when the number of trees is raised from one to two and from two to three. After four or five trees the improvement is less dramatic.

Figure 3.23 shows the cumulative distribution function of path lengths for a ten thousand node network. The shortest path routing distribution, and several multitree routing distributions are shown. The distribution for single tree AR is the furthest from the shortest path routing distribution, and the addition of each tree to multitree routing brings it closer to the shortest path. Again though, the benefit of an additional tree is lower if there is already a large number of trees.

Due to the greater number of links used, the improvements in path length with multitree routing are more significant than any improvements possible in finding an optimum single spanning tree.

So far we have examined the changes in the bulk behaviour without looking at how individual paths change. For instance, we know that multitree routing reduces total path length, but we do not know whether all paths are equally reduced, or whether, say, only the longest paths are shortened.

Figures 3.24 to 3.25 focus on different aspects of the route length changes in a 1600 node network. Figure 3.24 plots the cumulative distribution function of the ratio of multitree route length to shortest path route length. Even with single tree algorithmic routing, almost 70% of routes are uninflated. This increases to over 95% with nine trees. With single tree routing there is a small fraction of

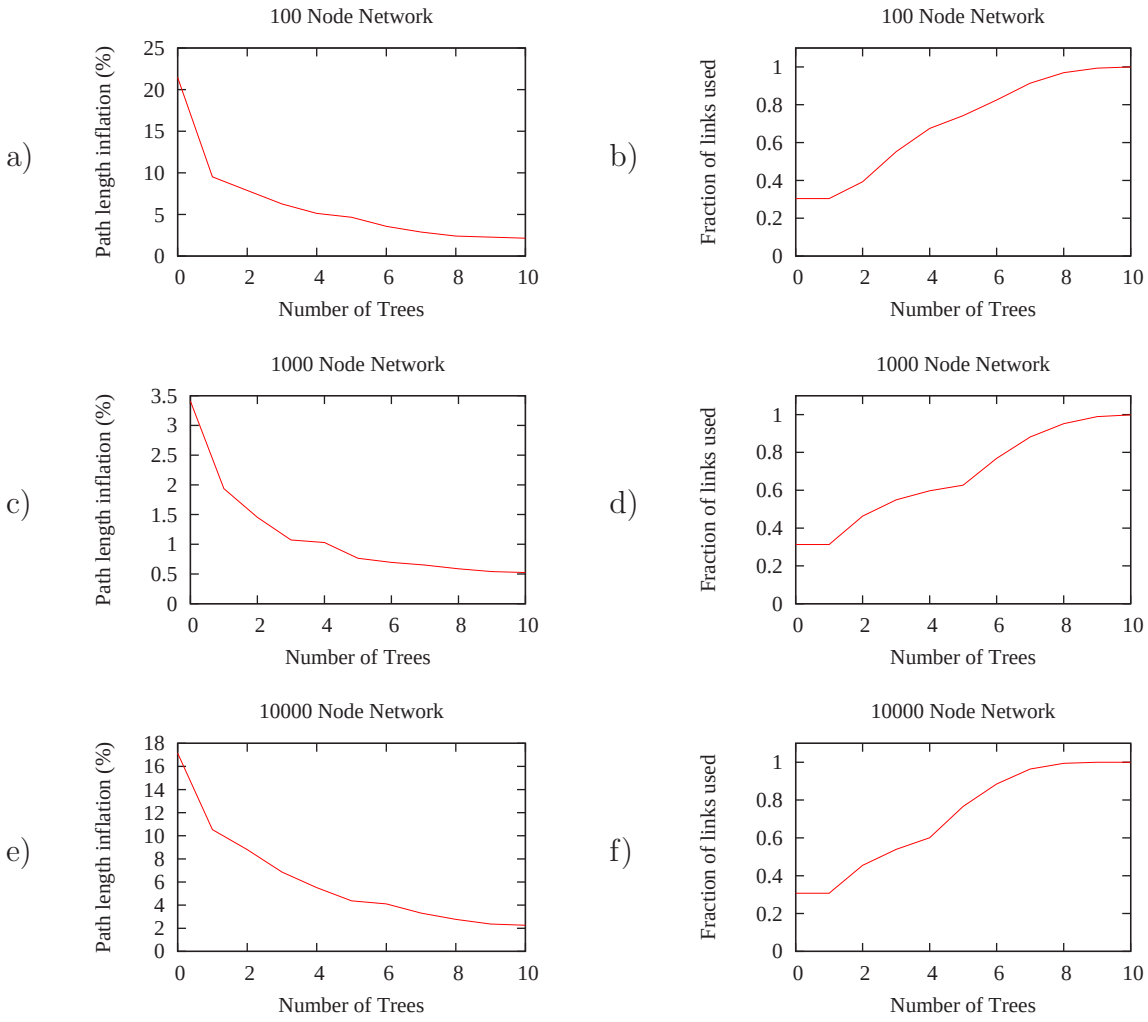



*Figure 3.22: Improvement of route quality for several networks as the number of routing trees is increased. a) and b) One hundred nodes. c) and d) One thousand nodes. e) and f) Ten thousand nodes. a), c) and e) As the number of trees used in routing is increased, the quality increases. The value at $x = 0$ is the value of a single unimproved tree. The value at $x = 1$ is the value for a single improved tree. b), d) and f) The fraction of links used in routing increases with the number of trees.*

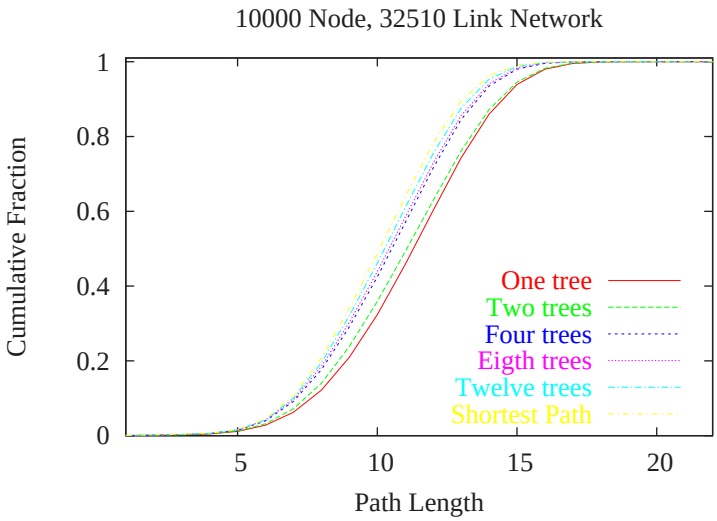



*Figure 3.23: A ten thousand node network. As the number of trees is increased, the cumulative distribution of path lengths approaches that of shortest path routing.*

routes that are over five times the minimum, some up to eleven times. Multitree routing reduces the maximum inflation ratio. In single tree routing over 90% of paths are inflated by four hops or less, but some are inflated by up to ten hops. With multitree routing, over 90% are inflated by one hop or less.

It is interesting to isolate certain paths and examine them. In Figure 3.26 we take the node pairs with the longest 10% of shortest path route lengths and examine the route length ratio and route length difference between multitree and shortest path routing. It can be seen that these long routes are far less inflated than the average. The fraction of routes that are minimum length is higher and the maximum ratio of multitree to shortest path length is 1.3 rather than 11. Virtually all of these long inflated paths are lengthened by just one hop, though with single tree routing, some are up to three hops longer. These results are readily understandable. With node pairs whose shortest path is relatively long, there is less 'space' for AR to extend the path. If there is a cycle between the nodes, they are already, most likely, at opposite ends of the cycle, AR cannot make this situation worse. It is when the shortest path route traverses nodes that are close together on a cycle that the path is lengthened by AR. These longest length paths are also the least affected by increasing the number of trees in multitree routing.

The situation with the node pairs that have the shortest 10% of shortest path route lengths is the reverse, Figure 3.25. The number of uninflated paths is lower than that for the network as a whole, and lower than that of the longest path. With single tree routing, under 60% of paths in our sample network are minimum length. Even with nine tree multitree routing, less than 95% are minimum length. With single tree and three tree routing a small number of paths are up to five times longer than the minimum — some up to twelve times longer. The difference between these paths and the longer paths is that a small difference in hop count has a much larger relative affect on the path length. Increasing the number of trees used has the greatest affect on these routes.

### 3.4.4 Utilisation of Links in Multitree Routing

So far discussion of AR has centred around the length of routes generated. In this section attention is switched to the distribution of link utilisations: how frequently a given link is used. We assume that every source–destination pair has the same probability of occurring.

AR generates routes that may not be minimum length. However, this is less

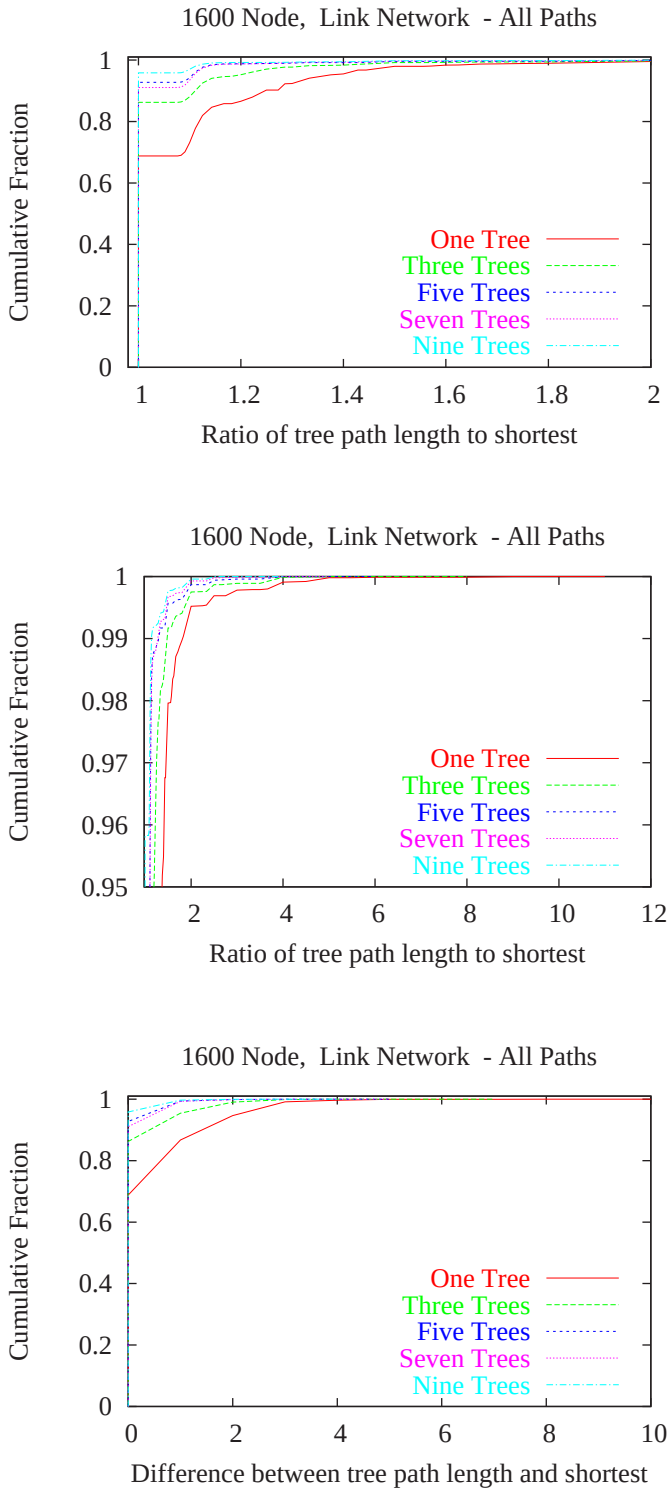



*Figure 3.24: Two sections of the cumulative distribution function for the ratio of path length in multitree AR to the path length in shortest path routing. Increasing the number of trees increases the fraction of uninflated paths, and reduces the extent of the inflation of the average.*

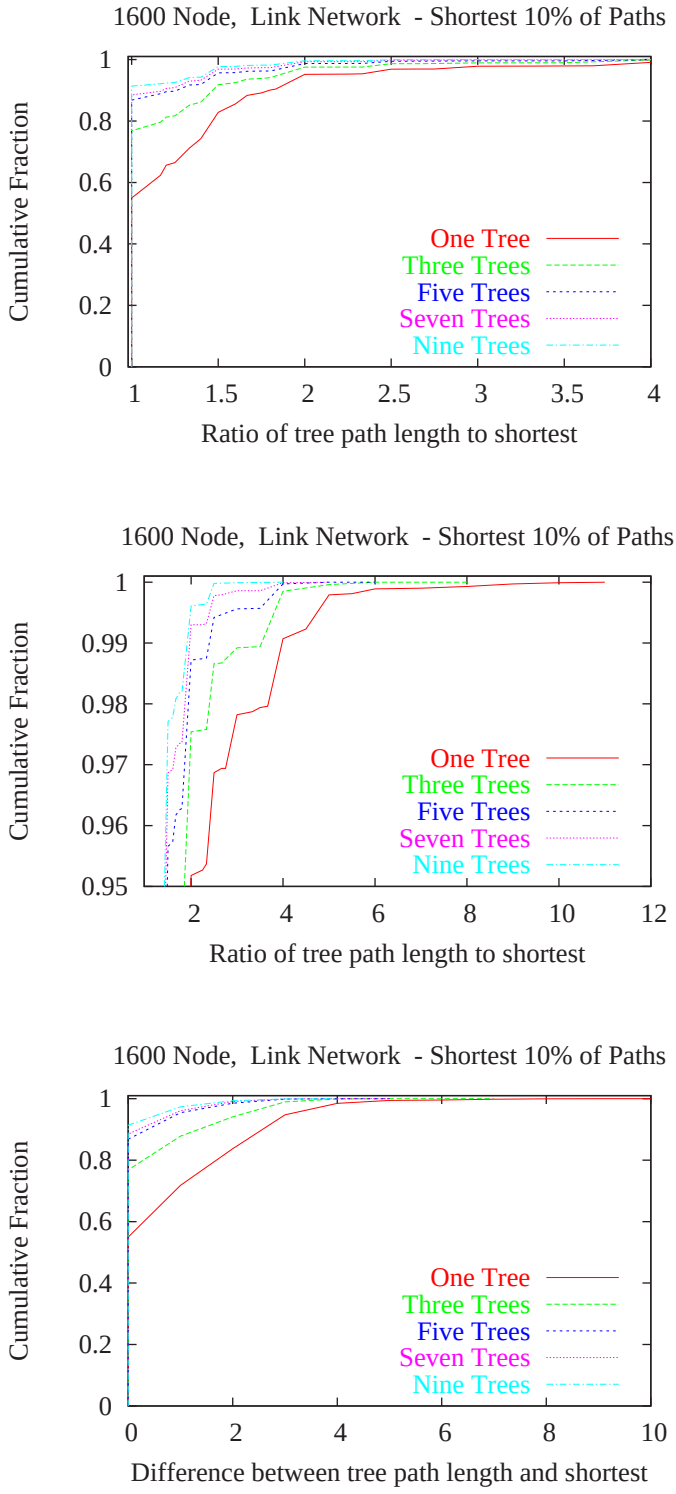
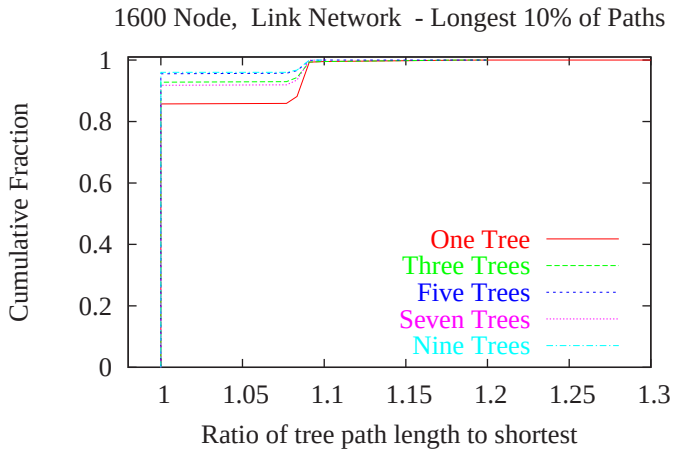
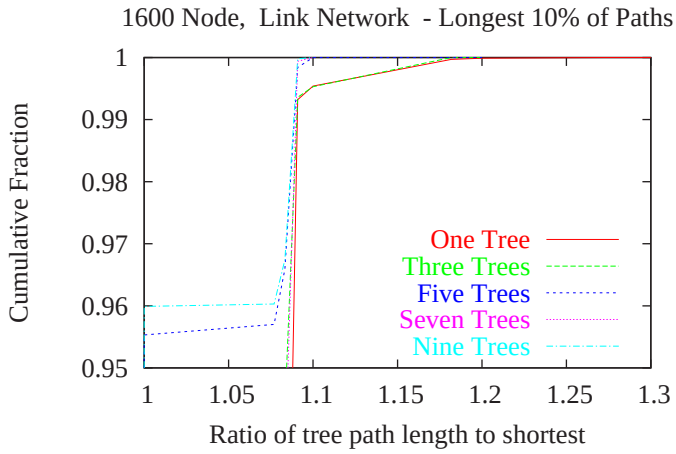



*Figure 3.25: The short paths in shortest path routing tend to be the most inflated in AR, both in absolute hops and in ratio to the minimum.*



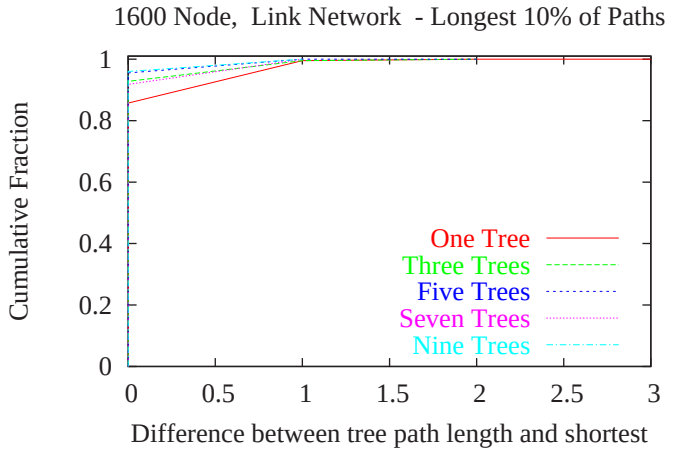




*Figure 3.26: If we consider only the longest paths from the set of shortest path routes, we see that they are less inflated than the others. The maximum path length ratio is* 1.3 *as compared to approximately* 11 *in the previous diagram. Increasing the number of trees reduces the number of inflated paths, and the extent of those still inflated.*

significant than the unbalanced link utilisation pattern. In algorithmic routing, only $N-1$ links are used. $E-N+1$ links are completely unused and the others are overused. To compound the problem, the route lengthening effect causes still more use of the already overused $N-1$ links in the tree. The discrepancy increases as the number of links in the network increases beyond the minimum of $N-1$.

The first task in quantifying this effect is to establish a standard which can be used to compare different routing schemes. In measuring $H$, the shortest path value of $H$ was taken as the point of comparison. The situation is more complex when comparing link utilisations. If there are any cycles in the graph, there may be more than one shortest path between two points. This then implies that the utilisation patterns for two full sets of shortest path routes may be different. In addition, in a large network or even in a highly connected small network, it is not feasible to calculate every possible set of shortest path routes. The solution used involves generating as large as possible a selection of shortest path routes, then taking the average utilisation of each link as the baseline for comparison. The next task is how to compare a given utilisation pattern with our new standard.

Due to the complexity and range of possible behaviour, it is not usually useful to compare the utilisation of a given set of routes to the standard by a single criterion. Let $U_{ij}$ be the average number of times the link from node $i$ to node $j$ is used, where the average is taken over all possible sets of shortest path routes. In practice, this would have to be estimated by an appropriately sized sample. Let $u_{ij}$ be the utilisation for the set of routes whose quality we wish to examine. Two possible single number measurements of utilisation quality are:

$$U_{\text{diff}} = \sum_{i,j} |(U_{ij} - u_{ij})|$$

and

$$U_{\text{ratio}} = \sum_{i,j} \frac{u_{ij}}{U_{ij}}$$

However, these single value measurements are not as useful as an examination of the distribution of either the difference or ratio of $u_{ij}$ and $U_{ij}$. It should be noted that even for a full set of shortest path routes, there will be a difference between $u_{ij}$ and $U_{ij}$.

A graphical approach can yield a greater understanding of utilisation of the links. The graphs used as examples in the following sections show the cumulative

distribution function. They plot either $(U_{ij} - u_{ij})$ or $u_{ij}/U_{ij}$ on the $x$–axis and the probability that $x \leq X$ on the $y$–axis.

However, even this method of studying utilisation is not very discriminating. For instance, a given value of $(U_{ij} - u_{ij})$ has very different importance depending on whether the link from node $i$ to $j$ is a busy link or a quiet link — it's a smaller relative difference for the busy link. Similarly, if a core link is twice as busy as the average, this will affect more routes than if a peripheral link is twice as busy as the average.

Since a core link has a greater affect on traffic than a peripheral link, it is sometimes illuminating to isolate the busiest links — those with the highest $U_{ij}$ — and examine the cumulative distribution of the ratio and difference restricted to just those links.

### Utilisation and Connectivity

Let us now consider the link utilisation patterns of some example networks. We look at three networks. Each network has one hundred nodes, but different connectivity: 146 links, 949 links and 1995 links.

Figure 3.27 plots the cumulative distribution functions for the three networks. Each graph plots the CDF for a single shortest path set of routes and for a basic AR set of routes. We can see the fraction of unused links increasing as link/node ratio increases for AR. In addition the maximum link utilisation ratio also increases as the link/node ratio increases For the sample shortest path route some under utilisation and some over utilisation is also visible, but is less than that for AR. However the slope of the CDF flattens as the link/node ratio increases.

### Utilisation in Multitree Routing

In multitree AR several trees are constructed. They are chosen so as to maximise the number of links used. This, as we saw in Section 3.4.1, improves the value of $H$, but should also improve the utilisation of links.

Figures 3.28 to 3.31 show the cumulative distribution function of the ratio and difference of link utilisations for a one thousand node network. Figure 3.32 shows the distribution for the busiest nodes in a larger ten thousand node network.

We will make some general comments about these graphs. The multitree algorithm is closer to a typical shortest path set of routes than single tree AR. There

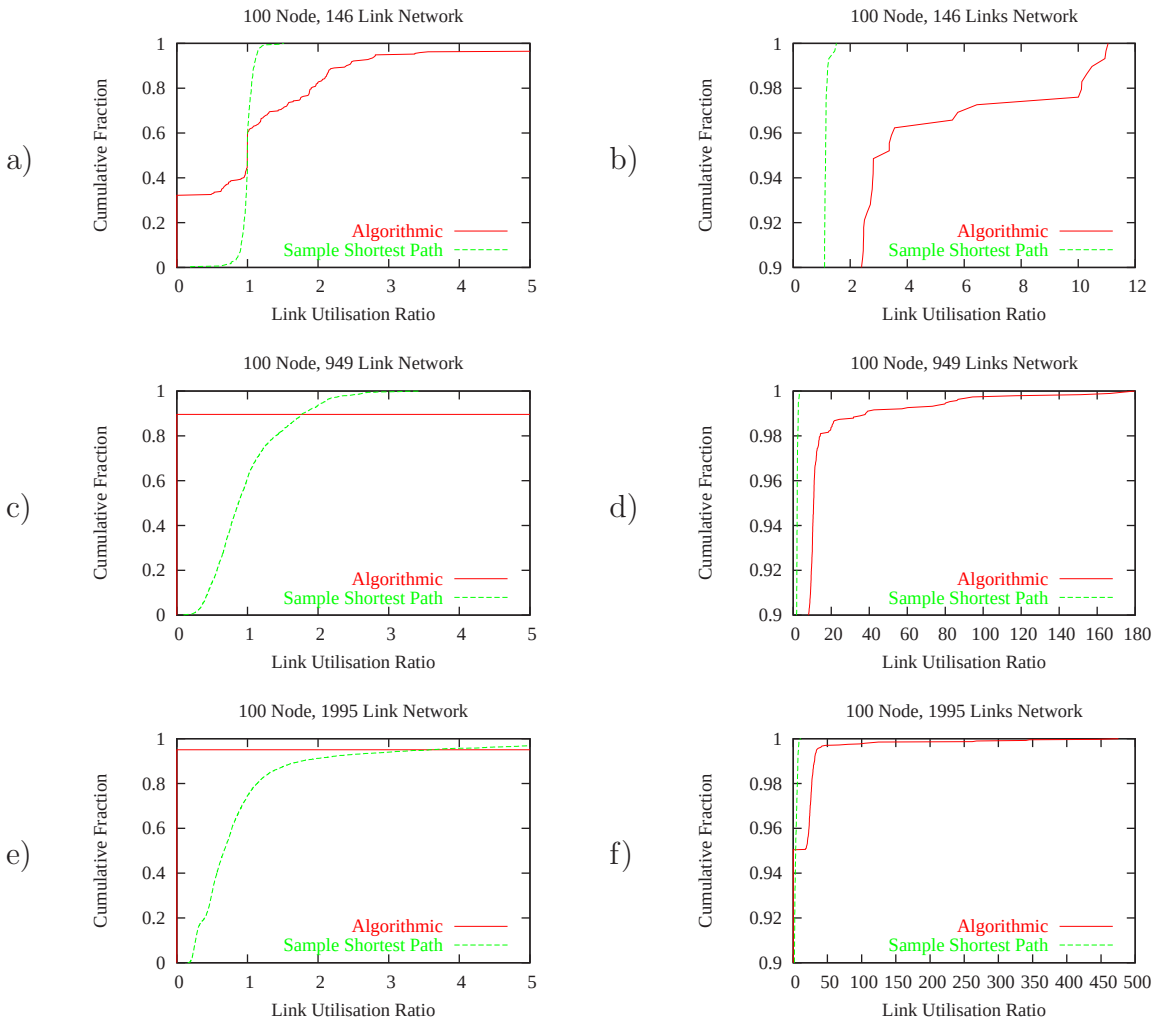



*Figure 3.27: One hundred node network. With a higher ratio of links to nodes, the proportion of unused links increases and the overuse of the utilised links worsens. a) and b) show the utilisation ratio of a network with 146 links. c) and d) show the utilisation ratio of a network with 949 links. e) and f) show the utilisation ratio of a network with 1995 links. a), c) and e) show the CDF at low utilisation ratios, b), d) and f) at high utilisation ratios.*

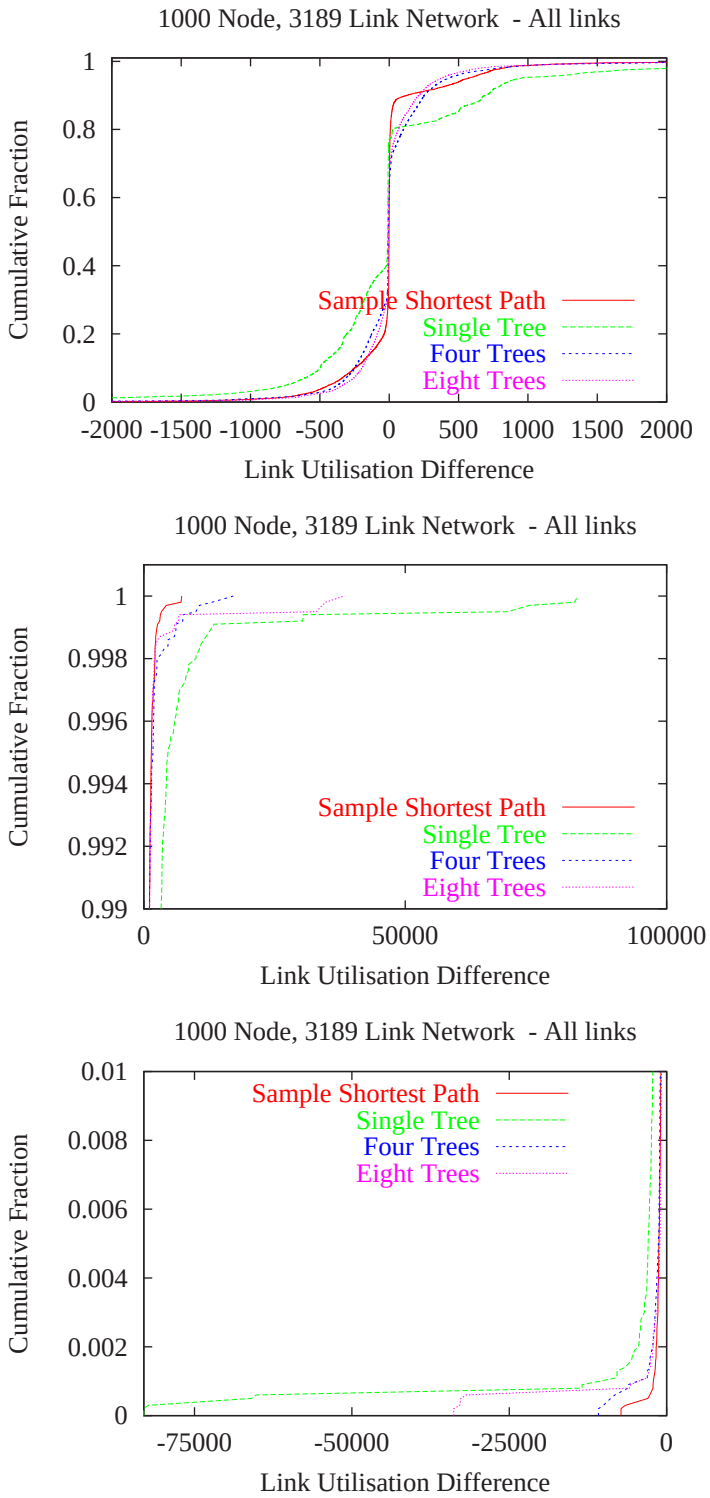



*Figure 3.28: Utilisations for several routing schemes. Each plot is the CDF of the difference between the routing scheme link utilisations and the averaged shortest path link utilisations. The four and eight tree multitree schemes are similar to each other. The multitree schemes are closer to the sample shortest path scheme than the single tree scheme. The three graphs look at different areas of the CDF plots.*

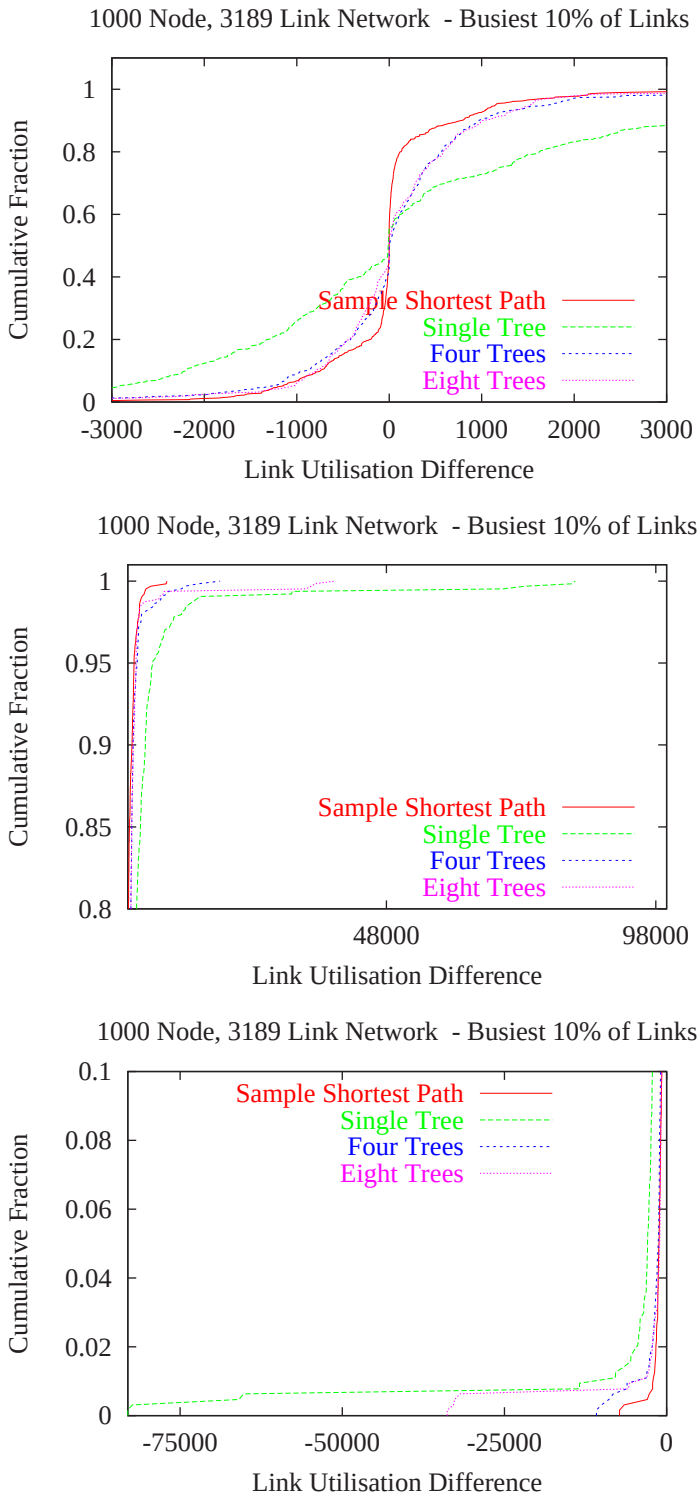



*Figure 3.29: These graphs plot the CDF of the link utilisation difference of the busiest 10% of links in the network. As in the previous diagram, the multitree routing schemes have similar utilisation patterns, and are closer to the sample shortest path than the single tree AR scheme.*

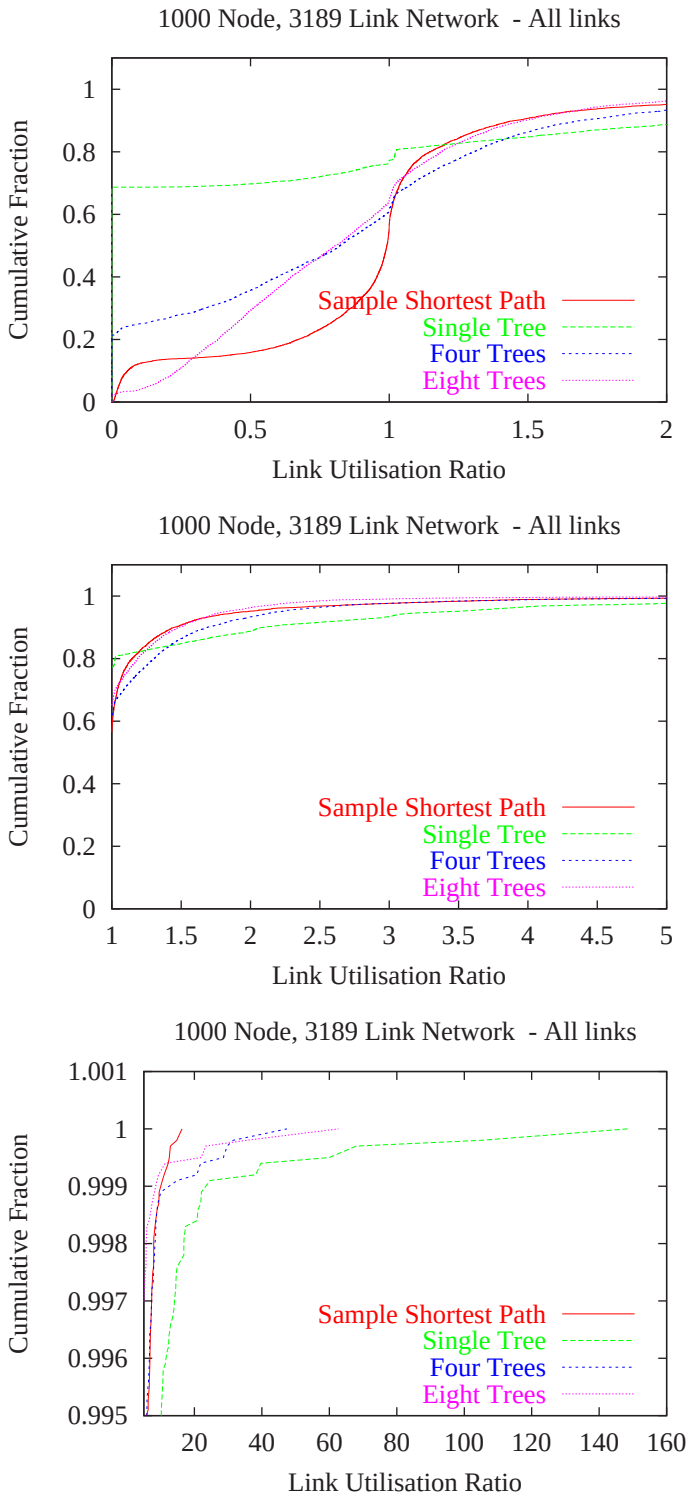



*Figure 3.30: Utilisations for several routing schemes. Each plot is the CDF of the ratio of the routing scheme link utilisations to the averaged shortest path link utilisations. The four and eight tree multitree schemes are similar to each other. The multitree schemes are closer to the sample shortest path scheme than the single tree scheme. The three graphs look at different areas of the CDF plots. The biggest differences are for small utilisation ratios — less than one, and for large utilisation ratios.*

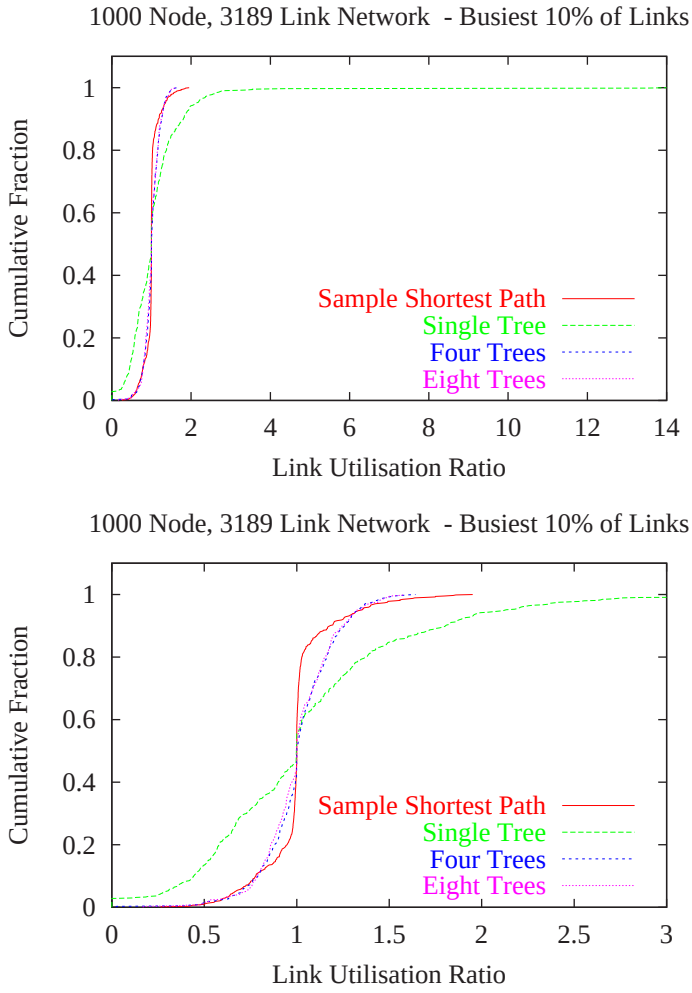



*Figure 3.31: In the busiest 10% of links, the difference between single tree AR and multitree or sample shortest path, is large. In the sample shortest path scheme and the multitree schemes, no link is used more than twice as often as in the averaged shortest path. In single tree AR, some links are used up to fourteen times more often than in the averaged shortest path.*

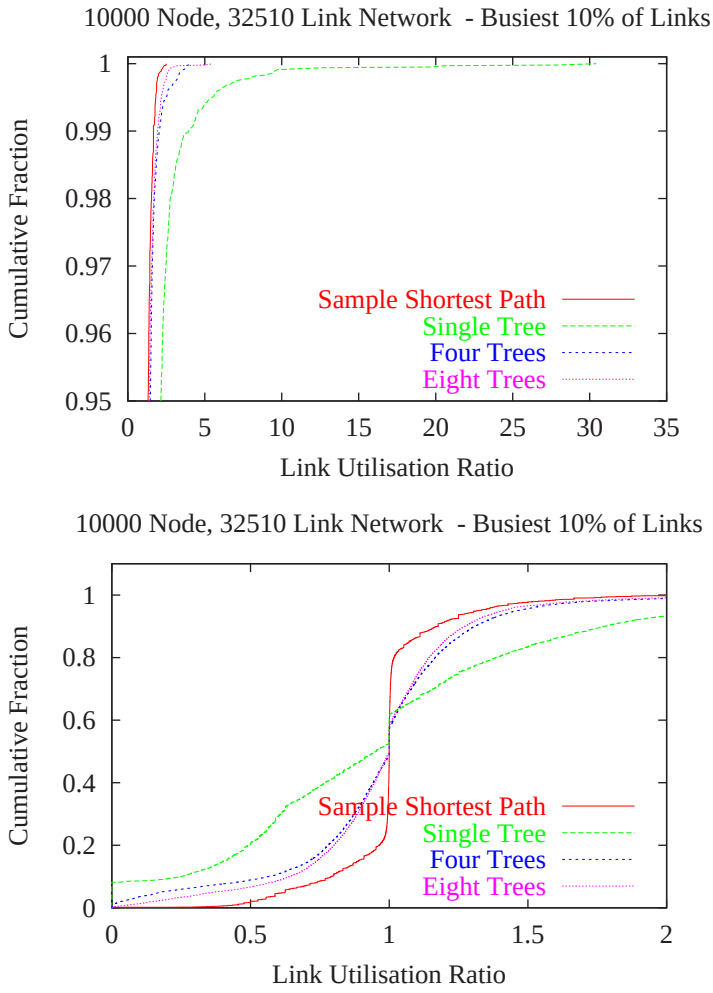



*Figure 3.32: In a large ten thousand node network, the link utilisation ratio on busy links is even higher, and the difference between single and multitree routing more pronounced.*

are both fewer under utilised links and fewer over utilised links. This difference is particularly noticeable when the busiest links are considered in isolation (Figures 3.31 and 3.32). These graphs plot the cumulative distribution function for two multitree sets of routes, one with four trees and one with eight trees. The plots are similar. It seems odd, initially, that doubling the number of trees from four to eight does not result in a significant improvement of link utilisation. The cause of this is the variation in link utilisation. Each extra tree uses more links in the network. Links that result in the best change in $H$ are chosen first. Thus, most busy links are present even when the number of trees is low. The situation is different for quiet links. Few of these are present when the number of trees is low. These two phenomena are illustrated by Figures 3.33 and 3.34. There is a considerable improvement in the utilisation of quiet links as the number of trees is increased. A small fraction of these quiet links are greatly overused. This effect is reduced as the number of trees is increased.

Constructing multitree routing is an inexpensive operation. Therefore, before running a very large simulation, it is worthwhile testing the network topology with varying numbers of routing trees to determine the optimum number to use in a full network simulation.

## 3.5 Discussion of Routing and Network Topology in the Internet

We now turn to research on the topology and routing behaviour of the Internet. Section 3.6 follows with a discussion of the appropriateness of AR to large scale simulations in the light of this research.

### 3.5.1 Distortion and the presence of tree structures in the Internet

Recent studies [13] [30] [67] on the topology of the Internet have revealed that large sections of it have a tree structure. This has remained true during several years of sustained high growth. The structure is present in both router level and Autonomous System level network graphs.

Faloutsos et al. [30] studied three data sets containing partial snapshots of the inter-domain topology of the Internet between 1997 and 1998. The graphs ranged

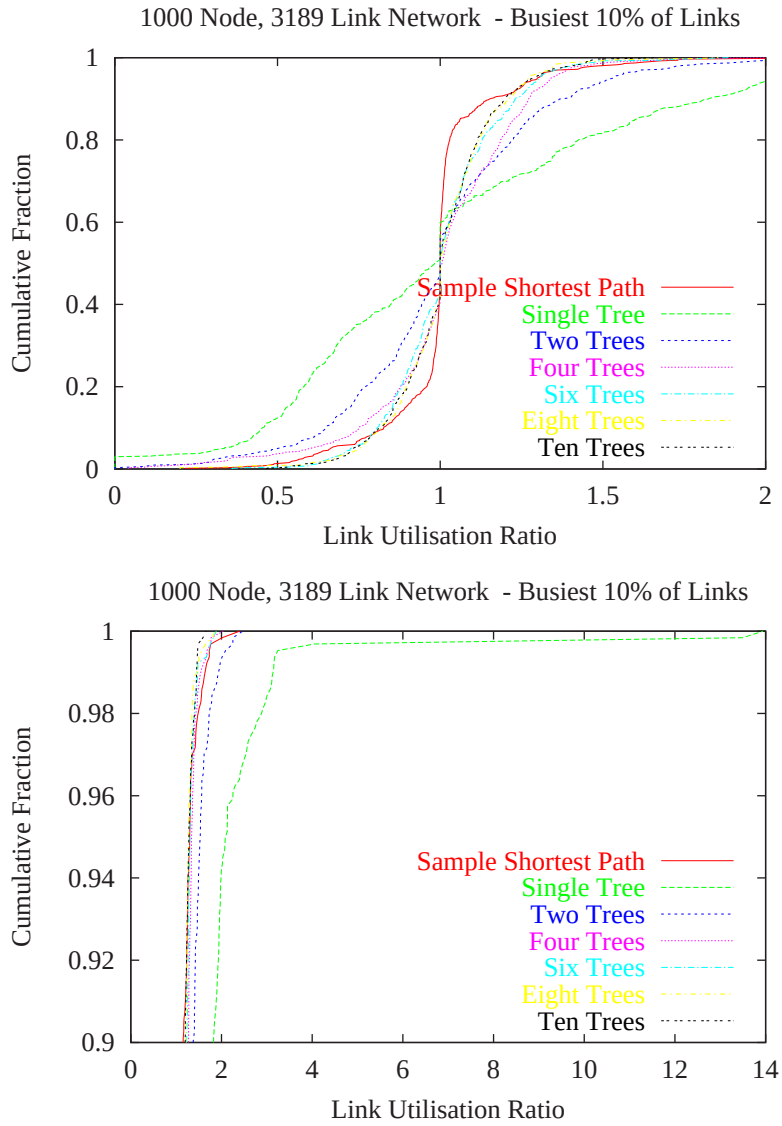



*Figure 3.33: For the busiest links, increasing the number of trees beyond a certain point does not significantly improve the link utilisation ratio. Contrast this with the quiet links in Figure 3.34.*

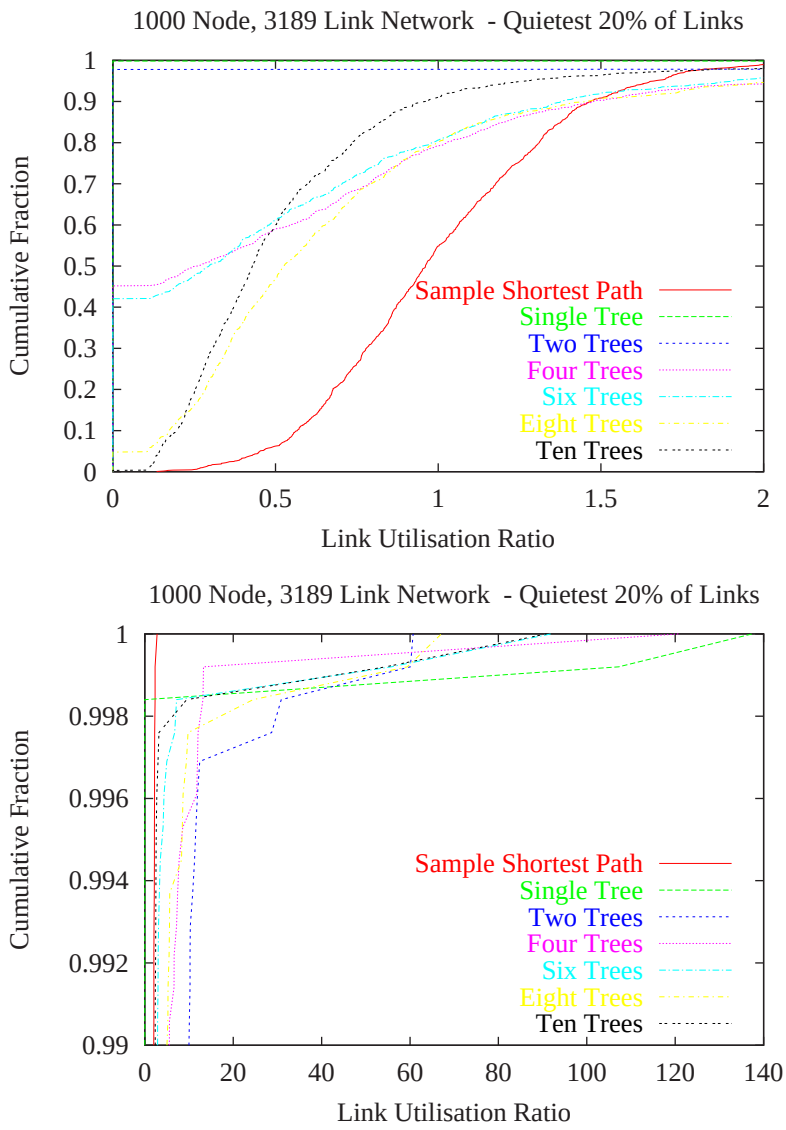



*Figure 3.34: The number of trees used in multitree routing has a large effect on the utilisation of quiet links. As the number of routing trees is increased, so does the utilisation of quiet links. Some of these quiet links are overused in single tree routing. Multitree routing helps reduce this.*

in size from 3015 nodes in 1997 to 4389 in 1998. They observed that 40% to 50% of nodes (Autonomous Systems) were in trees. However the maximum observed tree depth was three, and 80% of trees had a depth one. They also noted that the tree size was decreasing; the Internet was becoming more connected on the scale of Autonomous Systems.

Magoni and Pansiot [67] collected BGP data between 1997 and 2000, to build Autonomous System level graphs of the Internet. In the most recent data set, there were 7624 Autonomous Systems. Of these, 2801 (38%) were in trees (excluding the tree roots). There were 591 distinct trees, with a maximum depth of three. Once again the trees were small: a mean size of 5.74 nodes, including the tree root. Over 90% of the Autonomous Systems in trees consisted of single leaves directly connected to the tree root.

Broido and Claffy [13] studied data collected by CAIDA with their skitter tool in 2001. Their data set consisted of over 655,000 Internet hosts. They found that 55% of nodes are in trees (not including the roots of the trees). The maximum tree depth was nine. It should be noted that this was a router level survey rather than the Autonomous System level surveys of [30] and [67].

A very recent paper by Tangmunarunkit et al. [91] uses a metric called *distortion* to compare topologies. They include router level and Autonomous System level Internet maps as well as topologies generated by several widely used network topology generators [98] [4] [95] [29]. Distortion [91] [45] is a metric describing how treelike a graph is. It is calculated as follows. Take a spanning tree $T(V, F)$, of a graph $G(V, E)$ where $F \subseteq E$. Calculate the average distance in the tree between all nodes that are connected in the original graph $G(V, E)$. This measures how much $T$ distorts $G$, that is, how many extra hops are needed to get from one side of an edge in $E$ to the other if the path is restricted to the edges in $F$. The distortion, $D$, is the minimum of all such averages taken over all possible trees, $T$. Random and mesh networks have a high distortion; they are not treelike. On the other hand, Tangmunarunkit et al. found that the two real Internet topologies had a low distortion, as did the topologies generated by most generators. Unfortunately, distortion is quite hard to calculate (it is an NP complete problem), otherwise it would be a useful metric in predicting the accuracy of AR for a given graph.

### 3.5.2 Asymmetries in Internet Routing

A routing asymmetry arises when the forward path between a pair of hosts differs from the return path. Asymmetries can arise for several reasons. Firstly, errors in network equipment or errors in configuration can cause a link in one direction to be unused or underused. Secondly, there may be an inbuilt imbalance between the two directions: for instance due to a bandwidth difference. The third case is the most common. It is due to the hierarchical and policy based nature of inter domain routing in the Internet.

*Hot potato* routing is the name given to a very common policy for routing packets between Autonomous Systems. Two Autonomous Systems may be linked at several points. For instance, two national networks, each an Autonomous System, might be connected at several major cities. Suppose that network $A$, needs to deliver a packet to network $B$. In hot potato routing, network $A$ hands over the packet to network $B$ at the transfer point between the networks that's closest to the source. Network $B$, when returning packets from the destination, chooses the transfer point that is closest to the destination. For example, in network $A$, Galway and Cork are connected via Dublin and Waterford. In network $B$, they are connected by Limerick. In addition the two networks have transfer points in both Cork and Galway. Consider the path between a node $a$ in network $A$ in Galway, and a node $b$ in network $B$ in Cork. The path in one direction is:

$$a \longrightarrow A(\text{Galway}) \longrightarrow B(\text{Galway}) \longrightarrow B(\text{Limerick}) \longrightarrow B(\text{Cork}) \longrightarrow b$$

the return journey is:

$$b \longrightarrow B(\text{Cork}) \longrightarrow A(\text{Cork}) \longrightarrow A(\text{Waterford}) \longrightarrow A(\text{Dublin}) \longrightarrow A(\text{Galway}) \longrightarrow a$$

So, in this case, not only are the paths asymmetric, but one has more hops than the other.

There are two reasons for this. The practical reason is due to the lack of knowledge each network has about the other. Network $A$ will typically have no knowledge of the internal topology of network $B$. Therefore it is unable to determine a shortest path between individual nodes in the networks, and the best it can do is to choose the shortest path out of its own network and into network $B$. The economic reason is that the faster the packet leaves network $A$, the smaller

the cost is to network $A$, since that packet is not using its resources.

Paxson [76] studied routing data from 1995. The data included over 11,000 paths. In this set, 49% of paths were asymmetric — visiting at least one different city. When considering Autonomous Systems, rather than cities, about 30% were asymmetric. Most asymmetric paths on the Autonomous System level were only asymmetric by one hop. However, on the city level, about a third of these paths were asymmetric by two or more cities.

### 3.5.3 Suboptimal Routing in the Internet

In Section 3.5.2 we discussed the asymmetries observable in Internet paths, due in the most part to policy based and hierarchical routing. Several papers [88] [93] [92] have demonstrated that the policy and hierarchy based routing protocols of the Internet are also responsible for the choice of suboptimal paths. (It was known in theory that hierarchical routing can be suboptimal, but its effects had not been previously studied in practice).

Savage et al. [88] have shown that for 30% to 80% of Internet routes, a superior alternative exists. They examined several datasets collected between 1995 and 1999, consisting of paths between a number of hosts. Bandwidth, round trip time and drop rate were used as measures of path quality. To test for the existence of a superior path between two endpoints, they composed two or more other paths so that the composition formed a new path between the same points. Despite some of these synthetic paths traversing the same links twice, they found that the new path was superior surprisingly often. For 30% to 55% of paths they could create a path with a lower round trip time. For 75% to 85% of paths, an alternative with lower packet loss was found. Finally, 70% to 80% of routes had alternatives with a higher bandwidth.

Two papers by Tangmunarunkit and colleagues [93] [92] used routing simulations to gauge the effect that the hierarchical nature of the Internet routing protocols has on path length. They took a router level map of the Internet [40] and, using simulations, compared shortest router level paths with the paths generated by hierarchical routing. They used a simplified model of inter-AS routing where the AS level path was chosen to minimise the AS-level hop count. A more faithful system was used in their second paper [93], and confirmed the earlier findings.

Comparing shortest path and policy based routes, they showed that 20% of

paths are longer than the shortest path by more than five hops. In addition 20% are inflated by 50%. Only 20% of policy based paths are shortest paths.

### 3.5.4 Node Degree

The node degree distribution and link/node ratio have a strong influence on routes in a network.

In the three data sets of Faloutsos et al. [30] the link/node ratio at the Autonomous System level increased from 1.71 to 1.88 over the course of the two years of the study. They proposed a power law relation for the frequency of outdegree values (the outdegree of a node is the number of directed links from that node to other nodes). The maximum outdegree was 979.

Magoni and Pansiot [67] in their Autonomous System level graphs of the Internet found a link/node ratio of 2.0. The maximum node degree was 1704.

In the CAIDA [13] survey of individual routers, Broido and Claffy found the link/node ratio to be 1.92. The maximum node degree in the core of the graph was 850.

For comparison, the link/node ratio in a tree is $(N-1)/N$ which is close to unity for large trees.

## 3.6 Case Studies

The research on Internet topology surveyed in Section 3.5 provides some encouragement for the use of AR in large scale simulations of the Internet. Two factors, in particular, suggest that it is an appropriate approximation. First the low link/node ratio and presence of tree structures mean that the Internet topology is close to that of a tree. This suggests that the approximation error introduced by AR is low. Second, routes in the Internet are frequently not shortest path so that the lengthening introduced by AR may in fact be an advantage, especially if the magnitude of the lengthening can be adjusted to match that observed in the Internet.

Other factors must also be taken in account: link utilisation and the performance of the algorithm, for example.

In previous chapters we used relatively small networks generated by GT-ITM as test cases. We wish to use AR as a tool in large scale Internet simulation.

Therefore in this chapter we use two networks that embody Internet topologies. The first is the SCAN map of the Internet [51]. It is a router level map consisting of 2,282,298 nodes and 320,203 links (the original contains several loopback and duplicate links which have been removed). The second test case is a 10,000 node Autonomous System level map with 20575 links, generated by the Inet 3.0 topology generator [95]. This is a modern generator, which creates networks consistent with recent topology research [91]. These networks provide a true test of the usefulness of AR in large scale network simulation.

### 3.6.1 Route Length

Since the networks, in particular the SCAN map, are large, it is impractical to measure the lengths of all $N \times (N-1)$ routes. Instead a large random sample of routes was selected and used to provide an estimate. Figure 3.35 shows the ratio of $H$ to $H_{\min}$ for several varieties of AR. This diagram clearly shows the dramatic improvements made possible by the techniques introduced in this chapter. For both networks $H/H_{\min}$ is over 1.3 with AR using a BFS tree. The tree improvement algorithms of Section 3.3 reduce the error by approximately half. Multitree routing reduces the error still further so that with eight trees $H/H_{\min}$ is 1.08 for the SCAN map and 1.05 for the Inet map. The graph of $H/H_{\min}$ flattens considerably as the number of trees is increased. In fact, as Figure 3.35 shows, the effectiveness of multitree routing increases only as the log of the number of trees.

Figures 3.36 and 3.37 plot the cumulative distribution function for individual paths: their ratio to the minimum and the increase in length. There are several noteworthy points. The fraction of unlengthened paths is very low with the BFS routing tree. The multitree schemes show a large improvement over the BFS tree, but adding extra trees only results in small incremental improvements.

Table 3.6 compares route length inflation figures as estimated by Tangmunarunkit et al. [93] and those resulting from AR using both a BFS tree and multitree routing. Tangmunarunkit et al. used a 102,000 node map of the Internet, while the figures from AR were generated using the SCAN 2,282,298 node network.

Somewhat surprisingly, multitree AR generates routes that are actually *shorter* than those generated by hierarchical routing. These figures are not conclusive, as two different network graphs are involved. However as both maps are taken from the Internet, it shows that AR can generate paths that are shorter than real

| | BFS Tree | Multitree (with eight routing trees) | Internet Survey [93] |
|---|---|---|---|
| Percentage of shortest length paths | 5% | 38% | 20% |
| Percentage of paths over 50% inflated | 50% | 3% | 20% |

*Table 3.6: Comparison of the frequency of path length inflation. The columns marked BFS tree and Multitree used AR on the SCAN network map, the column marked Internet Survey shows the results from an Internet survey by Tangmunarunkit et al. [93]*

Internet paths. If desired, multitree AR could generate longer paths by not always choosing the best routing tree for a source destination pair.

### 3.6.2 Link Utilisation

It is difficult to calculate link utilisation in the two test cases. Due to the size of the networks, it is not feasible to calculate the large number of paths necessary for a good estimate of average shortest path link utilisation. There are $N \times (N-1)$ source–destination pairs, which may each have several shortest path routes joining them. Taking a large random sample of node pairs allows for a good estimate of the utilisation of busy links, but is less accurate for quiet links.

Figures 3.38 and 3.39 show the ratio of AR link utilisation to a shortest path link utilisation. It is important to note that the ratio is to a *sample* shortest path, not to an averaged shortest path utilisation. In addition, only the busiest 10% of links were compared. It can be seen that a large proportion of links are either over– or under– used by AR as compared to the sample shortest path routing. Multitree routing reduces the difference, but does not match the sample shortest paths. A better comparison would be to match the AR utilisation with the utilisation that results from the hierarchical, policy based routing of the Internet.

### 3.6.3 Performance

The performance of AR is a final, important consideration. It proves to be efficient. There are three aspects to the issue: the time taken to create the routing trees, the time taken for a next hop calculation, and the time taken to decide on the best routing tree (for multitree routing). These results are summarised in Table

| Routing Scheme | Creation Time (s) | Choosing best tree ($\mu$s) | Next Hop ($\mu$s) |
|---|---|---|---|
| BFS Tree | 0.211 | NA | 0.992 |
| Improved Tree | 4.070 | 0.211 | 0.945 |
| 2 Multitree | 7.930 | 2.359 | 1.016 |
| 3 Multitree | 11.242 | 3.586 | 1.008 |
| 4 Multitree | 14.133 | 4.758 | 0.969 |
| 5 Multitree | 17.242 | 6.039 | 0.984 |
| 6 Multitree | 20.250 | 7.297 | 0.984 |
| 7 Multitree | 23.016 | 8.539 | 0.992 |
| 8 Multitree | 26.055 | 9.969 | 1.000 |

*Table 3.7: Timings for the three phases of multitree routing using the SCAN network. The column titled* Creation Time *records the time taken to setup AR of the type specified in the Routing Scheme column. The column titled* Choosing best tree *records the time taken to choose the best routing tree in multitree AR. It is the average of one million samples. The column titled* Next hop *records the time taken to choose the next hop in a routing path. It is the average of one million samples.*

3.7 using the SCAN network.

These timings were gathered on a 1.0 GHz Intel Pentium. Unsurprisingly, the tree creation and improvement time increases linearly with the number of trees. The next hop calculation time remains roughly the same.

## 3.7 Summary

One necessary part of large scale network simulation is the modelling of a routing protocol. Three factors must be balanced when implementing such a model: accuracy, speed and scalability. The most faithful method involves detailed simulation of the routing protocols, building routing tables at each node. However, if full routing tables are maintained at each simulated router, the memory required precludes the simulation of large networks. Some level of approximation is called for.

Two methods already address the problem of routing in large scale network simulations: algorithmic routing (AR) and $NIx$–vectors. $NIx$–vectors creates routes on demand, and caches them. However in the worst case scenario this could consume $\mathcal{O}(N^2)$ memory. Each route computation requires $\mathcal{O}(N)$ time. Algorithmic routing (AR) maps the network to a tree and uses a simple algorithm

to calculate the path between two nodes. It trades a small increase in computation for a significant reduction in memory use. AR has an advantage in that it requires only $\mathcal{O}(N)$ memory in total and $\mathcal{O}(\log N)$ time per packet forwarded. *NIx–Vectors* can generate shortest path routes while AR lengthens some routes and concentrates traffic onto $N - 1$ links.

We chose to enhance algorithmic routing so as to improve both its speed and accuracy; it already scales well. The improvements made to the basic technique include:

- A modification to the original method that represents the routing tree in a different manner, reducing memory usage and increasing performance (Direct AR).
- A new fixed computational cost routing algorithm.
- A fast, efficient, method for improving routing tree quality.
- A routing scheme using multiple trees for generating shorter paths.
- A method for creating multiple trees to minimise route length and spread link utilisation.

While the performance gains are considerable, the increase in route quality is more significant. Routing tree improvement reduces the the length of generated routes in a single tree. Multiple tree routing reduces route length further, and increases the diversity of paths.

We explored the behaviour of AR across a range of network sizes, including topologies created by up to date network generators, and taken from Internet surveys. The approximations introduced by AR were found to affect long and short paths, quiet and busy links differently.

AR is not hierarchical Internet routing. We have examined data from surveys of Internet routing behaviour and demonstrated that multitree AR can generate routes that are as short or shorter than those found in the Internet. Those generated using a BFS tree are considerably longer. Nevertheless, there are some situations in which AR cannot be used, for instance:

- The testing of routing protocols.
- Simulations in which the behaviour of interest depends sensitively on the routing protocol.

- Simulations in which detailed delay timings are important — route length and path changes will alter packet delay times.

In Chapter 4 we demonstrate that it is possible, using AR, to model networks with over ten million nodes. Although AR adds some computational complexity, this is compensated by reduced memory usage. Without a method such as AR, it would not be possible to simulate such large scale networks.

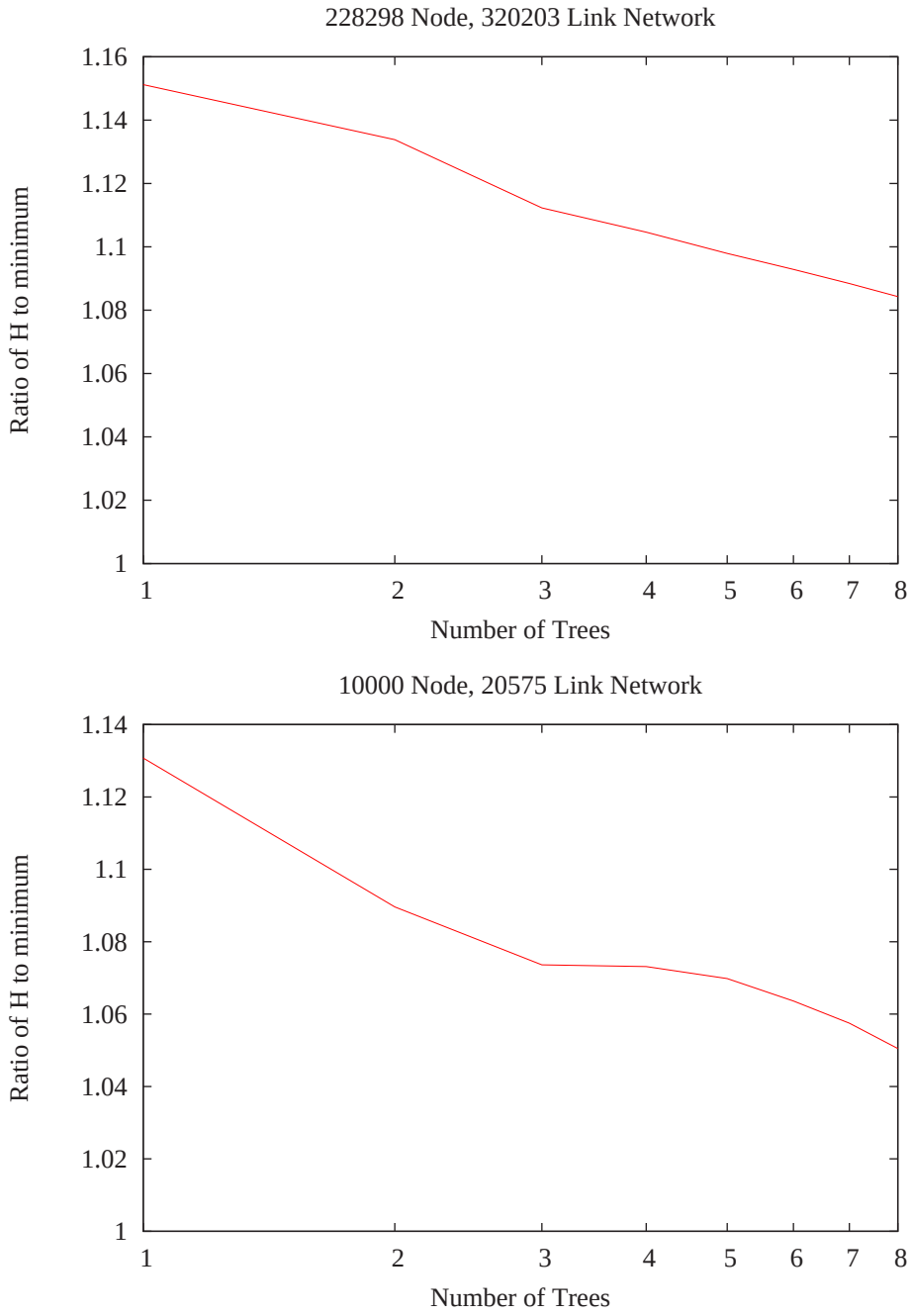



*Figure 3.35:* $\frac{H}{H_{\min}}$ *improves roughly with the log of the number of trees.*

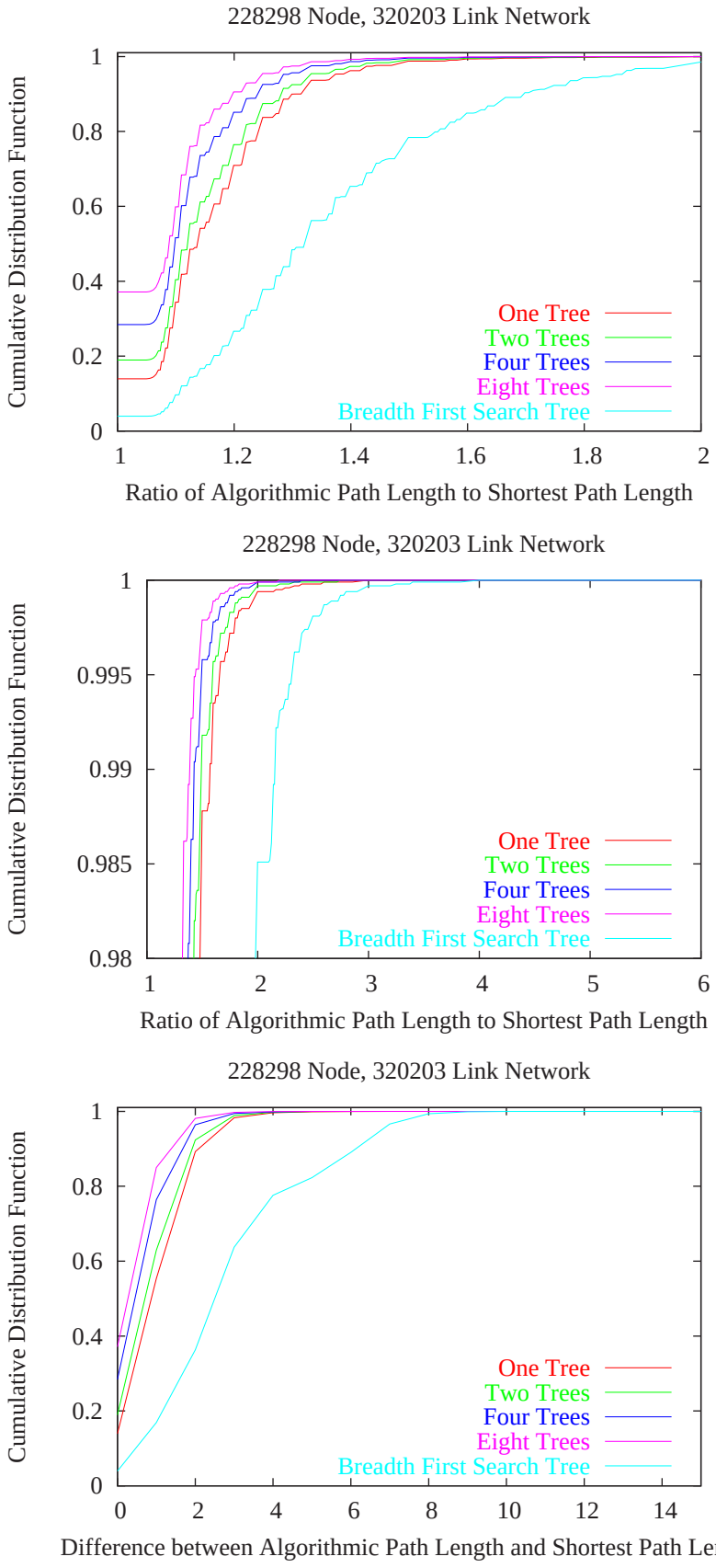



*Figure 3.36: Different sections of the CDF of the path length ratio. Using the route improvement techniques is clearly better than using a BFS tree. Multitree routing offers further improvements*

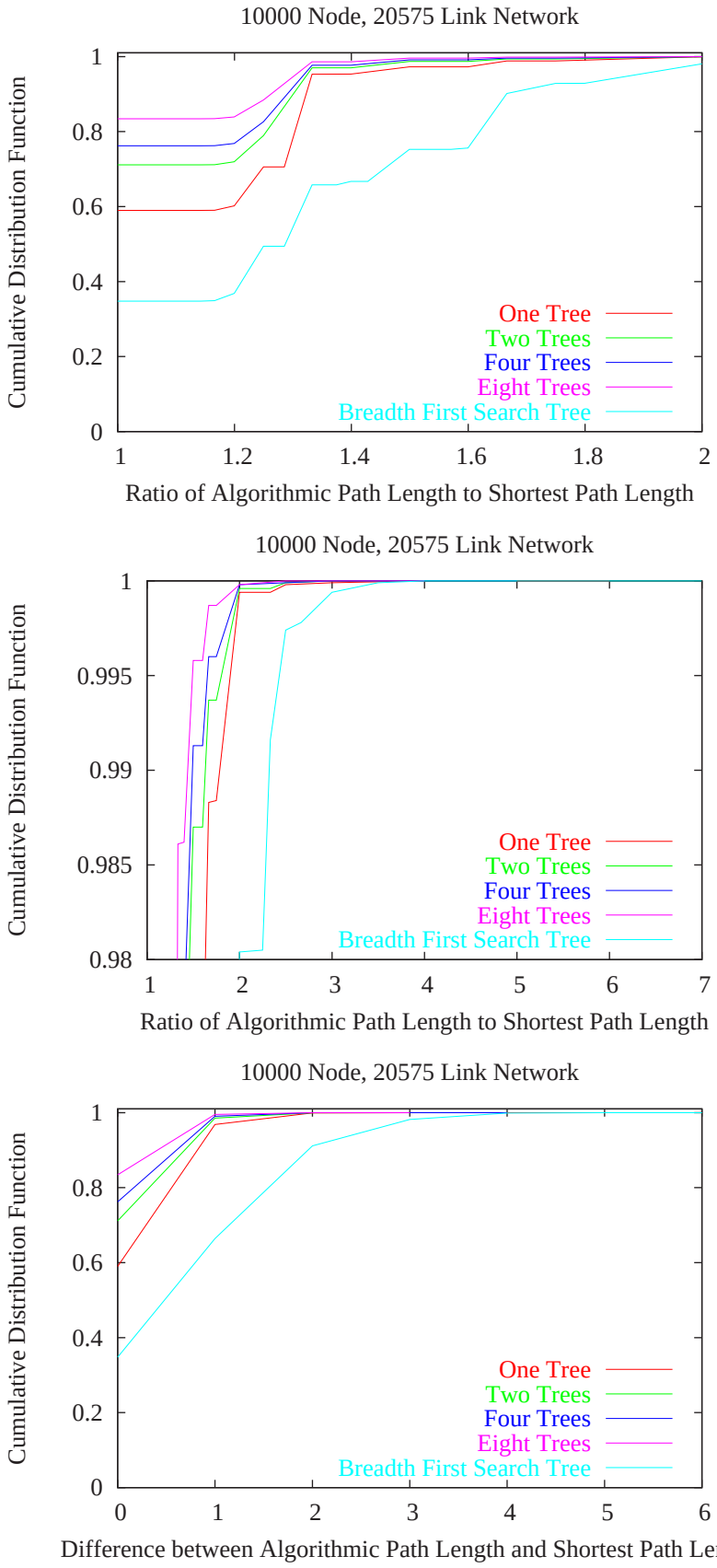



*Figure 3.37: Different sections of the CDF of the path length ratio. Using the route improvement techniques is clearly better than using a BFS tree. Multitree routing offers further improvements*

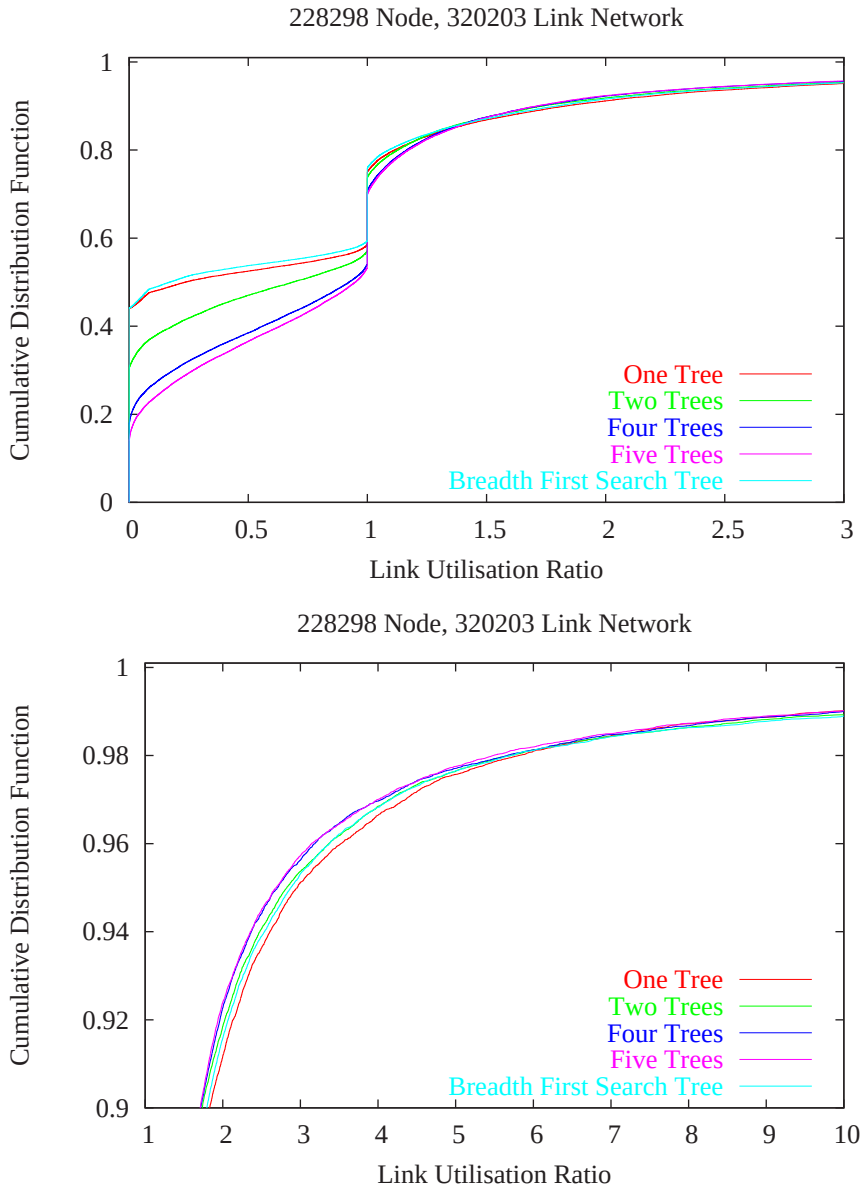



*Figure 3.38: Multitree routing results in less non– and under–utilisation of links. The decrease in over utilisation is not as dramatic.*

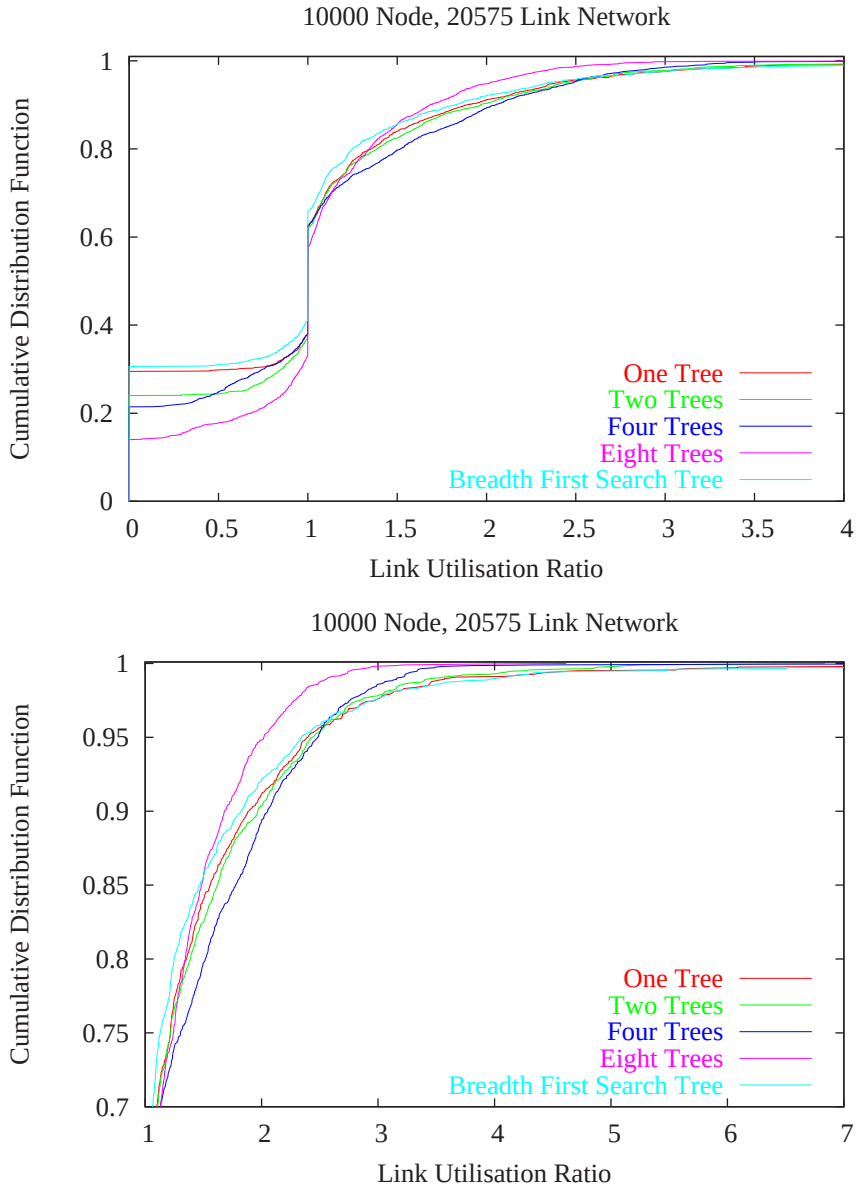



*Figure 3.39: Multitree routing results in less nonutilisation and under utilisation of links. The decrease in over utilisation is not as dramatic.*

# Chapter 4

# Large Scale Simulation

This chapter introduces the second half of our work: the Psim simulator. Psim is a large scale parallel network simulator. Our approach stresses memory efficiency and its design is tailored specifically for network simulation. We aim to simulate the largest possible networks. We believe that with Psim we can model networks of unprecedented size; on a single processor one hundred thousand nodes is possible, with eight processors a million nodes and on a sixty four processor cluster we have modelled a ten million node network. Using sixty four processors we have achieved a packet event rate of over $16 \times 10^6$ events per second on a large network.

In addition the code parallelises well. Our PDES synchronisation scheme is a hybrid of null message synchronisation and windowing [66] [72] synchronisation. However it avoids the global synchronisation point of windowing methods and also generates far fewer null messages than a traditional null message approach. We have been able to produce almost linear speedups.

Despite an approach targeted primarily at large scale simulation, Psim allows for easy addition of new network device and protocol types through its modular design. It currently has network modules to represent routers, links, UDP traffic sources and TCP traffic sources.

The chapter is structured as follows. We begin with a survey of PDES synchronisation techniques. This is an extension of the introduction to PDES in Section 2.4. We move then to an overview of the simulator design and an in-depth look at the simulator kernel, network configuration and the commonly used modules. In Section 4.7 we perform small experiments to explore aspects of the parallelisation scheme. This is followed in Section 4.8 by experiments to determine the scalability and performance of the simulator using large networks. Section 4.9 summarises

the work.

## 4.1 PDES Synchronisation

PDES was introduced in Section 2.4. In this section we explore in greater depth the synchronisation protocols available. Synchronisation schemes can be divided into two classes: conservative synchronisation and optimistic synchronisation. As our work uses a conservative synchronisation protocol we discuss these protocols in depth. Before proceeding, we provide our rationale for avoiding optimistic synchronisation.

In optimistic synchronisation a processor may execute an event $e_1$ with timestamp $t_1$ without the guarantee that another event $e_2$ with timestamp $t_2 < t_1$ cannot arrive. Since $e_2$ could alter the state of the simulation at time $t_1$, the simulation must be restored to its state at time $t_2$, then event $e_2$ processed and finally event $e_1$ must be re–executed. In a large simulation, the necessity of having to continually save the simulation state could prove prohibitively expensive, especially since a *rollback*, or reversion to an earlier simulation time, on one processor may result in a cascade of rollbacks on other processors. In addition, it has been shown that an optimistic PDES running on a $P$ processor parallel machine is on average no more than $\mathcal{O}(\log P)$ faster than a conservative PDES [32].

The first conservative PDES synchronisation protocol was developed by Chandy and Misra [22] and Bryant [17], and is known as the CMB algorithm. It uses null messages, as described in Section 2.4 to enforce causality. It was soon found that large numbers of null messages were generated, indeed that null messages could outnumber real messages. As this leads to a decrease in efficiency, methods to reduce the number of null messages were developed.

The *carrier null message* [18] method adds extra information to a null message. For instance it adds a list of nodes and the timestamps reached at those nodes to null messages. This extra information about the state of its neighbours allows a node to process events with greater timestamps than would otherwise be possible. However even this approach still generates many null messages.

The cost of sending these null messages (and indeed ordinary messages) between processors can be reduced by amalgamating several messages into one large message. The cost of sending a $B$ byte message can be approximated by $\alpha + \beta B$, where $\alpha$ is a fixed overhead needed to initialise a communication between pro-

cessors, and $\beta$ is the cost per byte of sending a message. If $n$ identical messages are sent independently the cost is $n\alpha + n\beta B$. However if it is possible to send them simultaneously in one large message, the cost is only $\alpha + n\beta B$. We use this optimisation in our model in the case where two or more separate simulated links join two physical processors. If several simulated packets need to be transferred from one processor to the other in a sufficiently short period of time, they are sent as part of one large MPI message.

The concept of *lookahead* is important in building efficient PDES models. Lookahead is the distance into the simulation future that a processor can predict. The further ahead it can predict, the more information it can give other processors on when to expect messages. Let the lookahead at time t on a processor be $l$. If a neighbouring processor knows that it will receive no messages until time $t + l$, then it can process events up to time $t + l$. The larger the value of $l$, the less time wasted in unnecessary blocking. Of course, a large lookahead cannot eliminate blocking. For instance if one processor has a lighter workload than another, it will have to block while waiting for the slow processor to catch up. In order to improve lookahead it is often necessary to make use of model specific information. For example, Nicol [71] describes a network of first-come first-served (FCFS) queues. As soon as a job enters a queue its service time is calculated, rather than when the job reaches the server. This increases the lookahead, especially if there are many jobs in a queue. The technique is possible due to the non-preemptive nature of the FCFS queue. It would not be possible in a preemptive queue, as the exit time of a job arriving at $t_1$ could be altered by a more important job arriving at time $t_2 > t_1$.

Chandy and Sherman [23] introduced the idea of *conditional events*. They divided events into two types, *definite events* and *conditional events*. It is always safe to process a definite event. A conditional event may only be processed under certain circumstances, and may depend on the outcome of earlier events. Thus, even if the simulation cannot model any conditional events due to blocking, it can continue to work by processing definite events.

Window based synchronisation [66] [72] is an alternative form of conservative synchronisation to null messages. It typically involves global synchronisation between all processors. Each processor $i$, at time $t$ calculates the interval $\delta_i$ till the next message it will send. A global minimisation to determine $\delta = \min_i \delta_i$ is performed and all processors are then free to simulate any events in the *window* from

$t$ to $t+\delta$. At time $t+\delta$ a new window is calculated. In order for this method to be efficient, it is necessary to calculate the largest possible lookahead. Lubachevsky demonstrated [66] that performance of this scheme does not worsen when both the problem size and the number of processors are simultaneously increased.

Our approach to PDES can be characterised as a hybrid of null message and windowing synchronisation. Unlike the window scheme described above, the window is not recalculated globally at the end of each window period. Instead each pair of processors $i$ and $j$, where $i$ simulates node $n_i$, $j$ models node $n_j$ and $n_i$ and $n_j$ are linked, determine at startup a window $\delta_{ij}$ based on the type of simulated communications link. Processors $i$ and $j$ must synchronise at simulated times $n\delta_{ij},\ n > 0$. The requirement for a global minimisation of $\delta_i$ is removed, but processors $i$ and $j$, with nodes $n_i$ and $n_j$ respectively, must synchronise if $n_i$ and $n_j$ are linked even if the link between the two nodes is quiet for long periods. In practice, in a large simulation, such null synchronisations are not likely to occur, especially if synchronisations due to multiple links between simulated nodes on two processors are amalgamated.

## 4.2 Overview

We now outline our approach to constructing the Psim simulator. Psim is designed primarily for large scale simulations. This required a conscious decision to make many tradeoffs in favour of scalability and performance. Nevertheless, due to the modular nature of its design, it is highly flexible and not restricted to large scale simulation alone.

Among the first issues that must be examined in building a simulation is that of abstraction. How detailed a model do we wish to build? A researcher developing a new wireless protocol may wish to model as faithfully as possible the attenuation of the signal and interference between sources. Another researcher investigating a TCP congestion avoidance algorithm may require that the IP layer be modelled, but might ignore the MAC layer of the network.

In our case we wish to construct as large a model of the Internet as possible. This necessitated some approximations: in particular the use of algorithmic routing and the minimisation of the number of protocol stacks. It should be noted though that these approximations are not imposed by the simulator kernel, but are features of the modules in question.

The use of algorithmic routing eliminates the need for routing tables at every node. The issues of route lengthening and link utilisation have been discussed in Chapter 3. Some scheme of this sort is a prerequisite for large scale simulation.

In order to simulate the largest possible models, Psim was designed to use distributed memory parallel computers. However the simulator kernel is entirely sequential. Most parallel simulators (for example SSF [27], *pdns* [85] and USSF [80]) integrate the parallelisation scheme into the simulator kernel.

Our approach is orthogonal to those mentioned above. The event list on each processor is entirely sequential, and all the parallelisation is handled in the modules. This approach is motivated by a simple observation. Consider a large network partitioned among $n$ processors. Each partition contains roughly the same number of nodes, and is as contiguous as possible. There is only one reason that two processors, simulating two separate partitions, need communicate; that is if a packet departs one partition and enters the other. This provides a clear interface. If we provide a module that accepts packets from the first partition and sends them to the processor responsible for the second partition, we can parallelise the simulation.

If two connected nodes are in the same partition, we join them using the normal *link* module. If they are in separate partitions we join them with a special *bridge* module. There is a *bridge device* on both partitions. The two *bridges* must communicate at intervals to exchange packets. The communication interval is the same length as the delay of the *link* that would be otherwise used. In fact, a *bridge* has the same properties as a *link*, except that packets are transfered into or out of a partition, rather than within it. Figure 4.1 illustrates the idea.

With our approach, a modeller can manually create the network, assign network nodes to processors and then link them appropriately. However it is generally more convenient to use a network building module, described in Section 4.5.5, to automatically partition and link an entire network or subnetwork of nodes.

If the *bridges* between processors are chosen so that their delay is high, then the communication costs involved will be lowered. Long distance links typically have a high delay, which makes them good candidates for conversion to *bridges*. Of course, the constraints of the network topology may occasionally dictate that a low delay link be used.

It is important that the work is distributed equally between all processors. Otherwise a *bridge* on one processor will need to stall until the corresponding

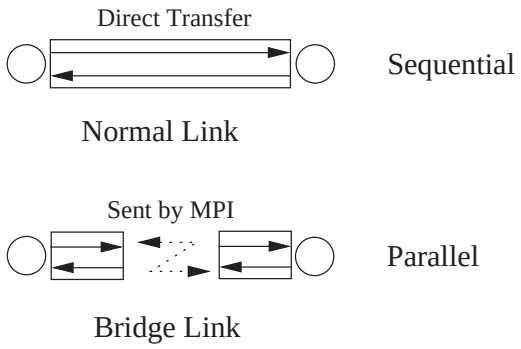



*Figure 4.1: In a normal* link *packets are transferred directly to the destination node after the link delay time. In a* bridge*, the packets are collected and later transferred using MPI message passing to the connecting* bridge *on the other processor. Both* bridges *then pass the packets on to the destination nodes. Interprocessor communication takes place at intervals of the link delay time.*

*bridge* on the second processor is ready to communicate. Psim does not provide dynamic load balancing: it cannot currently migrate nodes from a busy processor to a more lightly loaded one. However, offline load balancing is available. During a simulation the kernel tracks how much work each network node generates. In subsequent simulations this can be used by an automatic partitioning module to distribute the work more fairly. While the exact load generated by a node will vary from run to run, a backbone node, for instance, will always be busier than a peripheral end node. This scheme is quite successful in reducing communication waiting times.

A *mapreader* module, described fully in Section 4.5.5, can read topologies as generated by tools such as GT-ITM [19] and Inet [95].

We developed two modules for traffic generation. One is a simple connectionless packet source and sink. It alternates between *on* and *off* phases. During the *on* phase it emits a train of packets with a specified interdeparture time. When it has sent the required number of packets it switches *off*. After a period of time it turns *on* again and starts sending packets to a new destination. This can be viewed as a simple UDP traffic source.

The second traffic module developed is an implementation of the TCP protocol. A number of simplifications were made. For instance a node is either a client or server, but not both. However, it follows most of RFC 793 [50], and implements *slow start* and *congestion avoidance* [52].

## 4.3 Kernel of the Simulator

The kernel of the simulator consists of a framework for building network components and a discrete event handler.

Five core C structures describe an interface through which network components exchange data traffic. A module for a network component fills in the members of these structures with the appropriate values and function pointers. Network components communicate with each other solely through this API.

Events are either tied to a particular network component or components, for instance a TCP timer, or are global events such as the *stop* event that terminates the simulation. The event simulator on a processor stores events in a single time ordered event queue, implemented as a splay tree [54, 90] (see also Section 4.3.2). The event simulator does not itself perform any synchronisation between processors in a parallel simulation. All necessary synchronisation is performed by special modules. These modules perform interprocessor communication: exchanging packets that cross interprocessor links and blocking event execution on a processor when necessary to preserve causality.

### 4.3.1 Network Component Structures

There are five core structures which enable generic network devices to be described and linked together. In addition a mechanism for defining more general properties and behaviour is provided so that specialised network *devices* can be built as extensions of the generic *device*. A module simulates the behaviour of a piece of network equipment using this framework. The core structures in the framework are:

**class** One instance of this structure exists for each module, and is initialised when a module is loaded. It contains pointers to functions for initialising *subclasses* and *devices*. Examples of a network *class* include routers, links and traffic sources.

**subclass** A *subclass* is a specialisation of a *class*. For instance a *subclass* of a TCP module might be either a server or a client, or might be a client with a different request rate to another client.

**device** This is an actual instantiation of a *subclass* (and hence of a *class*). It

provides an interface for linking to other, compatible, *devices* and may have storage space for keeping track of its state as the simulation proceeds.

**outpoint** A *device* may have one or more *outpoints*. These contain pointers to the *inpoints* of other *devices*.

**inpoint** A *device* may have one or more *inpoints*. An *inpoint* provides a pointer to a handler function that is called whenever a packet is passed to the *inpoint*. This function takes the appropriate action for the module in question.

The most important fields from the core structures are shown in Code Fragments 1 to 7. Appendix B presents the API in detail and illustrates its use by building a simple network *device*.

```
struct class {
  char *name;
  struct subClass *(*xmlDefineSubClass)
    (struct class *type, xmlDocPtr doc, xmlNodePtr params);
  int (*initDevice) (struct device *node, struct network *net,
    xmlNodePtr params, xmlDocPtr doc);
  int (*delSubClass) (struct subClass *subClass);
  int (*firstEvent)
    (struct device *d, struct eventList *eList, int state);
  int (*postProcess)(struct device *d, struct network *net);
  int parallel;
};
```

*Code Fragment 1: Class structure*

```
struct subClass {
  char *name;
  struct class *class;
  int numInPoints;
  int numOutPoints;
  struct inPointTemplate *inProto;
  struct outPointTemplate *outProto;
};
```

*Code Fragment 2:* Subclass *structure*

```
struct device {
  char *name;
  struct subClass *subClass;
  void *data;
  struct inPoint *inPoints;
  struct outPoint *outPoints;
};
```

*Code Fragment 3:* device *structure*

```
struct outPointTemplate {
  char *name;
  enum dataType type;
  enum pointType pointType;
};
```

*Code Fragment 4:* outPointTemplate *structure*

**Rationale**

Memory efficiency was a primary motivation in the design of the simulator kernel data structures. A *device* structure is created for every component in the network model — whether link, router, traffic source or sink. It must be small. As far as possible any data common to more than one *device* has been abstracted to a higher level — the *subclass* most often. For instance, if a network contains a large number of 100BaseT Ethernet links, there is no need for each device to store the value of the bandwidth in its own state space, when a single copy can be stored in the *subclass* parameter storage space.

Each *device* is 38 bytes at a minimum, an *inpoint* is 12 bytes, and *outpoint* 8 bytes. Therefore a *device* with three links will consume 98 bytes. A complex node that needs to maintain state will require more memory. Nevertheless, this scheme is extremely lightweight: one million *device* structures with three connections each would need less than 100MB of memory.

### 4.3.2 Event List

Each processor in the simulation maintains its own time ordered event list. The event list uses a splay tree [54] [90] to order the events. This is an efficient data

```
struct outPoint {
  struct inPoint *inPoint;
  struct outPointTemplate *template;
};
```

*Code Fragment 5:* outPoint *structure*

```
struct inPointTemplate {
  char *name;
  enum dataType type;
  int (*handler)(struct device *device, struct inPoint *in,
    struct packet *p, double time, struct eventList *eList);
};
```

*Code Fragment 6:* inPointTemplate *structure*

structure for maintaining a pending event set. It is a self adjusting binary tree, that applies a simple restructuring heuristic called *splaying* whenever the tree is accessed. Insertion and deletion of events from the tree takes $\mathcal{O}(\log N)$ time per operation where $N$ is the number of events in the list. However, for very large numbers of events a calendar queue might have been more efficient. Section 4.8.1 discusses the performance of the splay tree as model size is increased. Each event has associated with it a pointer to the *device*, if any, responsible for the event and a pointer to the function that will process the event.

### 4.3.3 Communication between Network Devices

*Devices* communicate through the API provided by *inpoints* and *outpoints*. The typical data type passed between two linked *devices* is a packet (Code Fragment 8). In keeping with the spirit of the rest of the kernel the *packet* structure is also striped to the essentials. It has fields to record source, destination and packet size. In addition since TCP/IP traffic is most common, it has fields for sequence and request numbers and a control flag.

```
struct inPoint {
  struct device *device;
  int count;
  struct inPointTemplate *template;
};
```

*Code Fragment 7:* inPoint *structure*

```
struct packet {
  int srcgid, dstgid;
  TAILQ_ENTRY(packet) queue;
  short seq ,req, size, ctl;
};
```

*Code Fragment 8:* packet *structure*

## 4.4 Loading and Configuring a Network

Prior to running a simulation, Psim must load a description of a network. We decided to use XML (the eXtensible Markup Language) for the network configuration file. XML has a number of advantages over a handcrafted parser. It is a standard for creating markup-based, structured, extensible documents. This is advantageous for modularising the simulator. The Psim kernel understands and parses top level constructs, but module specific configuration details are passed to the module in question. The structured nature of XML greatly eases the separation of network level configuration from module level configuration while its extensibility allows each module to define a rich configuration language for itself without requiring modification to the network level configuration language.

The basic structure of a network description file, as understood by the simulator kernel is shown in Code Fragment 9. A typical network would define several *subclasses* of the main modules and have many *devices* attached together.

Normal *devices* are assigned to a single processor. If the *proc* attribute does not match a processor's MPI rank, then it ignores that *device*. *Devices* of the *mapreader* and *bridge class* are initialised on all nodes, as they involve either interprocessor communication or creation of other *devices*.

A number of tools exist to validate XML files against a Document Type Definition (DTD). By creating a DTD for our network markup language we can verify

```
<?xml version="1.0"?>
<Network>

  <Subclass class="module name" name="subclass name">
    Parameters describing this subclass. Parsed by the class.
  </Subclass>

  <Device type="subclass name" name="device name"
       proc="processor number">
    Parameters describing this instantiation of the subclass.
    Parsed by the subclass.
  </Device>

  <Attach>
    <Node name="device name1"
       inport="inpoint name" outport="outpoint name"></Node>
    <Node name="device name2"
       inport="inpoint name" outport="outpoint name"></Node>
  </attach>

</Network>
```

*Code Fragment 9: Network configuration example*

the syntactic correctness of a configuration file prior to running a simulation.

We currently create network configuration files by hand. However, with the structured design that XML facilitates, it would not be difficult to create networks through a graphical interface.

Nevertheless, despite the structured design and extensibility of the configuration language, creating a large network, with perhaps tens of *subclasses* and thousands of *devices*, would be a tedious process — even with the aid of a graphical interface. Much of this work can be automated using the *mapreader* module described in Section 4.5.5.

## 4.5 Network Modules

In this section we describe the commonly used network modules.

### 4.5.1 The Link Module

The *link* module simulates a point to point bidirectional data link. It buffers packets, places them on the wire and sends them to their destination (after the appropriate link delay).

### 4.5.2 The Router Module

The *router* module provides a *device* that implements algorithmic routing. This *device* accepts packets on an input port, determines the next hop that the packet needs to take and assigns the packet to the appropriate onward link. The *device* performs no buffering — the *link device* does that.

Each processor in a parallel simulation must maintain a full routing tree or trees, even though it does not simulate all nodes. This requires $\mathcal{O}(kN)$ memory per processor where $k$ is the number of routing trees maintained.

### 4.5.3 The Bridge Module

This module is the key component that enables parallel simulation in Psim. It represents a bidirectional point to point link. Two *bridge devices* replace the normal *link device* whenever two connected hosts are simulated on separate processors. To a packet it appears identical to the *link device*. Packets are buffered on entry to the *bridge* and the service time of this buffer is the time it takes to load the packet onto the wire: packet size divided by bandwidth. Packets arriving at a full buffer are dropped. Once a packet is on the wire, it arrives (after a delay corresponding to the type of link being modelled) at the remote host. This remote host is simulated on a different processor to the source host.

A single duplex connection between two nodes is represented by two *bridge devices*, one for each end of the connection.

Once a packet arrives at a *bridge*, the *bridge* must ensure that the remote processor receives the packet at or before the time the packet is due to arrive at the remote host. If the delay time is $\Delta$ seconds, then two *bridges* must synchronise at least every $\Delta$ simulated seconds. All packets that arrive and are processed

from time $t$ up to time $t + \Delta$ are stored and at time $t + \Delta$ are transferred to the corresponding *bridge* on the remote processor. This processor holds the simulation at time $t + \Delta$ until the packets have been transferred, then allows the simulation continue. Since the connection is bidirectional, the local processor may receive packets from the remote processor. These packets will have arrival times at the local host of between $t + \Delta$ and $t + 2\Delta$. We will refer to packets that need to be transferred from one processor to another as inter–processor packets

The above description outlines the basic operation of the *bridge device* but optimisations are necessary to avoid undesirable behaviour. There are two problematic aspects of the scheme as described above: each cross processor link, represented by two *bridge devices*, introduces extra communication overhead; secondly, the synchronisation introduces a rigid lockstep between processors.

The first issue is dealt with by collating data. Suppose a processor has several hosts with links to hosts on another processor. If these links are of the same type, with the same delay, then the packets to be transferred can be gathered into a single large message and sent together. This reduces the number of interprocessor communications needed and the associated overhead.

A conservative synchronisation scheme will inevitably introduce a lockstep between processors. Let $P_i(n)$ be the time it takes processor number $i$ to simulate the timeslice $(n-1)\Delta$ to $n\Delta$ ( where total simulation time $T = N\Delta$). In general the total run time will be at least $\max(\sum_{n=1}^{N} P_i(n))$. In other words, we can at best hope that the total time to run the simulation is the time taken by the processor with the most total work. This discounts communication overhead. With a rigid lockstep the situation may be worse: the total run time could be $\sum_{n=1}^{N} \max(P_i(n))$, again discounting communication overhead. This is particularly bad if there is a large variation in $P_i(n)$ on individual processors — even if the variation of $\sum_{n=1}^{N} P_i(n)$ is low.

Two measures were adopted to avoid the worst problems of conservative synchronisation. MPI provides asynchronous (nonblocking) communication interfaces. These were used wherever possible. This allows a processor to communicate with several other processors simultaneously. For example, suppose processor $A$ needs to transfer packets to processors $B$ and $C$. Processor $B$ has a lot of work to do and takes longer to reach the synchronisation point. If synchronous (blocking) communication is used then it is possible that processor $A$ may have to wait until processor $B$ has received its packets before it can commence sending to proces-

sor $C$. With asynchronous communication processor $A$ can send the packets to processor $C$ even if processor $B$ has not yet fully received its share.

In the basic synchronisation scheme described above, all processors exchange inter–processor packets at times $n\Delta,\ n \in [1 \ldots N]$. At these times a processor must collect all inter–processor packets, send them to connected processors, and wait to receive packets from those processors. If one processor is slow to reach the synchronisation point (perhaps due to a traffic surge in the portion of the network it models), all connected processors must wait for it. Our second measure to reduce overhead confronts this issue. It involves preemptive transmission of inter–processor packets. Each synchronisation period $\Delta$ is divided into $s$ *slots*. Processors now communicate every $\Delta/s$ simulated seconds. However the inter–processor packets sent at simulated time $(n + \frac{k}{s})\Delta,\ 0 \leq k < s$ do not need to be processed by the remote processor until $(n + \frac{s-1+k}{s})\Delta$.

The combination of asynchronous communications and preemptive transmission of inter–processor packets reduces the rigidity of synchronisation. If one processor has a sudden spike of work, other processors can continue the simulation to at least $\frac{s-1}{s}\Delta$ simulated seconds beyond the slow processor. This relaxation permits the parallel simulation to proceed more smoothly. However, increasing the number of slots also increases communication overhead. A balance must be found.

Experiments to explore the behaviour of the parallel simulation using the *bridge* module are described in Section 4.7.

### 4.5.4 The TCP Module

The TCP module implements a TCP stack. It conforms largely to RFC 793 [50], and implements *slow start* and *congestion avoidance* [52]. However some simplifications were made. In particular

- A TCP *device* is either a client or server, not both.
- Only one connection between a given client–server pair is allowed. This removes the need to allocate port numbers to connections.
- All TCP initial sequence numbers are zero.
- Clients make connections to servers. The servers respond by sending a stream of packets. The connection is then closed. Interconnection time

and the amount of data sent can be specified.

- Clients can reorder packets that arrive out of sequence (due to a packet loss for instance), but can only handle one gap in the packet sequence at a time.

A client can be instructed to connect to a random member of a specified server *subclass*. In addition a client can be instructed to preferentially connect to a nearby server, allowing some control of the distribution of traffic in the network.

TCP requires that timers be set in order to trigger retransmission of lost packets. A timer event could be maintained for each active TCP connection. However, with large numbers of connections this proves expensive. Consequently a single periodic timer causes a sweep of a number of connections checking for expired timers. The frequency of such a timer, and the number of connections for which it is responsible, can be tuned so as to tradeoff accuracy and performance in a flexible manner.

### 4.5.5 The Mapreader Module

Creating a large network topology by hand is a slow, tedious process. The *mapreader* module automates the task. A *mapreader device* is a virtual *device* — it does not represent a part of the physical network or generate traffic for example. Instead, it creates and connects routers, links and traffic sources. As its name suggests, it reads a network map and generates the nodes and links present in the map topology. For example, it could read a map generated by Inet [95]. Inet generates topologies resembling the inter–Autonomous System topology of the Internet.

A *mapreader device* has several initialisation parameters, in addition to a map filename, two of which are required: node type and link type. A node is created for each vertex in the map graph and vertices connected in the graph have their corresponding nodes connected by the specified *link device*.

The power of the *mapreader device* is that the node type can be a *mapreader device* itself. To use the example above, the primary *mapreader device* might take an Inet file as input to create an AS level topology. Each vertex in this topology causes the creation of a node. If this node is a *mapreader device* which reads a map of a typical AS topology, then we have very easily created a two level hierarchical model of an Internet like network. The nodes of the second level *mapreader device* might be routers, or might be yet another *mapreader device* to create a third level.

Traffic sources or sinks can be connected to the nodes of a *mapreader device*. In the example above TCP servers might be connected to the first level nodes while TCP clients might be connected to the bottom level nodes.

Code Fragment 10 illustrates the syntax used to create an instance of the *mapreader device*.

```
<Device type="map" name="Name of Network" proc="0">
  <map filename="Filename of topology" type="itm"></map>
  <node type="name of a router"></node>
  <link type="name of a link to connect nodes"></link>
  <traffic>
    <source type="Name of a source or sink"></source>
    <link type="Name of link that connects sources to nodes"></link>
    <distrib name="Name of distribution to decide number of sources">
      Parameters for distribution
    </distrib>
  </traffic>
</Device>
```

*Code Fragment 10:* mapreader *configuration example*

The simplified XML in Code Fragment 11 creates the network shown in Figure 4.2. It consists of a *mapreader device* that creates three connected nodes (using the topology from the file *triangle*). These nodes are *mapreader devices* also, each of which creates a tree of routers with one client and one server on each. A far larger and more complex network could be created with equal ease.

A further feature of the *mapreader device* is automatic partitioning of a network. When a *device* is created it is assigned to a processor. For example Code Fragment 12 creates a router called 'router012' that is to be simulated on processor number three in a parallel simulation. However in a large network it would be cumbersome to manually partition the nodes among processors, and then replace an ordinary *link* with a *bridge device* wherever two connected nodes are simulated on different processors.

The *mapreader device* can partition its map using the METIS graph partitioning library [55]. METIS attempts to create balanced partitions with minimum edgecut. Once the map is partitioned, the nodes, links, bridges and traffic sources are all created and assigned to the correct processor.

In addition, METIS can perform a weighted partitioning of a graph. This

```
<Device type="map" name="smallnetwork" proc="0">
  <map filename="triangle" type="itm"></map>
  <node type="map">
    <map filename="tree" type="itm"></map>
    <node type="router"></node>
    <link type="100baseT"></link>
    <traffic>
      <distrib name="constant"><value>1.0</value></distrib>
      <source type="tcpclient"></source>
      <link type="10baseT"></link>
    </traffic>
    <traffic>
      <distrib name="constant"><value>1.0</value></distrib>
      <source type="tcpserver"  store="yes"></source>
      <link type="100baseT"></link>
    </traffic>
  </node>
  <link type="100baseT"></link>
</Device>
```

*Code Fragment 11: A two level network, in which the highest level is created from the file called* triangle*, and each node in this is expanded to a network with a topology from the file* tree*.*

```
<Device type="router" name="router012" proc="3"></Device>
```

*Code Fragment 12: XML code to create a router* device *on processor 3.*

feature allows us to perform offline load balancing for networks created using the *mapreader device*. At the end of a trial run, the number of events that occur at each of the nodes of a *mapreader device* are recorded. These are stored and used as weights for partitioning in a future simulation. If different nodes have widely differing numbers of packets to process, then a weighted partitioning of the map may allow a more even distribution of work and hence a greater speedup in a parallel simulation.

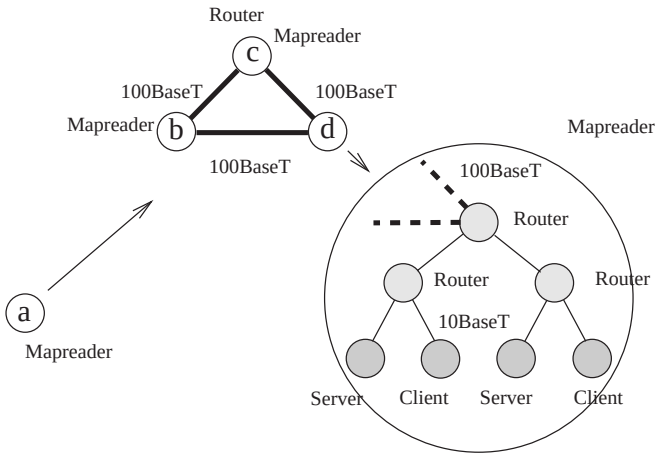



*Figure 4.2: The* mapreader device *a creates three connected nodes b, c and d. These are also* mapreader devices*. They each create a tree topology of three routers, two servers and two clients one of which is shown in the large circle.*

## 4.6 Parallelisation Issues

This section discusses some problems that arise in parallel network simulation but are not present in single processor simulation. The lack of full knowledge of the network configuration leads to what is known as the namespace problem, discussed in Section 4.6.1. Several different issues make exact repeatability of simulation runs on varying numbers of processors difficult. These problems are discussed in Section 4.6.2.

### 4.6.1 Global Namespace

Usually on a distributed memory parallel computer, each processor is responsible for simulating a small section of the network. A processor has full knowledge of its own part of the network, but little or no information about the rest. This makes efficient use of memory and allows very large networks to be simulated. However one problem does arise. Certain tasks require knowledge of the network outside a processor's own partition. For example, suppose a client node wishes to make a request to a random server node. The processor simulating the client may only simulate a small number of the servers present in the network, so will be unable to select a server at random. One solution is store a copy of the entire network on every processor. Though wasteful, this provides all necessary information to each processor. The approach taken in Psim is more efficient. The name of a node may

optionally be stored on all processors. For example, the names of routers do not need to be accessible everywhere, so no extra action is taken. The existence of a server does need to be known globally, so the name, though not other properties, of a server is stored on all processors. This allows a client to randomly select a server with which to communicate. The storage of names can be toggled on a per-*subclass* basis.

### 4.6.2 Repeatability of Simulation Runs

Consider a serial network simulator. If a network is simulated twice, using the same initial conditions, then its state should evolve identically. This is a desirable property for a parallel simulator also.

Indeed, for a parallel simulation it is also desirable that two runs with the same initial state, but different numbers of simulating processors should evolve identically.

Unfortunately this behaviour is far harder to achieve in a parallel simulation. Three factors cause problems: simultaneity, random seeds and routing. Let us examine each in turn.

#### Simultaneity of Events in a Parallel Simulation

It is conceivable that two events might be scheduled for the same time. In most cases this is not a problem, however in some instances the order in which these events are processed does matter. For instance a server might remove a packet from a full buffer and at the same instant a new packet might arrive. Depending on the ordering of these simultaneous events the newly arrived packet could validly be added to the buffer, or be dropped because the buffer is full.

In a serial simulation a simple rule, such as inserting events into the timeline after all other simultaneous events, will guarantee reproducible results between simulation runs. In a parallel simulation, and in particular using the parallelisation scheme provided by the *bridge device*, it is no longer as easy to order simultaneous events.

Reconsider the example above in a parallel context. As before a *process* and *deliver* event are scheduled for the same time. In the serial case the packet is delivered by a *link device*, in the parallel case it might be delivered by a *bridge device*. In the *bridge device* packets are stored and transmitted at intervals. This

affects the order in which events are scheduled. As the connections which are represented by *bridge devices* change according to the partitioning of the network, the order in which simultaneous events are processed also changes. It is important to note that any ordering of the events is valid, but that the state space evolution of two simulations may diverge.

In some situations this may just be an inconvenience, in others (such as code verification) it is a problem. We found no completely satisfactory solution. One approach is to give each event a *creation* time attribute in addition to its processing time. Simultaneous events could then be processed in order of their creation time (this is implicit in the ordering in the serial case). However there is nothing to prevent two events having the same creation time. Another approach is to base a secondary sorting on the lexical ordering of the nodes involved — for instance if simultaneous events are scheduled for nodes $p$ and $q$, then the event for $p$ is processed first.

However, since the performance of the event scheduling code is already of critical importance, we decided by default to add no extra code to order simultaneous events. The rationale is that any ordering is logically valid, and that simultaneous events that cause state space divergence are relatively rare.

### Random Seeds

In a serial simulation it is sufficient to maintain a single random number generator state and maintain reproducibility between runs. In a parallel simulation each processor maintains at least one random number state. This has the consequence that simulations using a different number of processors have a different sequence of random numbers and a correspondingly different simulation evolution.

Our solution to this problem was to associate a random number generator state with each *device* in the network. The state is seeded using an attribute (a *device's* name, for instance) that does not change. This makes the seeding independent of the number of processors used in the simulation. However, since saving random number generator state for each *device* can require considerable memory, it is a compile time option.

#### Algorithmic Routing in Parallel

Algorithmic routing constructs a tree using a subset of the connections present in a graph. In a parallel simulation the *router subclass* on one processor gathers topology information from all other processors, builds the tree, then redistributes it back to the other processors. Unfortunately in building the global graph, the order in which node connections are listed may change when the number of processors changes. This can cause the resulting routing tree to vary depending on the number of processors being used to run the simulation. One solution is to precompute the routing trees, save them to file, and load them from this file rather than computing them each time.

## 4.7 Small Network Experiments

In this section we begin by testing the performance of Psim on a medium size network of over one hundred thousand nodes. Following this we consider the impact of synchronisation and load balancing on the parallel performance. We conclude with less detailed demonstrations of Psim's scalability, using networks from one million up to ten million nodes.

### 4.7.1 Terms and Definitions

In this section we introduce some terms and definitions that will be used in following sections to discuss the performance of the simulator.

**Runtime** is the time taken to execute a simulation. We exclude the time required for model initialisation, as it is amortised over the time spent processing events, and its significance decreases with longer runtimes.

**A packet event** is the creation, routing or reception of a packet. We will often shorten *packet event* to just *event*. For example, suppose a packet is sent from a source to a destination, traversing three links and two routers. One creation, two routing and one reception events are processed for a total of four events.

**The Packet event rate** of a simulator is the number of packet events that a processor can execute per second of real time.

**Total packet event rate.** If each processor $P_i$ in an $N$ processor parallel simulation has a local packet event rate of $R_i$, then the total packet event rate is the sum $R = \sum_{1 \leq i \leq N} R_i$.

**The event count** on a single processor is the number of packet events processed by that processor in a simulation.

**The total event count** is the sum of the $N$ event counts in an $N$ processor parallel simulation.

**Work ratio** on an individual processor is defined to be the ratio of the time spent processing events to the total simulation time. In a sequential simulation it is unity, but in a parallel simulation, the time spent waiting at synchronisation points decreases its value.

**Work imbalance** is defined as

$$\max_j \frac{p_j}{\frac{1}{N}\sum_{i=0}^{N} p_i}$$

where $p_i$ is the event count simulated on processor $i$, in a $N$ processor parallel simulation. The higher the work imbalance, the lower the speedup of a parallel simulation.

**Global traffic fraction** is the fraction of network traffic that stays within the subnetwork of the originating host.

**Parallel speedup** is an indication of the efficiency of the parallelisation. We define the parallel speedup $S(N)$ of a $N$ processor simulation to be

$$S(N) = \frac{T_{\text{seq}}}{T(N)}$$

where $T(N)$ is the runtime of a $N$ processor simulation and $T_{\text{seq}}$ is the runtime of a sequential simulation. For some large models it is not possible to execute a simulation on a single processor. In this case we approximate $T_{\text{seq}}$ by $T_{\text{seq}} = nT(n)$, where $n$ is the smallest number of processors that can simulate the model. Linear speedup, where $S(N) = N$, is used as a baseline for many comparisons.

### 4.7.2 Offline Load Balancing

In Section 4.5.5 we introduced the *mapreader* module and described its ability to perform offline load balancing of the partitions it creates. To summarise, a *mapreader device* can record the number of events that occurs in each of its nodes. The next time the simulation is run, the *mapreader device* can pass this information to the METIS graph partitioning library. METIS then attempts to partition the nodes in such a way as to balance the work between processors and minimise the interprocessor connections.

The degree to which the load balancing can improve simulation runtime depends on the nature of the network. A regular network, in which the packet events are evenly distributed between nodes, will already be well balanced by the unweighted METIS partitioning. At the other extreme, if a very large proportion of the total packet events is concentrated in a small area, there is little that METIS can do to redress the balance. In addition, the greater the ratio of nodes to processors the better the balance.

In order to test the efficacy of the load balancing, we simulated one hundred different networks and compared the runtime with and without load balancing. Each network consisted of a one thousand node *mapreader device*. Each node of this *device* consisted of a subnetwork of ten routers. Attached to each router were five servers, plus twenty more at the connection point to the higher level. Each router also had a number of clients attached to it. This number was randomly generated according to a pareto distribution (mean 101, shape parameter 1). This distribution was chosen in accordance with research on Internet topology, showing that outdegree distribution follows a power law [91] [30]. On average each network had 121801 clients, or over ten per router. Each client made connections to a server, with a 40% chance that the server was within its own subnetwork. The off period was exponentially distributed with a mean of 0.5 seconds. A server sent an average of 1001 packets to a connected client — this transfer size was also pareto distributed.

Clients were connected to routers by a 10Mbps link. Servers were connected by a 1000Mbps link. Routers within a subnetwork were connected by a 1000Mbps link. Subnetworks were connected by OC-48 (2488 Mbps). The delay on the OC-48 links was 4ms. To avoid variations in routing between runs, there were no cycles in the network. Each simulation was run for ten seconds of simulated time.

The simulations were run on an eight processor cluster. There were four nodes

in the cluster each with two Pentium 1.0GHz processors and 512MB of memory. The nodes were linked with 100Mbps Ethernet. The cluster runs Linux kernel 2.4.18 SMP. GCC 2.96 was used to compile the code. The MPI implementation used was Mpich 1.2.

Figures 4.3 and 4.4 show the reduction in runtime and the reduction in work imbalance respectively. The average percentage reduction in runtime, $100 \times \frac{T_{before} - T_{after}}{T_{before}}$, is $12.1 \pm 0.9\%$, though in some cases the reduction was up to 35%. However in five of the one hundred trials the runtime actually increased by a small fraction. We speculate that this increase occurred in trials where the initial network was already well balanced and METIS was unable to improve the partition. This might be aggravated by the assumption that all events take exactly the same computational time to process. For instance, a packet creation may not take as long as a packet routing event and hence two partitions with equal numbers of events may have an unequal amount of real work.

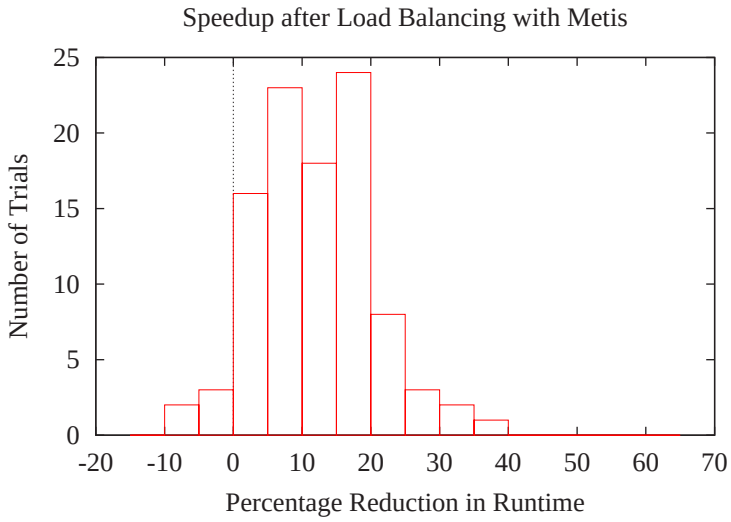



*Figure 4.3: In 95 out of a hundred trials, using the event count on nodes to balance the partition of nodes between processors resulted in a decrease in runtime.*

Figure 4.4 shows that the unweighted partition has a work imbalance that ranges from 1.2 to about 1.8. The values are evenly spread up to 1.4: values above this point are less frequent. In almost all cases the work imbalance is reduced after the weighted partitioning. The values are almost all clustered between 1.0 and 1.3

The average work imbalance with unweighted partitioning is $1.38 \pm 0.01$, with weighted partitioning it is $1.13 \pm 0.01$.

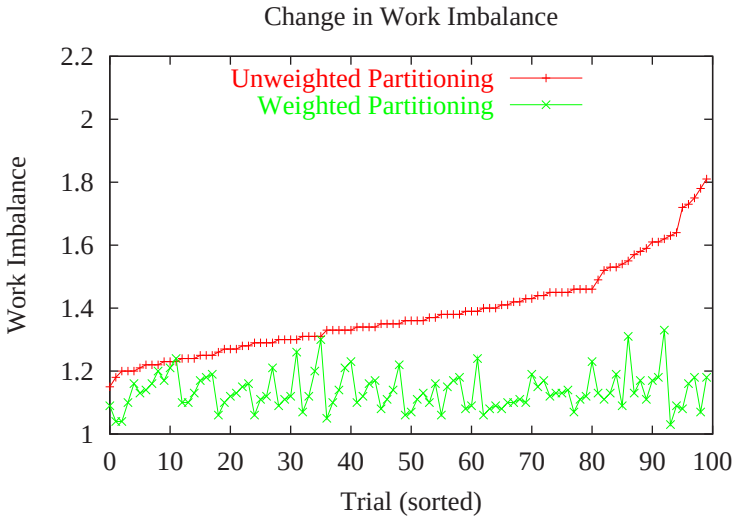



*Figure 4.4: A low work imbalance reflects an even partitioning of work between processors. Using the event count on a node in a weighted partitioning of the nodes results in a better work imbalance. The trials have been sorted by unweighted work imbalance for clarity.*

It is clear that the offline load balancing performed by the *mapreader* modules and METIS has a noticeable effect on the simulation runtime. The magnitude of the improvement will depend on the regularity of the network in question, but since the weighted partitioning improved the runtime in 95% of our trial networks, it is clearly worth using.

### 4.7.3 Time Slots in the Bridge Module

As discussed in Section 4.5.3, the *bridge* module uses asynchronous communication as one part of the effort to reduce the overhead of synchronisation. The second measure involves splitting the synchronisation period (link delay time) into $n$ slots and transferring packets at the end of each slot, rather than just at the end of each synchronisation period. Increasing the number of slots relaxes the rigidity of synchronisation, but increases communication overhead. We conducted an experiment to examine the impact that the number of slots has on the runtime.

The network has one thousand subnetworks, each of which has one hundred routers. Each router has one client and one server. There are a further twenty servers at the connection point between subnetworks. Servers and routers are connected by 100Mbps links, clients by 10Mbps links, subnetworks by OC-48

(2488 Mbps).

We modified two parameters of this network: global traffic fraction and slot number. The global traffic fraction varies from 1.0 (destination nodes chosen totally at random) down to 0.0 (destinations all within the same subnetwork as the source). We incremented it from 0.0 to 1.0 in jumps of 0.2. For each of the global traffic fractions we tested the network using one to five slots. Finally, each of these configurations was run ten times. We ran the simulations on the cluster used in Section 4.7.2 using all eight processors.

The results are summarised by Figure 4.5. Especially for a high global traffic fraction, using two slots results in the lowest simulation runtime. Above two slots and the runtime steadily increases, as the communication overhead rises.

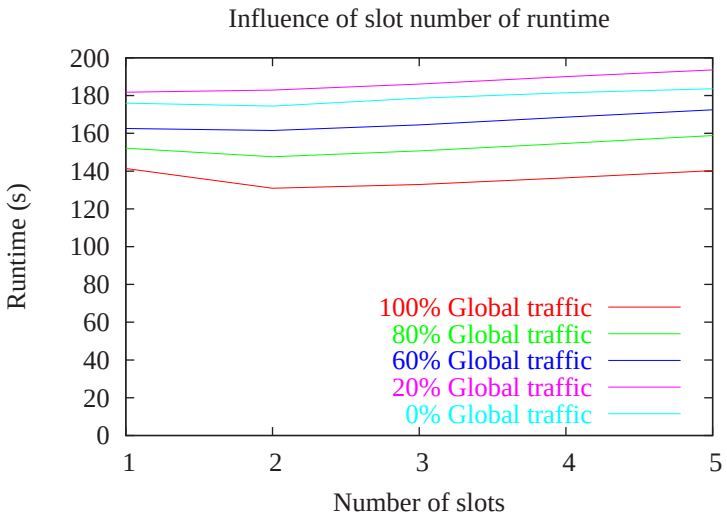



*Figure 4.5: By communicating more often than is strictly necessary (one slot per synchronisation period), a parallel simulation may avoid some blocking. Two slots is optimal for most situations. However if there is little interprocessor traffic (0 to 20% global traffic) then extra slots just introduce overhead.*

Using two slots appears to offer a small but noticeable advantage for high global traffic simulations. Nevertheless, this may vary between networks.

### 4.7.4 Parallel Speedup

Using the same network as in Section 4.7.3 we conducted a series of measurements to evaluate the efficiency of the parallelisation. For each traffic fraction (0.0 to

1.0 in increments of 0.2), we ran the simulation on one to eight processors. Each configuration was run ten times. The speedup graphs for each traffic fraction are shown in Figures 4.6 and 4.7. A quick examination immediately shows that Psim parallelises very well, at least up to eight processors.

However there are some initially strange features of the speedup graphs. All six graphs have a 'kink' after the addition of the fifth processor. Also, the speedup below five processors is actually superlinear; a simulation on $n$ processors is *more* than $n$ times faster than a simulation on one processor. These anomalies require explanation.

The kink at $n = 5$ is easily accounted for. When four or fewer processors were used for the simulation, one processor on each of the four nodes was used. Each node has two SMP processors. Now, a logical process in the Mpich implementation of MPI creates two real processes. One process is the simulation code, the other manages interprocessor communication. If one logical process is run on a two processor SMP node, then the two processors can share the MPI processes, and other system processes, between them. If there are two logical MPI processes on a single SMP node, then the two processors have four real MPI processes as well as the system processes.

With four or fewer processes, we ran one on each of the four SMP nodes. However, when a fifth MPI process is added, one of the four SMP nodes must then run two logical MPI processes. This accounts for the kink between $n = 4$ and $n = 5$.

The next anomaly is the super–linear speedup. This unexpected, but welcome, behaviour is due to increased efficiency as the partitions grow smaller. In particular, the size of the pending event set has an effect on the speed of the event handling code. As more processors are added, each processor has a smaller number of network *devices* in its partition, and correspondingly fewer events in its future event list. As the time for insertion and deletion of events depends on the number of events in the system, a small partition not only has fewer events to be simulated, but each event can be simulated more quickly.

This behaviour is explored more fully in the large network experiment of Section 4.8.

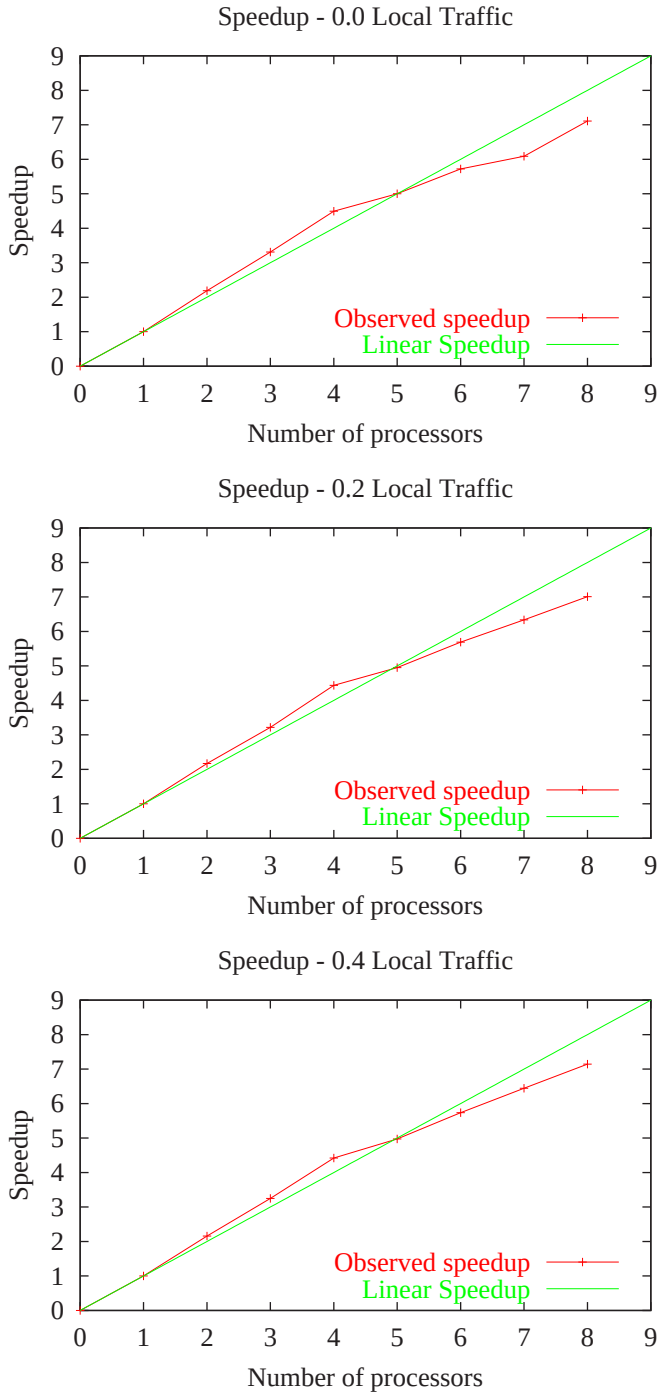



*Figure 4.6: Each point in the graphs represents the average of ten measurements. The error bars are too small to display.*

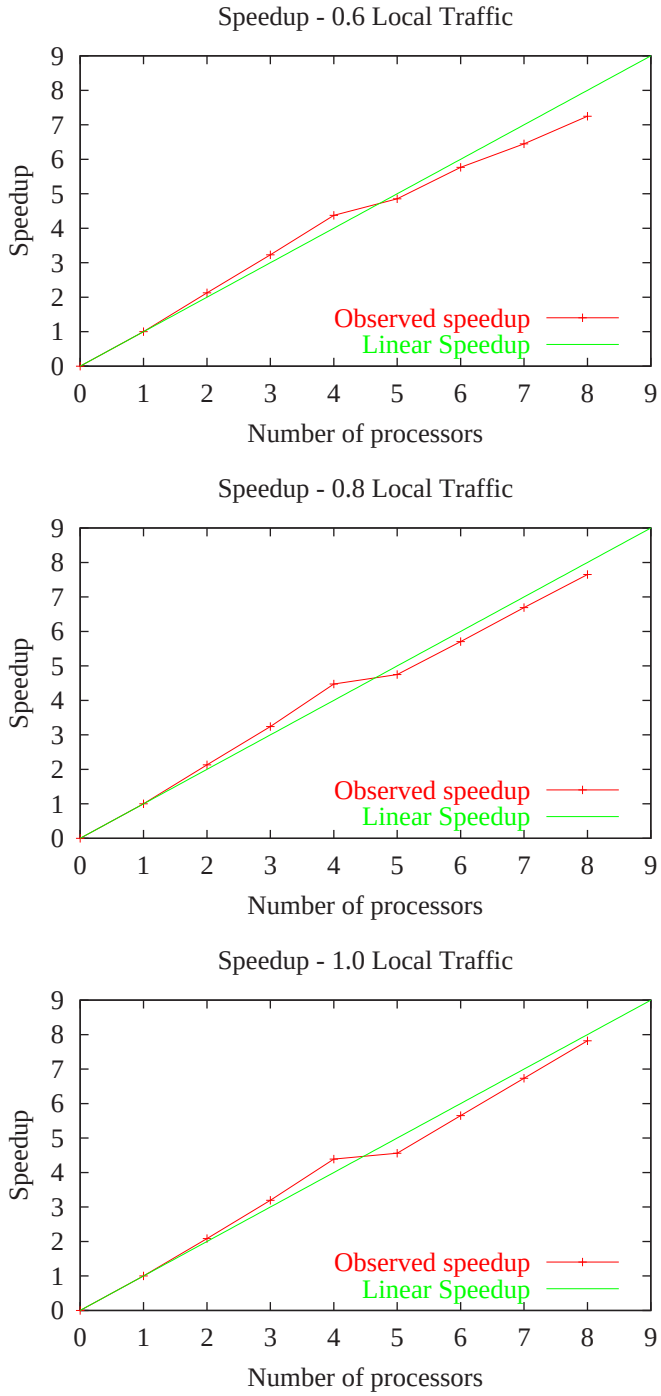



*Figure 4.7: Each point in the graphs represents the average of ten measurements. The error bars are too small to display.*

## 4.8 Large Network Experiments

In this section we test the scalability and parallelism of Psim. To find the extremes of its scalability we simulate networks with one million up to ten million nodes, and varying levels of traffic. To test the efficiency of the parallelisation we run these networks on one to sixty four processors.

The computer used for these experiments is again a cluster of dual Pentium nodes. Each node has two 1.0GHz processors and 1.0GB of memory. They are linked by Myrinet. The operating system is Linux kernel 2.4.18 SMP and the MPI implementation is Mpich 1.2.

### 4.8.1 One Million Nodes

The first network we will examine is a one million node network. It is split into one thousand subnetworks. These consist of one hundred routers, each with ten clients and one server. The subnetworks are connected by OC-96 links (4976 Mbps), routers and servers by 100Mbps Ethernet, and clients by 10Mbps Ethernet. This topology is not realistic, but will suffice for a performance analysis. The link delay on the OC-96 link is 2ms.

Clients make connections to servers. The server sends a file with pareto distributed size (average 1001 packets). When the client receives the file it sleeps for a period (exponentially distributed, mean 1 second). The global traffic fraction is 0.1 — most traffic stays within the source subnetwork.

We ran as many simulations as possible using one to sixty four nodes of the cluster. Each simulation was run for ten seconds of network time. The code used was not the latest version and lacked some of the features later developed. In particular, per processor rather than per *device* random number generators were used. This resulted in a varying number of events when the model was run on different numbers of processors. This complicates the performance analysis somewhat as some scaling is needed.

Figures 4.8 to 4.17 graph various aspects of simulation performance. As in the experiment of Section 4.7.4, we have achieved superlinear speedup, see Figure 4.8. Again, this is somewhat surprising. Several factors impact on the runtime of the parallel simulation. In particular the total number of events processed, the packet event rate and the work imbalance all have an affect on the parallel speedup. We will examine these in turn.

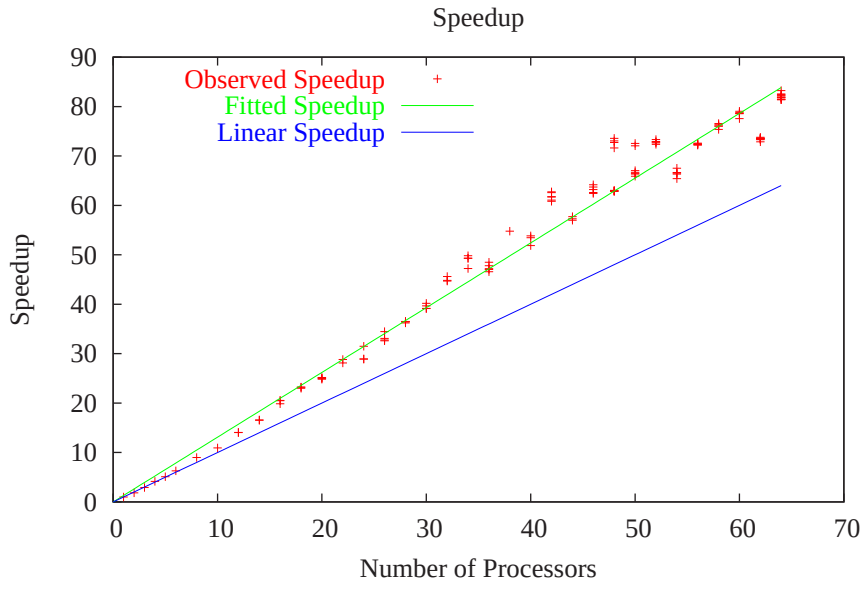



*Figure 4.8: Parallel speedup in the simulation of a network with one million TCP clients, one hundred thousand servers and one hundred thousand routers.*

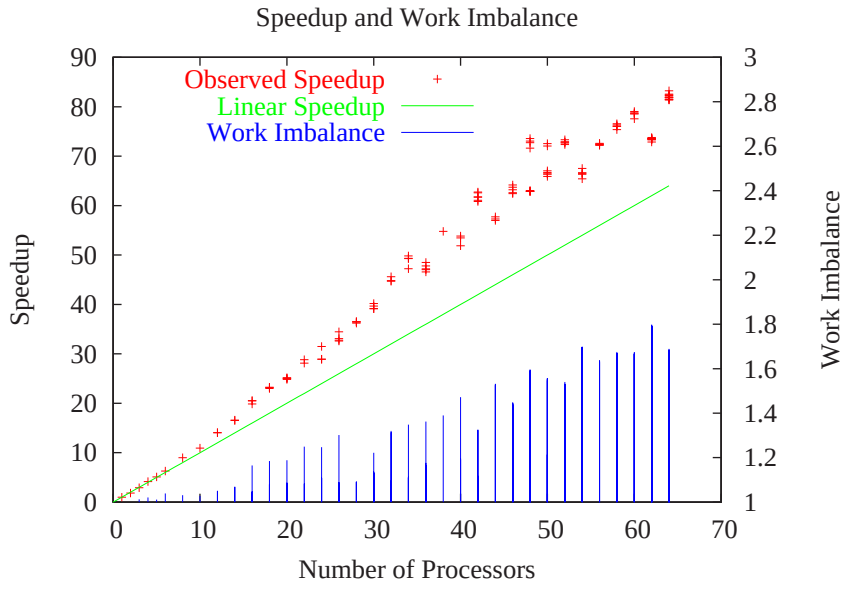



*Figure 4.9: Some correlation can be seen between a high work imbalance and a low speedup, for instance with 40, 62 and 64 processors.*

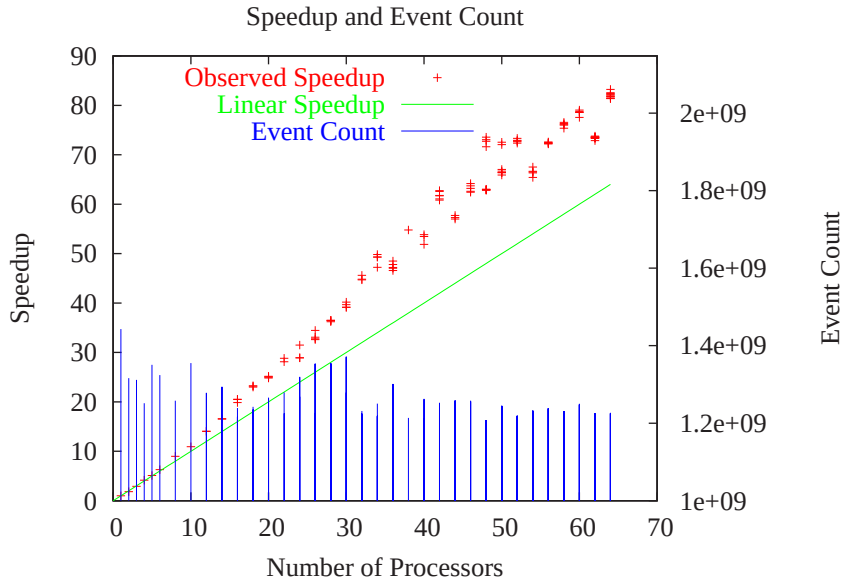



*Figure 4.10: There is less correlation between event count and speedup — the variation in event count is less than that of work imbalance.*

Figure 4.9 plots the speedup against the number of processors, and on the same graph plots the work imbalance. Some correlation between a relatively high work imbalance and a relatively low speedup is visible. This accounts for some of the local variations in speedup, but not for the overall trend.

The next graph, Figure 4.10 also plots the speedup but this time with the total event count for the simulation. Again, some correlation can be seen.

However it is the packet event rate of an individual processor that has the closest correlation with the superlinear speedup. Figure 4.11 shows the speedup and the packet event rate of processor zero in an $n$ processor simulation. Figure 4.12 shows the maximum, minimum and average packet event rate of each simulation. The single processor event rate in the sixty four processor simulation is almost double that of a one processor simulation. Figure 4.13 displays the total event rate. As with the parallel speedup, this is superlinear. It shows that the code can reach a total packet event rate of almost $16 \times 10^6$ packet events per second.

This greater efficiency with many processors can be attributed to several factors. As the number of processors increases, the size of each partition falls. Hence a larger proportion of the network state can be held in cache. Secondly the efficiency of the splay tree structure used varies with the number of events in the tree.

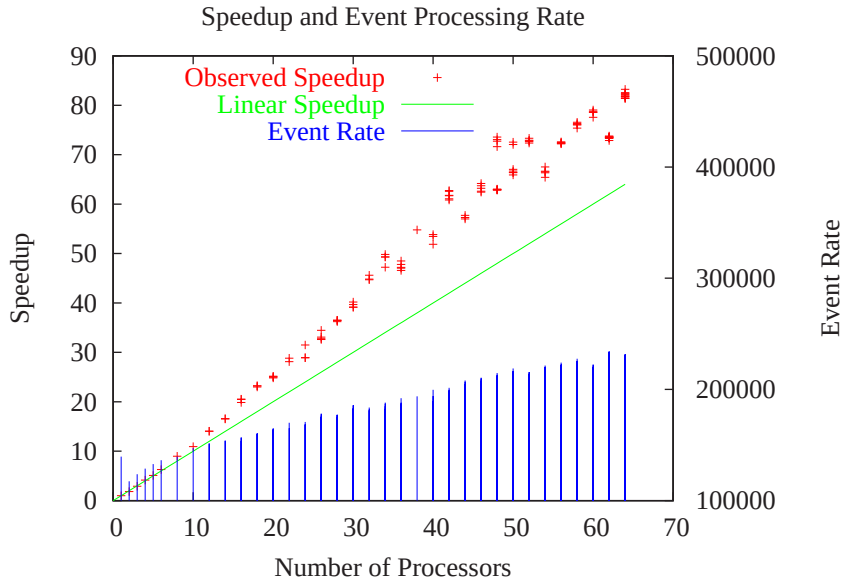



*Figure 4.11: There is a close correlation in the general trend of the event rate on the individual processors and the parallel speedup.*

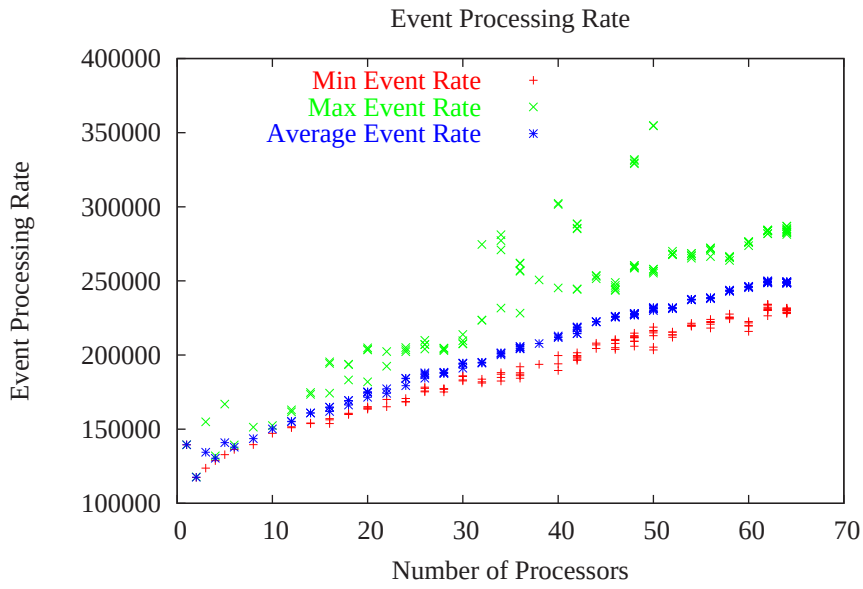



*Figure 4.12: The event processing rate on a processor increases as the number of processors is increased. With more processors each partition is smaller. The splay tree ordered event list has an efficiency that scales with $\mathcal{O}(\log N)$, where $N$ is the number of events. Thus, it can process events faster in a small partition.*

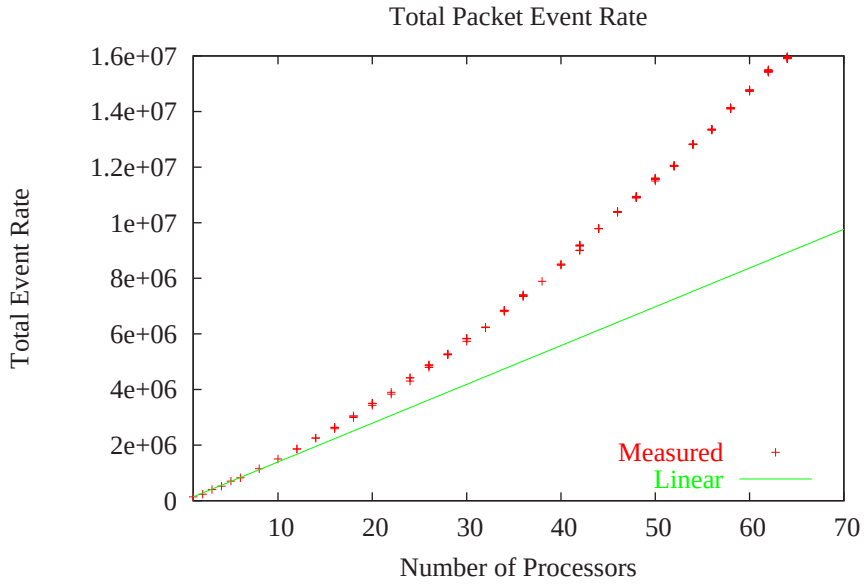



*Figure 4.13: The total event rate has a superlinear increase, as is expected if the splay tree operates more efficiently with a lower number of events.*

It takes $\mathcal{O}(\log N)$ time to insert an event into a splay tree. Thus, a small partition, with a small number of pending events, can be simulated more efficiently than a larger partition. A calendar queue [14], or a variation thereof, might reduce this effect. Calendar queues offer close to $\mathcal{O}(1)$ average performance, although some distributions of event times produce suboptimal performance.

It would be preferable to view the performance of the simulator unbiased by the above factors. To achieve this we scaled the speedup to account for the changes in event rate and event count. The scaled speedup is the speedup that would be achieved if each run processed exactly the same sequence of events and the processing rate on an individual processor remained constant.

The scaled speedup is determined as follows. Let $C_n$ be the total event count in a simulation run on $n$ processors. Let $C = \max_n C_n$ where $1 \leq n \leq 64$. Let $R_n$ be the highest per processor packet event rate in a $n$ processor simulation. Then the scaled time taken on $n$ processors is

$$T_n = \frac{C}{R_n}$$

This scaled time can be used to calculate the scaled speedup and total event rate.

This is a crude approximation, but useful. It provides a worst case scenario

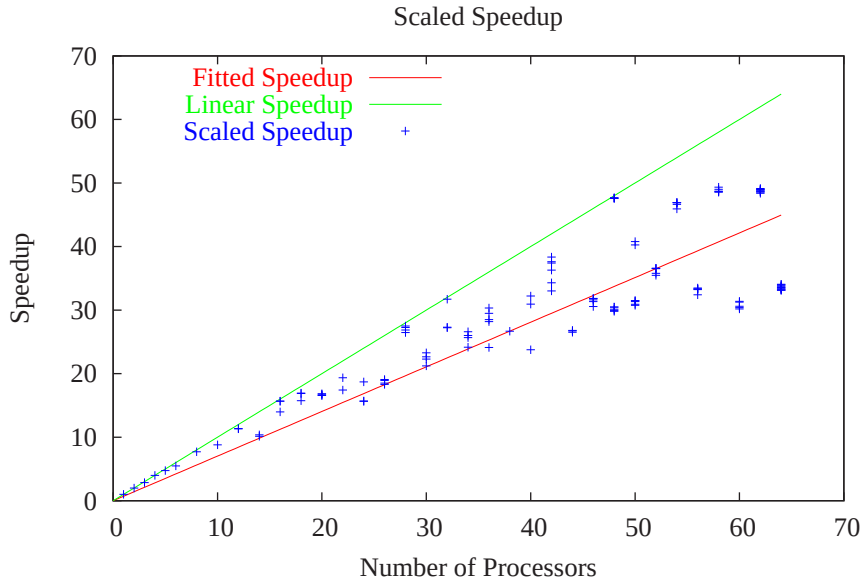



*Figure 4.14: The scaled speedup is sublinear. This speedup is obtained by calculating the time each simulation run would take if the event processing rates and total event counts were identical.*

for the efficiency of the parallelisation. The revised speedup graph is plotted in Figure 4.14. The speedup is now sublinear — as might be expected. The initial unscaled speedup is $1.310 \pm 0.007$, the scaled speedup is $0.70 \pm 0.01$. While the speedup has dropped considerably, it is still respectable for a sixty four processor cluster.

In order to investigate the effect of the splay tree event list we isolated the event handling code from the network simulation code and performed several tests. The first test added $N_i$ events to an event list, then removed the first event and added a new one. This insertion and removal step was repeated a million times in order to calculate the average time taken to perform the action. Figure 4.15 plots the average insertion and removal time for event lists with between ten and a million events. This clearly demonstrates the drop in event handling efficiency of the splay tree with large event lists.

In the second test, Figure 4.16 illustrates the performance of the splay tree in a parallel situation. For event lists of three different sizes, the list was partitioned between processors. The event rate on a single partition was measured by performing a million event insertion and deletion actions and used to calculate a total event rate. It can be clearly seen that the total event rate has a superlinear

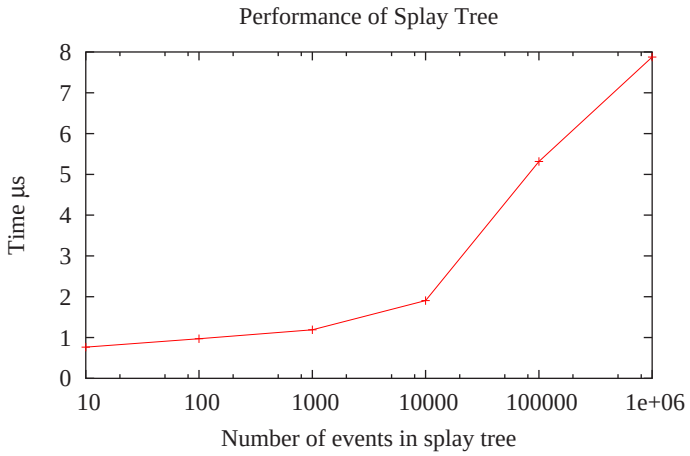



*Figure 4.15: As the number of events in a splay tree event list is increased, its efficiency drops.*

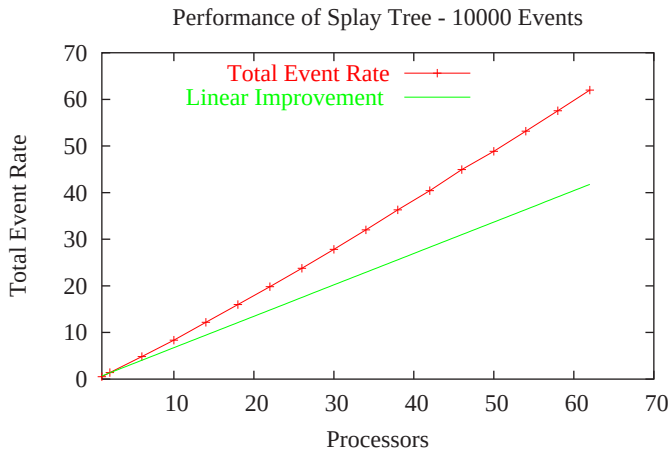
increase for increasing numbers of processors. Moreover, it has similar behaviour to the speedup in the full network simulation, Figure 4.13.

Figure 4.17 displays the work ratio. For each simulation run we have displayed the maximum, minimum and average work ratio. The maximum work ratio remains close to unity for all numbers of processors running a simulation. This is reassuring as it shows that little time is wasted on communication overheads. However the average work ratio drops almost linearly from unity, at one processor, to approximately 0.6 at sixty four processors. This is expected; as the number of processors is increased the work imbalance will increase and reduce the work ratio.

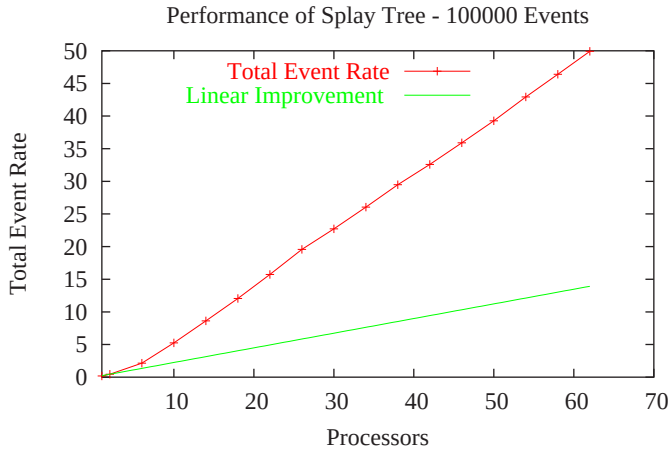
### 4.8.2 Larger Network Demonstrations

In this section we demonstrate some large simulations. These simulations, with up to ten million nodes and up to one hundred million flows, demand enormous resources. Consequently it was not possible to perform experiments at the same level of detail as in Section 4.8.1. We include these models to illustrate the extreme scale with which it is possible to model a network using Psim.

a)




b)




c)

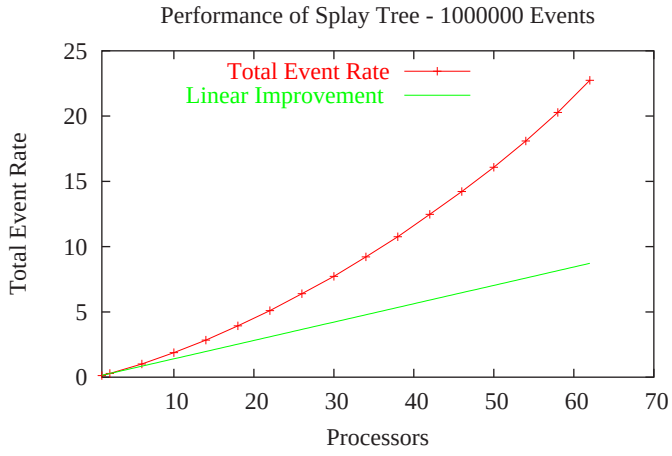



*Figure 4.16: The total event processing rate when performed on n processors shows a superlinear increase, since the efficiency of the splay tree increases as the number of processors increases and the size of an individual list decreases.*

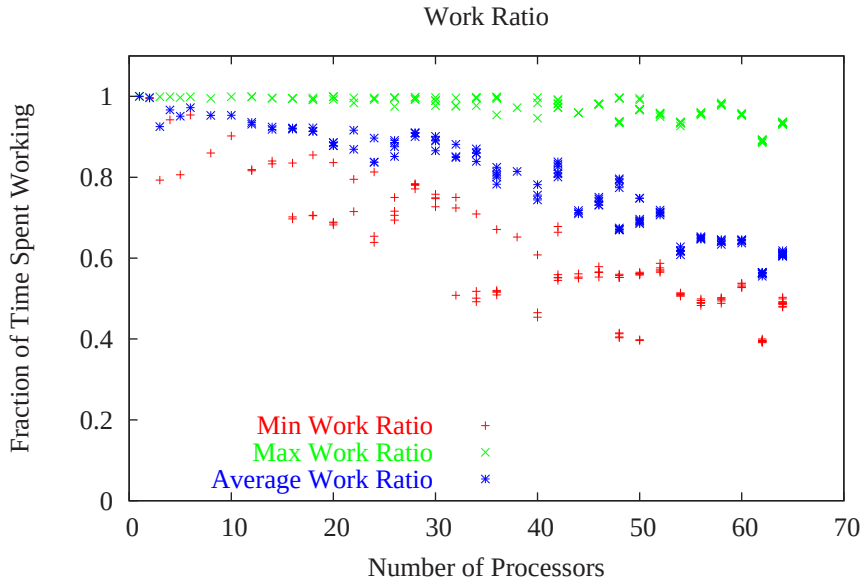



*Figure 4.17: Processors with a large, or busy part of the network have a work ratio near one. Those with a small or quiet portion have a lower work ratio.*

### One Million Nodes, Ten Million Flows

This network is identical to that of Section 4.8.1, except that each client can open up to ten simultaneous connections. This increases the traffic load in the network. The speedup and work ratio graphs are plotted in Figures 4.18 and 4.19. Since it was not possible to simulate this network (and the two following networks) on a single processor, the speedup was calculating using a value for a single processor simulation runtime $T(1)$ approximated by $T(1) = nT(n)$, where $n$ was the smallest number of processors capable of simulating the network. This approximation does not take account of the impact of the $\mathcal{O}(\log N)$ cost of event list insertions and deletions on runtime. Hence superlinear speedup is not as visible as in the smaller simulations.

### Ten Million Nodes, Ten Million Flows

This network is a larger version of that in Section 4.8.1. It has one million rather than one hundred thousand routers. Each router is connected to ten clients and one server. The simulation was run three times: on thirty two, forty eight and sixty four processors. The speedup was calculated by estimating the runtime of a single processor simulation from that of the thirty two processor simulation.

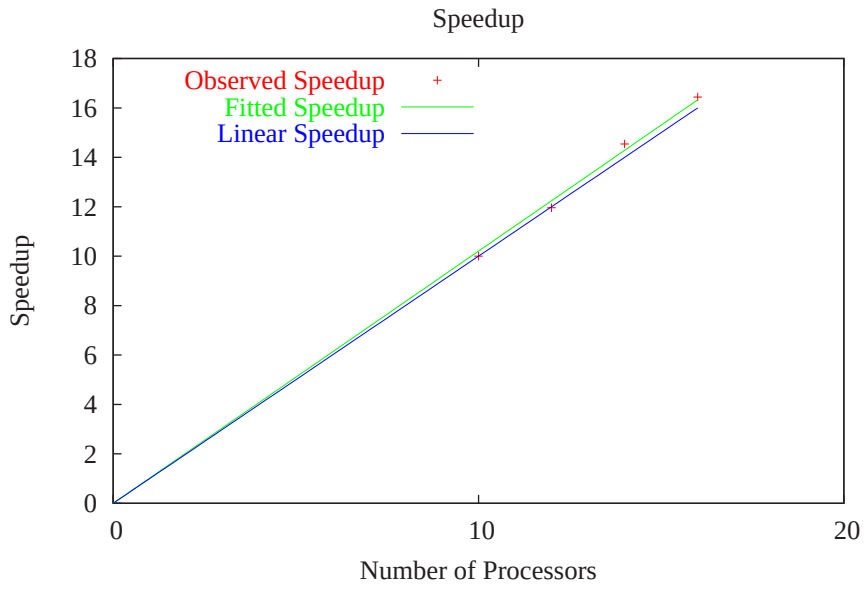



*Figure 4.18: The network simulated here is the same one million node network as before. However the number of TCP connections has been increased by a factor of ten. The speedup is lower, but this is partly because the single processor runtime had to be estimated from the ten processor runtime.*

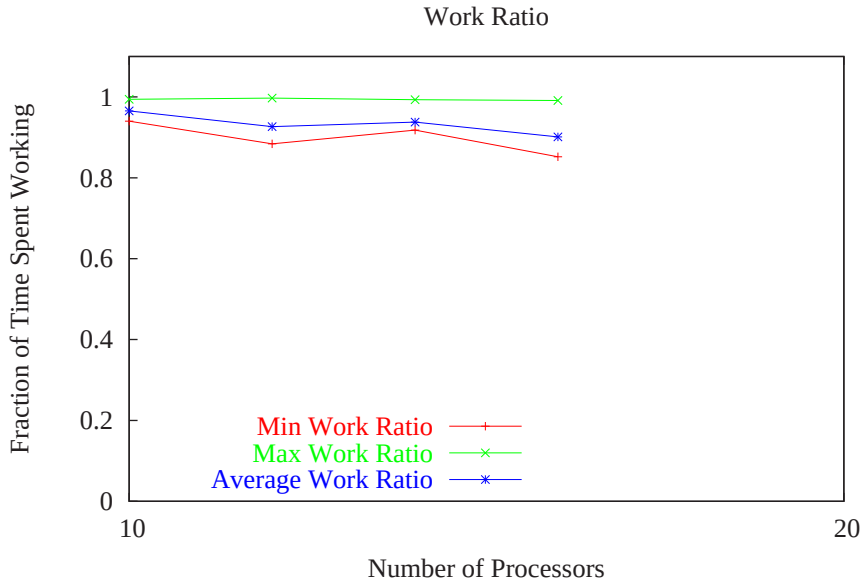



*Figure 4.19: The work ratio is high for this heavy traffic network.*

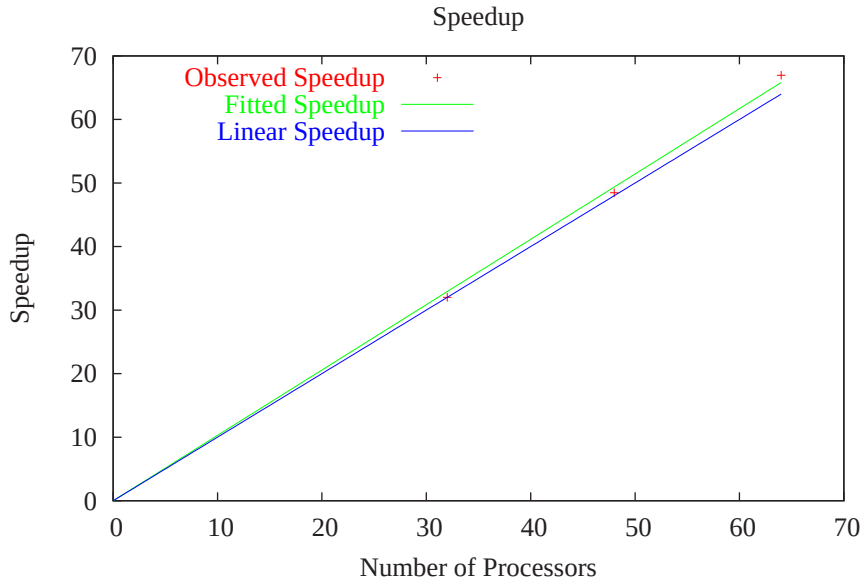



*Figure 4.20: This is the speedup graph for a network with over ten million nodes.*

Figures 4.20 and 4.21 show the speedup and work ratios respectively. The speedup is just superlinear. The work ratio also remains high. The maximum work imbalance is 1.4, at sixty four processors. This is lower than that of the smaller network. This is expected since it is easier to balance the partitioning in a larger network.

### Ten Million Nodes, One Hundred Million Flows

We could perform only one simulation run of this model. It has the same topology as above, but each client can now open ten connections. On sixty four processors it took just over five and a half hours to simulate 18 seconds of network time.

## 4.9 Summary

With the growth of the Internet, interest has arisen in its large scale behaviour. As its size increases, and the range of applications using it diversifies, so too do unforseen phenomena emerge. For instance, congestion storms, route flapping and virus propagation all call for further study.

Simulation is one of the primary tools used in studying networks. However the difficulties of simulating a network grow with its size. With the hardware

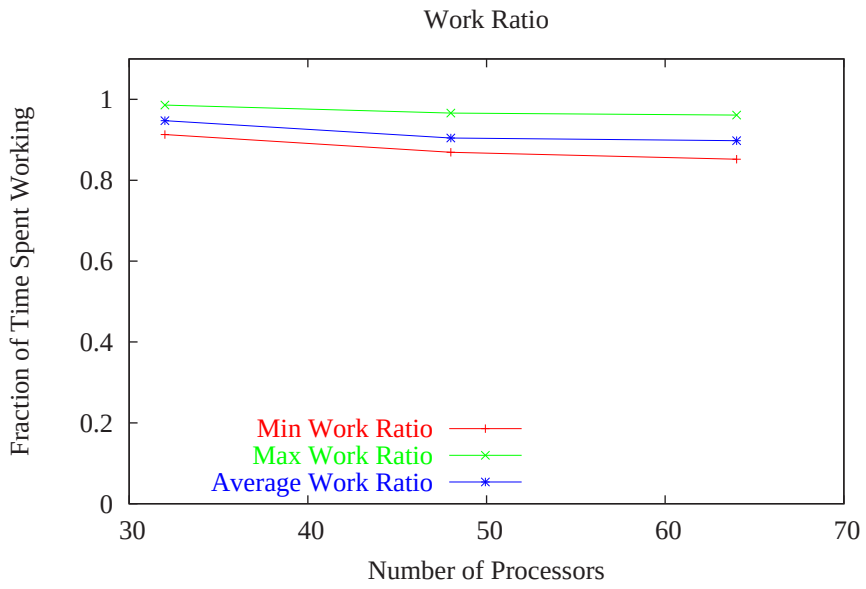



*Figure 4.21: This is the work ratio graph for a network with over ten million nodes.*

resources currently available it is not feasible to simulate Internet scale networks using conventional techniques. The sheer size of the Internet prohibits its detailed simulation by, for example, sequential discrete event simulation. The complexity of its underlying protocols has hindered the development of analytic models.

The problem has been attacked on two fronts. One approach involves increasing the computational power by harnessing many processors at once, typically using the methods of parallel discrete event simulation. The other approach advocates increasing the abstraction of a model, accepting that some approximation will be introduced into the model, but seeking to minimise its effect on behaviour of interest.

Parallel discrete event simulation in network modelling has met with mixed success. The overheads of the event handling system are high, and often the method does not scale well to many processors. On the other hand, abstraction methods, if applied too freely, can reduce the accuracy of a model and even eliminate the behaviour under study.

We believe we have struck a balance in our implementation, Psim, of a network simulator. Both abstraction and parallelisation techniques are used. In particular algorithmic routing is used to avoid the cost of per node routing tables. Our

parallelisation scheme combines aspects of window based synchronisation and null message based synchronisation. However it avoids the global minimisation need at each step in window based synchronisation and generates fewer null communications than a pure null message approach. In addition the complexity of synchronisation is removed from the performance critical event handling code and managed directly by a *bridge* module representing a network link. This has the advantage of confining interprocessor communication to just the area in which it is needed.

We have also implemented offline load balancing in the simulator. This analyses prior simulation runs so as to enable a better division of labour between processors in a future simulation.

The simulator models TCP traffic in high speed wired networks. We demonstrate an unrivalled degree of scalability, both in terms of number of simulated nodes, and in the number of simulating processors. A single processor is capable of simulating over one hundred thousand nodes. With sixty four processors we can simulate ten million nodes. This is an order of magnitude larger than previously recorded. Even with sixty four processors, and scaling the results to take into account the worst case performance of the event handling code, the simulator exhibits a half linear speedup.

The scalability of Psim offers new opportunities to study the behaviour of large scale networks by combining the power of abstraction techniques and parallel computing. Its modular design allows easy extensibility. Using Psim it is now possible to study the behaviour of many types of large scale networks using simulation.

# Chapter 5

# Conclusions

## 5.1 The Importantance of Network Simulation

As the Internet continues to grow, it is necessary that we develop techniques to simulate it in as large a scale as possible. Network simulation is important for not only the testing of network designs and protocols, but also as a tool to aid understanding of the dynamics of network behaviour. Since the modelling methods used in the past either do not scale to the size required, or are unable to model the complexities of today's Internet, much research has been focused on developing new techniques for large scale network simulation.

## 5.2 Aims of Thesis

Our work is presented in the previous two chapters. Let us now review the aims set forth in Chapter 1, and consider to what extent they have been achieved. We wished to model Internet like networks, characterised by:

- Large sizes - at least one hundred thousand nodes.
- Complex network protocols such as TCP.
- High speed, and high bandwidth links.

We have demonstrated in Chapter 4 that we can simulate TCP networks with up to a million routers, ten million hosts, and heavy traffic. A network simulation of this size requires the resources of a large parallel processing cluster. However,

smaller networks with hundreds of thousands of hosts can be simulated on a single processor.

What are the contributions that made this possible? Algorithmic Routing (AR) is the principle technique that allowed the extreme scaling. Without the memory efficiency of the method, the size of the routing tables would have prohibited the simulation of the larger models. However in order to make AR viable, we needed to introduce several techniques to enhance its performance and improve the quality of routes generated.

Our contributions to AR are:

- A modification to the original method that represents the routing tree in a different manner, reducing memory usage and increasing performance (direct AR).
- A new fixed computational cost routing algorithm.
- A fast, efficient, method for improving routing tree quality.
- A routing scheme using multiple trees for generating shorter paths.
- A method for creating multiple trees to minimise route length and spread link utilisation.

Our parallelisation scheme combines the best aspects of window based synchronisation and null message based synchronisation. The complexity of synchronisation is removed from the performance critical event handling code and managed directly by a *bridge* module representing a network link. In addition we use an offline load balancing method to ensure the highest possible utilisation of CPU time. In short, the simulator is characterised by:

- A high performance, memory efficient design.
- A parallelistion scheme tailored to network simulation.
- Simple, but effective load balancing.

This combination of lightweight parallel computing techniques and the abstraction method of AR has proved very successful in enabling large scale simulation.

## 5.3 Future Work

We have enhanced AR considerably, increasing its performance and improving the quality of routes generated. However, the range of its applicability could be extended further still by using it in conjunction with another routing protocol, or using it in a two layer hierarchy. For instance, as we have seen in Section 3.5, 55% of Internet nodes are in tree–structured subnetworks. Single tree routing could be used within these areas, while either full routing or multitree routing could be employed for routing between ASs. This would increase the fidelity of the routing while maintaining much of the efficiency of AR.

Although we have introduced a fixed cost algorithm for routing, a method using a Least Common Ancestor (LCA) [11] algorithm offers one useful advantage: if the LCA of two nodes can be calculated in fixed time, then the distance (in a tree) between two nodes can also be calculated in fixed time. This allows for an efficient choice of tree in multitree routing. Further, it would allow a packet to use several trees in one path — choosing the most efficient tree at each node, rather than making the choice once at the source node.

# Appendix A

# Hybrid Differential Traffic Modelling

The core work presented in this thesis falls into two categories: algorithmic routing and simulation parallelisation. However, in the course of developing these methods, other ideas were explored, and other avenues followed. This Appendix discusses some of the work that did not find its way into the main body of the thesis.

Our initial approach built on the analytical work of Garcia and Brun, et al. [16] [36] [15]. This theory termed differential traffic modelling, studies the transient and stationary states of network resources. Models for $M/M/1/\infty$, $M/M/1/N$, $M/D/1/N$, $M^k/M^k/1/\infty$ and $M/G/1/\infty$ queues have been developed. This analytic method, which has similarities to fluid simulation techniques, was combined with discrete event simulation to create a hybrid modelling method [37]. The hybrid approach also lends itself to parallelisation. In the remainder of this Appendix, we outline the principal results of the differential traffic theory and discuss its hybridisation and parallelisation. Finally, we note some of the issues that make its application to TCP/IP networks problematic.

## A.1 Differential Traffic Modelling

This Section introduces differential traffic modelling. We illustrate the theory of differential modelling by deriving the differential equations for $M/M/1/\infty$ and $M/M/1/N$ queues. The original papers [16] [36] [15] are more thorough and cover several other queue types.

### A.1.1 Transient Model of an $M/M/1/\infty$ Queue

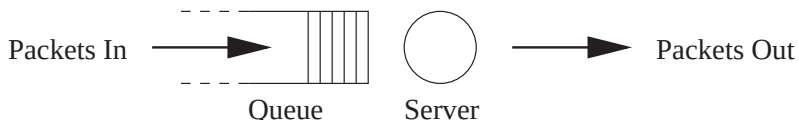



*Figure A.1: $M/M/1/\infty$ system. A single server queue with infinite buffer capacity. Packets arrive at the server. If the server is free the packet is processed otherwise it is queued until the server is free. Packet interarrival times follow a Poisson process. Packet service times are exponentially distributed.*

Among the simplest queueing systems is the $M/M/1/\infty$ queue, Figure A.1. It is an infinite capacity queue, in which packets arrive at a rate $\lambda$, according to a Poisson distribution, and are processed by the server. The service time is exponentially distributed with mean $\frac{1}{\mu}$. Packets are processed according to the FIFO principle.

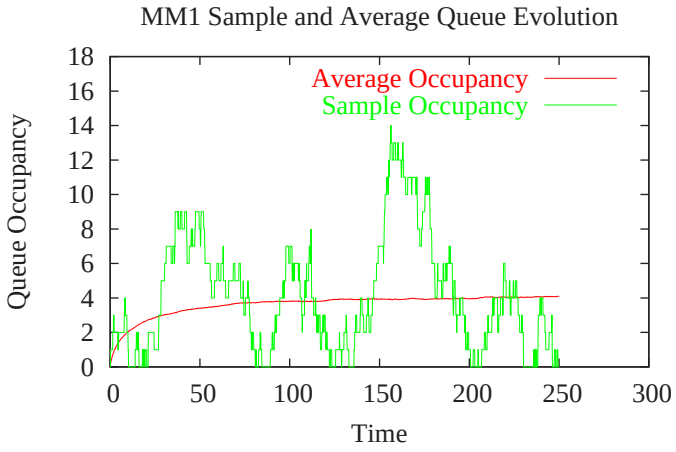



*Figure A.2: The evolution of a sample $M/M/1/\infty$ queue with $\rho = 0.8$, and the average of 10,000 such queues.*

Consider such a queue. Let $x(t)$ be the number of packets in the system at time $t$ (including any in the server). The instantaneous arrival rate of packets at the queue is $I(t)$, the instantaneous output rate from the server is $O(t)$. The aim of differential traffic modelling is to derive the average occupancy of the queue, $X(t) = E[x(t)]$. Now

$$\frac{dX(t)}{dt} = I(t) - O(t).$$

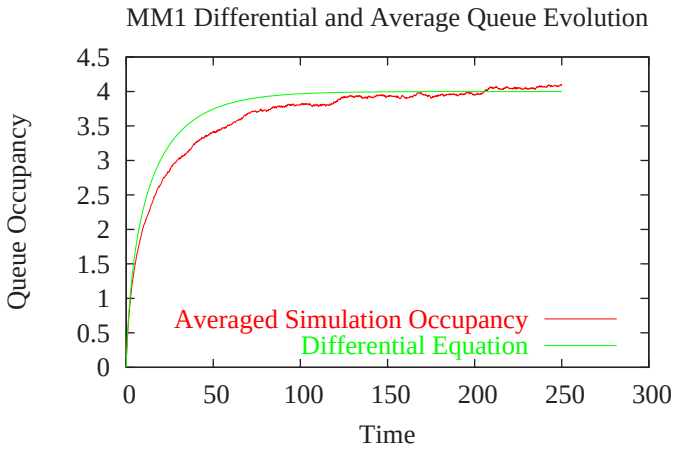



*Figure A.3: The averaged evolution of 10,000 $M/M/1/\infty$ systems and the evolution calculated using differential traffic modelling. There is good agreement in general, although for high $\rho$ the differential model approaches the stationary value more quickly.*

Now $x(t) \in \{0, 1, 2 \dots\}$. Let $P_i(t)$ be the probability that the system is in state $x(t) = i$. Then

$$\frac{dX(t)}{dt} = \frac{d}{dt}\sum_{i=0}^{\infty} iP_i t = \sum_{i=0}^{\infty} i\dot{P}(t) = \lambda - \mu(1 - P_0(t)).$$

Let the utilisation factor $\rho = \frac{\lambda}{\mu}$. If $\rho < 1$ then the steady state value of $X$ is given by

$$X(\infty) = \frac{\rho}{1-\rho} \quad \text{and} \quad P_i(\infty) = \rho^i(1-\rho).$$

(See, for example, [12]).

However, the expression for $P_0(t)$ is complex:

$$P_k(t) = e^{-(\lambda+\mu)t}\left[\rho^{\frac{k-i}{2}} I_{k-i}(at) + \rho^{\frac{k-i-1}{2}} I_{k+i+1}(at) + (1-\rho)\rho^k \sum_{j=k+i+2}^{\infty} \rho^{\frac{-j}{2}} I_j(at)\right]$$

where $a = 2\mu\sqrt{(}\rho)$ and $I_k(x) = \sum_{m=0}^{\infty} \frac{(\frac{x}{2})^{k+2m}}{(k+m)!m!}$, $k = -1 \dots \infty$ and $i$ is the initial number of packets, see [57] This expression is both analytically and numerically intractable.

The approach advocated by Garcia et al. [36], is to approximate $P_0(t)$. Note

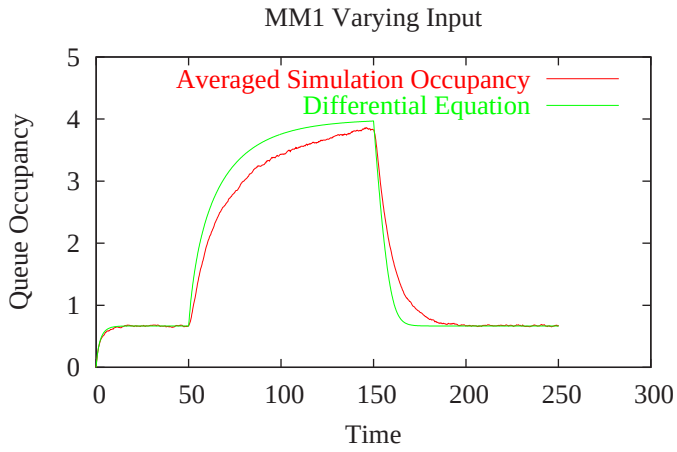



*Figure A.4: $M/M/1/\infty$ with varying input levels. From $t = 0$ to $t = 50$ the load $\rho = \frac{\lambda}{\mu} = 0.4$, at $t = 50$ the load is increased to $\rho = 0.8$, and at $t = 150$ it is reduced back to $\rho = 0.4$ again.*

that in the steady state

$$\rho = 1 - P_0(\infty) = \frac{X(\infty)}{1 + X(\infty)}.$$

It is asserted that this relation can be extended to the transient state of the queue:

$$1 - P_0(t) = \frac{X(t)}{1 + X(t)}.$$

The expression for $\dot{X}(t)$ then reduces to:

$$\dot{X}(t) \approx \lambda - \mu \frac{X(t)}{1 + X(t)}$$

It is easy to show that this differential equation converges to the stationary value $X(\infty) = \frac{\rho}{1-\rho}$. The trajectory of $X(t)$ can be easily computed numerically, using for example the Euler or Runge-Kutta methods. An advantage of the differential modelling technique is that the trajectory of $X(t)$ in the presence of varying $\rho$ can also be calculated.

Figure A.2 plots a single sample evolution of an $M/M/1/\infty$ queue, alongside the averaged evolution of many such queues. Figure A.3 plots the value of $X(t)$ using both the differential modelling method, and by direct simulation using the

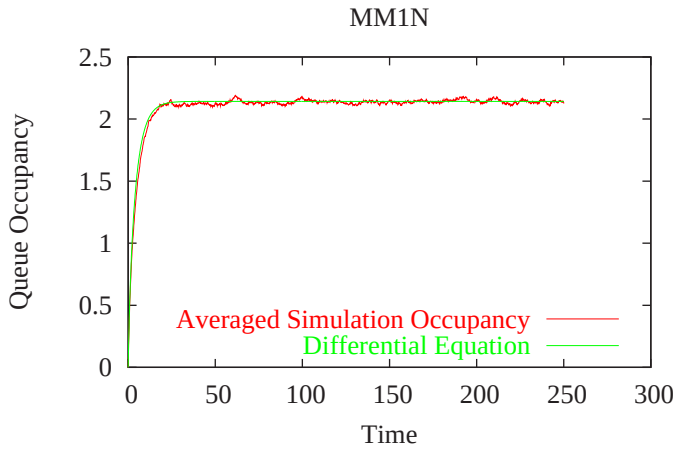



*Figure A.5: The averaged evolution of 10,000 $M/M/1/N$ systems and the evolution calculated using differential traffic modelling. The capacity of the system, $N = 6$.*

average of 10,000 runs. Figure A.4 displays a trajectory of $X(t)$ for a queue with varying $\lambda$.

It can be seen that there is close agreement between the differential and simulation trajectories. However the approximation error increases as $\lambda$ approaches $\mu$. This manifests as a faster convergence to the stationary value for the differential trajectory.

### A.1.2 Transient Model of an $M/M/1/N$ Queue

In most networks, queues for services have a finite capacity. The $M/M/1/N$ queue model describes a system with one server, one finite capacity buffer, Poisson distributed packet arrivals and exponentially distributed service times. The total capacity of the system is $N$ packets. Packets are processed according to the FIFO principle. Packets arriving at a full system are dropped.

The steady state probability distribution is:

$$P_k(\infty) = \rho^k P_0(\infty) = \begin{cases} \frac{1-\rho}{1-\rho^{N+1}}\rho^k & \text{if } \rho \neq 1 \\ \frac{1}{N+1} & \text{if } \rho = 1 \end{cases}$$

The corresponding queue occupancy is given by:

$$X(\infty) = \sum_{k=1}^{N} kP_k(\infty) = \begin{cases} \frac{N\rho^{N+2} - (N+1)\rho^{N+1} + \rho}{1-\rho-\rho^{N+1}+\rho^{N+2}} & \text{if } \rho \neq 1 \\ \frac{N}{2} & \text{if } \rho = 1 \end{cases}$$

The differential equation describing the transient average traffic for an $M/M/1/N$ queue is:

$$\dot{X}(t) = \lambda(1 - P_N(t)) - \mu(1 - P_0(t)).$$

Using the same approximation as the $M/M/1/\infty$ case we write

$$\dot{X}(t) \approx \lambda \left( \frac{1-\rho^N(t)}{1-\rho^{N+1}(t)} \right) - \mu\rho \left( \frac{1-\rho^N(t)}{1-\rho^{N+1}(t)} \right).$$

Now, $X(t)$ and $\rho(t)$ are related by

$$X(t) = \frac{N\rho(t)^{N+2} - (N+1)\rho(t)^{N+1} + \rho(t)}{1 - \rho(t) - \rho(t)^{N+1} + \rho(t)^{N+2}}$$

This cannot be directly solved for $\rho(t)$. However the roots of the equation can be easily found using numerical methods, for instance the Newton–Raphson method. Figure A.5 displays the trajectory of $X(t)$ in a finite capacity queue.

## A.2 Hybrid Model

While differential models have been developed for many queue types by Garcia et al., not all queues are amenable to analysis. A hybrid simulation model was introduced in response to this problem[37]. In the hybrid model, complex nodes are simulated using discrete event simulation, while the rest are modelled by differential equations. If a network consists of many simple nodes and few complex ones, then a hybrid model offers greater accuracy than a fully analytic model and better performance than a full simulation.

A node modelled by a differential equation takes $I(t)$ as an input. This is the expectation value of the packet input rate. Using this, and the expectation value of the occupancy $X(t)$, the values of $O(t+\Delta t)$ and $X(t+\Delta t)$ can be calculated, where $O(t)$ is the output rate. The arrival and departure processes have an associated distribution.

In contrast, a discrete event simulation deals with packets on an individual

basis. Events are inserted into a timeline and mark the arrival and departure of packets from the system.

In order to combine the two methods in one model, it is necessary to develop an interface that converts the output produced by one type of node to the input expected by the other. An analytic node expects the packet arrival time distribution. This can be generated by running a number of simulations (ensembles) of the complex node and performing a statistical analysis of the packets it outputs. Conversely, since the analytic node produces a distribution of output times for packets, this can be sampled and packets injected into the complex node at the corresponding times.

The input and output distributions are updated at intervals of $\Delta t$. Using a large value of $\Delta t$ has better performance, but less accuracy than using a small value. A small value can better catch the transient behaviour of a network. However, with a small $\Delta t$, more ensembles are needed in the simulated nodes to generate enough events for a good statistical analysis.

## A.3 Parallel Hybrid Model

The hybrid model presented above is an attractive target for parallelisation. One factor that creates difficulty in traditional parallel discrete event simulation is the volume of information that must be communicated between processors. This is due to the overhead of maintaining causality and the data associated with each packet that crosses interprocessor boundaries. In the hybrid model only statistical data need be communicated between different node types or between nodes on different processors. It is far more efficient to transmit the parameters of a distribution than to transmit each packet that the distribution describes.

## A.4 Weaknesses of Hybrid Differential Model

Differential traffic modelling is fast. A large network of $M/M/1/N$ nodes, with varying traffic levels, can be modelled easily. Examples of such networks are presented in [37]. However there are some circumstances in which it is difficult to apply differential or hybrid modelling effectively:

- Unusual or difficult to characterise traffic patterns.

- Highly multiplexed traffic.
- Networks with complex flow control protocols.

### A.4.1 Unusual Traffic Patterns

Differential modelling can describe queues with many types of service time distributions. These include exponential, constant, gamma and general distributions. However, it is not usually possible to model complex input distributions. As an example consider two nodes, $a$ and $b$, with constant service times. Node $a$ feeds node $b$. Node $a$ has Poisson distributed input traffic. However, if the input rate is sufficiently high, then the output distribution from $a$ has a constant packet interdeparture time. As there is no differential model for $D/D/1$ queues (constant interarrival and constant service times), node $b$ cannot be modelled by differential traffic analysis, and simulation must be used. Alternatively, the output from node $a$ can be approximated as Poisson, but this introduces errors.

In addition, complex, simulated nodes in a model are likely to create complex, non-Poisson output traffic. Thus, it easy to create a hybrid model with analytic nodes feeding simulated nodes but the reverse is more difficult.

### A.4.2 Highly Multiplexed Traffic

Consider a node with many inputs. Assume, for simplicity that each input has Poisson distributed interarrival times. This implies that the aggregate traffic is also Poisson, and so differential modelling can be used. However, if the input consists of very many low volume flows, the cost of the differential analysis may be as high as a full simulation. As long as the flow is active, the flow must be tracked. In a simulation, if a packet from a particular flow is not in the queue, no details about that flow need be stored.

### A.4.3 Complex Network Protocols

The Internet is governed by the TCP/IP suite of protocols. These protocols have feedback mechanisms built in, and traffic sources can interact in complex, unexpected ways. There has been some success in analysing the behaviour of TCP/IP flows, but typically for simplified situations [70] [21] [69]. In order to

create a realistic analytic model of a Internet like network, this analysis of TCP/IP flows would need to be extended and applied to differential traffic modelling.

## A.5 Conclusions

In this Appendix we have introduced the ideas of differential traffic modelling and its hybridisation with discrete event simulation. Although it holds great promise, it is not suitable for all types of network simulation. Due to our desire to model the Internet, detailed simulation of TCP/IP was essential, and hence highly efficient parallel simulation was pursued in place of hybridisation with differential modelling.

# Appendix B

# Module Definition API

## B.1 class

The *class* struct, Code Fragment 13, for a network device is initiated at program startup. Each *class* has its own initialisation function, usually named, for example *initXXXDevices*. This function is responsible for allocating and initialising a new *class* structure for this type of network equipment. The purpose of each entry, and an indication of where they are set is given below.

**name** This is a text string containing the name of the network device *class*. Examples are "router" or "link."

**printName** This function merely prints out a short description of the *class*.

**xmlDefineSubClass** This points to a function that takes as input a pointer to the *class*, and two pointers to XML configuration data provided to it by the XML parser. Its responsibility is to interpret the *class* specific data in the XML (using the libXML library) and to use it to build a *subClass* of the *class*. For instance, a generic *link device* does not have bandwidth or latency values. xmlDefineSubClass in this case would point to a function that would read latency and bandwidth values describing a particular type of network link, 10BaseT ethernet for example.

**initDevice** This points to a function that will create *device* structures that represent actual pieces of equipment in a network. A possible hierarchy is a generic link *class*, several *subClass*es (100BaseT ethernet, 64bps ISDN etc),

```
struct class {
  char *name;
  char *(*printName)(void);
  struct subClass *(*xmlDefineSubClass)
    (struct class *type, xmlDocPtr doc, xmlNodePtr params);
  int (*initDevice) (struct device *node,
    struct network *net, xmlNodePtr params, xmlDocPtr doc);
  int (*delSubClass) (struct subClass *subClass);
  int (*delReal) (struct device *d);
  int (*finaliseReal) (struct device *d);
  int (*firstEvent) (struct device *d,
    struct eventList *eList, int state);
  int (*postProcess)(struct device *d, struct network *net);
  int (*classPostProcess)(struct network *net,
    struct class *class);
  int (*acceptPacket)(struct device *d,
   struct inPoint *in, int size);
  int (*groupConnect)(struct device *d,
    char **realName, char **realInPoint, char **realOutPoint);
  int (*getDestination)(struct device *d,
    struct subClass *s);

  LIST_ENTRY(class) list;

  struct network *net;
  int parallel;
  int (*setParallelId)(struct device *d, int id);
  int (*getParallelId)(struct device *d);
  int n;
};
```

*Code Fragment 13: Class structure*

and actual links such as ethernet link "link001" and ISDN link "link002," for example.

**firstEvent** This points to a function that inserts any initial events for this *class* into the eventlist. For instance a traffic source amy have an event that tells it to start sending traffic at a certain time.

**delSubClass, finaliseReal, delReal** These point to functions responsible for tidying up memory use after the simulation is finished.

**acceptPacket** This points to a function specifying how a *device* of this type should handle an incoming packet.

**postProcess, classPostProcess** These point to functions that perform any initialisation that must take place after the network has been created. For instance, a AR router cannot create a routing tree until the network nodes have been connected and the network topology is available. The first function performs *device* only initialisation, the second performs initialisation common to all *devices* of the *class*.

**groupConnect** This points to a function that allows groups of nodes to be connected. It is only used by the mapreader module.

**getDestination** This points to a function that will select a *device* of the given *subclass* at random. It is commonly used by traffic source modules to select a destination for a traffic flow.

**list** This is used by the module loading code to keep track of all loaded modules (*class*s).

**net** A pointer to the global network structure

**parallel** A flag indicating whether or not this *class* performs interprocessor communication.

**setParallelId, getParallelId** These point to functions that set and get a globally unique id for a *device* that perform parallel communication.

```
struct subClass {
  char *name;
  struct class *class;
  LIST_ENTRY(subClass) subClasses;
  int numInPoints;
  int numOutPoints;
  int (*autoPoints)(struct device *d);
  int (*incrAutoPoints)(struct device *d);
  struct inPointTemplate *inProto;
  struct outPointTemplate *outProto;
  void *params;
  int storeGids;
  int *gids;
};
```

*Code Fragment 14: Sublass structure*

## B.2 subclass

A network *device*, such as a link or router, can come in many forms. These can often be grouped. For this reason, we use the concept of a *subclass* of a network device *class* to gather together properties shared between many instances of one *class*. For instance network links could be grouped into Ethernet links, IDSN links and so forth, each with its own bandwidth and latency parameters. While these could be stored in a structure representing a particular link instance, it is far more memory efficient to group common parameters in one place — the *subClass*, Code Fragment 14.

**name** A short name to identify the *subClass*.

**class** A pointer to the *class* of which this is a sub type.

**subClasses** Used by the module loading code to keep track of all the different *subclasses*.

**numInPoints, numOutPoints** The number of connections into and out of a *device* of this type.

**inProto, outProto** A pointer to a structures describing how to handle incoming and outgoing connections, common to all *devices* of this *subclass*.

**autoPoints, incrAutoPoints** Some modules, such as a traffic source have a fixed number of inputs and outputs. Others, such as routers may have varying numbers of inputs and outputs. These functions allow such modules to dynamically grow the number of input and output ports they provide on a per *device* level.

**params** A pointer that may be used by the module to store module specific data. For instance, a link module would define its own structure for specifying bandwidth and capacity and would use this pointer to access it.

**storeGids** A flag, set by the module loader, specifying whether to store all the global ids of *devices* of this type on all processors. Typically only used by traffic destinations.

**gids** A pointer to a list of all global ids of *devices* of this type (or void).

## B.3 device

The *device* structure, Code Fragment 15, represents actual instances of network objects. Structures of this type are connected together via *inpoint*s and *outpoint*s to replicate the network topology. This structure was kept as small as possible as one must be created for every network object.

```
struct device {
  char *name;
  int nid;
  struct subClass *subClass;
  void *data;
  struct inPoint *inPoints;
  struct outPoint *outPoints;
  struct device *group;
  short proc;
};
```

*Code Fragment 15: device structure*

**name** A short unique name for the *device*.

**nid** The local *device* id. The global id can be calculated from this.

**subClass** A pointer to the type of *subclass* this *device* is.

**data** A pointer to *device* specific data for this *device*. For instance a buffer *device* would store details of the packets stored in its buffer here.

**inPoints, outPoints** Pointers to structures describing the connections with the *device's* neighbours.

**group** A pointer to a *device* representing a group of *devices* (or void).

**proc** The processor responsible for this *device*.

## B.4 Example Module

In this section we present the code for a very simple module. This module represents a network device that does nothing but count the packets it receives. The code is shown in Code Fragments 16 to 30.

```
#include <device.h>
#include <data.h>
#include <lists.h>
#include <module.h>
#include <stdlib.h>
#include <stdio.h>
#include <counter.h>
#include <event.h>
#include <string.h>
#include <msgs.h>
#include <mpi.h>
```

*Code Fragment 16: Header files required in this module.*

```
struct counterDevice {
  int i;
};

struct counterData {
  int recv;
};
```

*Code Fragment 17: Structures used by the* counter class.

```
void initCounterDevices(struct classList *list) {
  struct class *class;


  class->printName = counterPrintName;
  class->firstEvent = counterFirstEvent;
  class->xmlDefineSubClass = counterXmlDefineSubClass;
  class->delSubClass = counterDelSubClass;
  class->delReal = delCounter;
  class->finaliseReal = NULL;
  class->type = makeString("CounterClass");
  class->initDevice = initCounter;
  class->insertEvent = counterInsertEvent;
  class->postProcess = NULL;
  class->classPostProcess = counterClassPostProcess;
  class->setParallelId = NULL;
  class->getParallelId = NULL;
  class->parallel = FALSE;
  return;
}
```

*Code Fragment 18: The* class *structure is initialised. This is called once from the module loading code.*

```
struct subClass *counterXmlDefineSubClass
  (struct class *type, xmlDocPtr doc, xmlNodePtr params) {
  struct subClass *subClass;
  struct counterDevice *a;

  subClass =
    (struct subClass *)malloc(sizeof(struct subClass));
  memset(subClass, 0, sizeof(struct subClass));

  subClass->class = type;

  a = (struct counterDevice *)malloc(sizeof(struct counterDevice));
  subClass->params = (void *)a;

  subClass->numInPoints = 1;
  subClass->numOutPoints = 1;

  subClass->outProto = (struct outPointTemplate *)
    malloc(sizeof(struct outPointTemplate) * subClass->numOutPoints);
  subClass->inProto =  (struct inPointTemplate *)
    malloc(sizeof(struct inPointTemplate) * subClass->numInPoints);

  {
    char *s;
    s=makeString("out");
    subClass->outProto->name = s;
    subClass->outProto->type = packetData;
    subClass->outProto->pointType = staticPoint;

    s=makeString("in");
    subClass->inProto->name = s;
    subClass->inProto->type = packetData;
    subClass->inProto->handler = counterAddPacket;
  }

  return subClass;
}
```

*Code Fragment 19: This code initialises a* subclass *of the* counter *type. However, the* class *is so simple, there is not much room for customisation. It is called once for each* subclass.

```
int initCounter(struct device *d,
  struct network *net, xmlNodePtr params, xmlDocPtr doc) {
  struct counterData *data;

  d->data = (struct counterData *)malloc(sizeof(struct counterData));
  data = (struct counterData *)d->data;
  data->recv = 0;
  return 0;
}
```

*Code Fragment 20: This initialises a* device *of the* counter class. *It is called once for each* device.

```
char *counterPrintName(void) {
  return makeString("Packet Counter");
}
```

*Code Fragment 21: Prints a description.*

```
int counterDelSubClass(struct subClass *subClass) {
  struct class *class;

  class  = subClass->class;
  free( subClass->name);
  free( subClass->params);

  return 0;
}
```

*Code Fragment 22: Delete the* subclass.

```
int counterAddPacket(struct device *d, struct inPoint *in,
    struct packet *p, double time, struct eventList *eList,
    int stateNum) {
  struct counterData *data;
  data = (struct counterData *)d->data;
  delPacket(p);
  data->recv++;
  return 0;
}
```

*Code Fragment 23: Delete the* class.

```
int delCounter(struct device *d) {
  struct counterData *data;
  data = (struct counterData *)d->data;
  printf("%s Received %d packets\n", d->name, data->recv);

  free(data);
  return 0;
}
```

*Code Fragment 24: Prints out the number of packets received and tidies up.*

```
int counterInsertEvent(struct insertEvent *e, struct network *net) {
  return 0;
}
```

*Code Fragment 25: Not needed for the* counter device.

```
int counterFirstEvent(struct device *d,
  struct eventList *eList, int stateNum) {

  return 0;
}
```

*Code Fragment 26: Not needed for the* counter device.

```
int counterSetParallelId(struct device *d, int id) {
  return 0;
}
```

*Code Fragment 27: Not needed for the* counter device.

```
int counterGetParallelId(struct device *d) {
  return -1;
}
```

*Code Fragment 28: Not needed for the* counter device.

```
int counterClassPostProcess(struct network *net,
  struct class *class) {
  return 0;
}
```

*Code Fragment 29: Not needed for the* counter device.

```
int counterPostProcess(struct device *d, struct network *net) {
  return 0;
}
```

*Code Fragment 30: Not needed for the* counter device.

# Bibliography


[1] Sumiyoshi Abe and Norikazu Suzuki. Gutenberg-Richter Law for Internetquakes. arXiv:cond-mat/0207302 11 Jul 2002 http://arxiv.org/pdf/cond-mat/0207302. 1

[2] Sumiyoshi Abe and Norikazu Suzuki. Omori's Law in the Internet Traffic. arXiv:cond-mat/0206453 24 Jun 2002 http://arxiv.org/PS_cache/cond-mat/pdf/0206/0206453.pdf. 1

[3] J. Ahn and P. Danzig. Packet network simulation: Speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking*, 4(5):743 – 757, Oct. 1996. 2.5

[4] William Aiello, Fan Chung, and Linyuan Lu. A random graph model for massive graphs. In *Proceedings of the 32nd Annual Symposium on Theory of Computing*, pages 171–180, 2000. 2.9, 3.5.1

[5] M. Allman, V. Paxson, and W. Stevens. RFC 2581: TCP congestion control, 1999. 2.1.1, 2.3

[6] Rajive L. Bagrodia and Wen-Toh Liao. Language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, 20(4):225 – 238, April 1994. 2.10

[7] Rajive L. Bagrodia, Richard Meyer, Mineo Takai, Yu an Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer October 1998*, 31(10):77 – 85, 1998. 2.10

[8] Sandeep Bajaj, Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, Padma Haldar, Mark Handley, Ahmed Helmy, John Heidemann, Polly Huang,

Satish Kumar, Steven McCanne, Reza Rejaie, Puneet Sharma, Kannan Varadhan, Ya Xu, Haobo Yu, and Daniel Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999. revised September 1999, to appear in IEEE Computer. 2.5, 2.10, 3.1

[9] Yair Bartal. Probabilistic approximations of metric spaces and its algorithmic applications. In *IEEE Symposium on Foundations of Computer Science*, pages 184–193, 1996. 3.3

[10] Yair Bartal. On approximating arbitrary metrics by tree metrics. In *Proc. of the 30th Ann. ACM Symp. on Theory of Computing*, pages 161–168, 1998. 3.3

[11] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Latin American Theoretical INformatics*, pages 88–94, 2000. 3.1.6, 5.3

[12] D. P. Bertsekas. *Data Networks*, chapter 3. Prentice-Hall, 2nd edition, 1991. 2.2, A.1.1

[13] A. Broido and K. Claffy. Internet topology: Connectivity of IP graphs. In *Proceedings of SPIE International Symposium on Convergence of IT and Communication*, Aug. 2001. 2.9, 3.5.1, 3.5.4

[14] Randy Brown. Calendar queues: A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220 – 1227, Oct 1988. 4.8.1

[15] O. Brun and J. M. Garcia. Analytical solution of finite capacity M/D/1 queues. *Journal of Applied Probability*, 37(4):1092 – 1098, Dec. 2000. 2.6, A, A.1

[16] O. Brun, J. M. Garica, and D. Gauchard. Transient analytical solution of M/D/1/N queues. *Journal of Applied Probability*, 39(4):853 – 864, 2002. 2.6, A, A.1

[17] R.E. Bryant. Simulation of packet communications architecture computer systems. Technical Report MIT-LCS-TR-188, Massachusetts Institute of Technology, 1977. 4.1

[18] Wentong Cai and S. J. Turner. An algorithm for distributed discrete-event simulation — the 'carrier-null message' approach. In *Proceedings of SCS Multi-Conference on Distributed Simulation*, pages 3—8, Jan. 1990. 4.1

[19] Kenneth L. Calvert, Matthew B. Doar, and Ellen W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, June 1997. 2.9, 3.2.3, 4.2

[20] Christopher D. Carothers and Richard Fujimoto. Background execution of time warp programs. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 12–19, May 1996. 2.10

[21] Claudio Casetti and Michela Meo. A new approach to model the stationary behavior of TCP connections. In *Proceedings of IEEE INFOCOM 2000*, pages 367–375, 2000. 1, 2.3, A.4.3

[22] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5:440 – 452, 1979. 4.1

[23] K. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, volume 21, pages 93—99, 1989. 4.1

[24] K. Claffy, Greg Miller, and Kevin Thompson. The nature of the beast: Recent traffic measurements from an internet backbone, 1998. http://www.isoc.org/inet98/proceedings/6g/6g_3.htm. 2.5

[25] Harold S. Connamacher and Andrzej Proskurowski. The complexity of minimizing certain cost metrics for k-source spanning trees. To appear in Discrete Applied Mathematics. 3.3

[26] James Cowie, Hongbo Liu, Jason Liu, David Nicol, and Andy Ogielski. Towards realistic million-node internet simulations. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Jun. 1999. 2.4, 2.10

[27] James Cowie, David Nicol, and Andy Ogielski. Modeling the global internet. *Computing in Science and Engineering*, 1(1):42 – 50, 1999. 2.10, 4.2

[28] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematic*, 1:269–271, 1959. 3

[29] M. Doar. A better model for generating test networks. In *Proceedings of Globecom '96*, Nov. 1996. 2.9, 3.5.1

[30] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM '99*, pages 251–262, 1999. 2.9, 3.5.1, 3.5.4, 4.7.2

[31] Arthur M. Farley, Paraskevi Fragopoulou, David W. Krumme, Andrzej Proskurowski, and Dana Richards. Multi-source spanning tree problems. *Journal of Interconnection Networks*, 1(1):61–71, 2000. 3.3

[32] R. Felderman and L. Kleinrock. An upper bound on the improvement of asynchronous versus synchronous distributed processing. *SCS Simulation Series*, 22:131 – 136, Jan. 1990. 4.1

[33] Free Software Foundation. GNU MP, 2002. http://www.swox.com/gmp. 3.2.2

[34] V. Frost, W. Larue, and K. Shanmugan. Efficient techniques for the simulation of computer communications networks. *IEEE Journal on Selected Areas in Communications*, 6(1):146 – 157, Jan. 1988. 2.7

[35] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33:30 – 53, October 1990. 2.4

[36] J. M. Garcia. A new approach for analytical modelling of packet switched telecommunication networks. Technical Report 98443, LAAS, 1998. 2.6, A, A.1, A.1.1

[37] J. M. Garcia, D. Gauchard, O. Brun, P. Bacquet, J. Sexton, and E. Lawless. Modélisation différentielle du trafic et simulation hybride distribuée. *Réseaux et Systèmes Répartis*, 13(6):635 – 664, 2001. A, A.2, A.4

[38] Michael R. Garey and David S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness*, page 207. Freeman, 1st edition, 1979. 3.3

[39] P. Glasserman, P. Heidelberger, P. Shahabuddin, and T. Zajic. Multilevel splitting for estimating rare event probabilities. *Operations Research*, 47(4):585 – 600, 1999. 2.8

[40] Ramesh Govindan and Hongsuda Tangmunarunkit. Heuristics for internet map discovery. In *Proceedings of IEEE INFOCOM 2000*, pages 1371–1380, Tel Aviv, Israel, March 2000. IEEE. 3.4, 3.5.3

[41] Yang Guo, Weibo Gong, and Don Towsley. Time-stepped hybrid simulation(TSHS) for large scale networks. In *Proceedings of IEEE INFOCOM'2000*, pages 441–450, 2000. 2.6

[42] Poul Heegaard. Speedup simulation techniques (survey). Workshop tutorial on Rare Event Simulation, 28-29 Aug 1997, Aachen, Germany, 1997. 2.8

[43] Philip Heidelberger. Fast simulation of rare events in queueing and reliability models. *ACM Transactions on Modeling and Computer Simulation*, 5(1):43 – 85, January 1995. 2.8

[44] Peter Hoare and Richard Fujimoto. HLA RTI performance in high speed LAN environments. In *Fall Simulation Interoperability Workshop*, Sept. 1998. 2.10

[45] T. C. Hu. Optimum communication spanning trees. *SIAM Journal of Computing*, 3(3):188 – 195, 1974. 3.3, 3.5.1

[46] Polly Huang. *Enabling Large-scale Network Simulations: A Selective Abstraction Approach.* PhD thesis, University Of Southern California, 1999. 1.2.1, 2.4, 2.5, 2.7, 2.10, 3, 3.1

[47] Polly Huang and John Heidemann. Minimizing routing state for light-weight network simulation. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, page to appear, Cincinnati, Ohio, USA, August 2001. IEEE. 1.2.1, 3.1, 3.1.6, 3.1.6, 3.4

[48] Polly Huang and John Heidemann. Minimizing routing state for light-weight network simulation. Technical Report ETH TIK-Nr. 106, ETH Zürich, 2001. 3.1.4

[49] Maria Hybinette and Richard Fujimoto. Cloning: A novel method for interactive parallel simulation. In *Proceedings of the 1997 Winter Simulation Conference*, pages 444–451, 1997. 2.8

[50] Information Sciences Institute. RFC 793, 1981. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc793.html. 2.1.1, 2.3, 4.2, 4.5.4

[51] Information Sciences Institute. The SCAN map, 1999. http://www.isi.edu/scan/mercator/maps.html. 3.1.5, 3.4, 3.6

[52] Van Jacobson and Micheal J. Karels. Congestion avoidance and control. In *Proceedings of SIGCOMM '88*, pages 314–329, 1988. 4.2, 4.5.4

[53] R. Jain and S. A. Routhier. Packet trains - measurement and a new model for computer network traffic. *IEEE Journal on Selected Areas in Communications*, 4(6):986 – 995, 1986. 2.5, 2.6

[54] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29:300 – 311, April 1986. 4.3, 4.3.2

[55] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48:96 – 129, 1998. 4.5.5

[56] G. Kesidis, A. Singh, D. Cheung, and W. Kwok. Feasibility of fluid event-driven simulation for ATM networks. In *IEEE Globecom 1996*, Nov. 1996. 2.6

[57] Leonard Kleinrock. *Queueing Systems Volume 1: Theory.* Wiley, 1975. A.1.1

[58] Leonard Kleinrock and Farouk Kamoun. Hierarchical routing for large networks. *Computer Networks*, 1:155 – 174, 1975. 1.2.1, 3

[59] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*, pages 118 — 120. Benjamin/Cummings, 1st edition, 1994. 2.4

[60] Krishnan Kumaran and Debasis Mitra. Performance and fluid simulations of a novel shared buffer management system. In *INFOCOM 1998*, pages 1449–1461, 1998. 2.6

[61] John Lewis and Raymond Russell. An introduction to large deviations for teletraffic engineers. ftp://www.stp.dias.ie/DAPG/LDtut96.ps, 1996. 2.8

[62] Y. Li and Y. Bouchebaba. A new genetic algorithm for the optimal communication spanning tree problem. In *Artificial Evolution*, volume 1829 of *Lecture Notes in Computer Science*, pages 162–173, 1999. 3.3

[63] Yi-Bing Lin. Parallel independent replicated simulation on a network of workstations. In *Proceedings of the eighth workshop on Parallel and distributed simulation*, pages 73–80. ACM Press, 1994. 2.4

[64] B Liu, Y Guo, J Kurose, D Towsley, and W Gong. Fluid simulation of large scale networks: Issues and tradeoffs. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume IV, pages 2136–2142, 1999. 2.6

[65] Xiaowen (Jason) Liu. Parallel simulation of large-scale wireless ad hoc networks. Research proposal for Doctoral Thesis. http://citeseer.nj.nec.com/liu01parallel.html, 2001. 2.4, 2.10

[66] B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989. 4, 4.1

[67] D. Magoni and J.-J. Pansiot. Analysis of the autonomous system network topology. *ACM SIGCOMM Computer Communication Review*, 31(3):26 – 37, July 2001. 2.9, 3.5.1, 3.5.4

[68] Alberto Medina, Ibrahim Matta, and John Byers. On the origin of power laws in internet topologies. *ACM Computer Communication Review*, Apr. 2000. 2.9

[69] Marco Mellia, Ion Stoica, and Hui Zhang. TCP model for short lived flows. *IEEE Communications Letters*, 6(2):85 – 88, February 2002. 1, 2.3, A.4.3

[70] V. Misra, W. Gong, and D. Towsley. Stochastic differential equation modeling and analysis of tcp windowsize behavior. Technical Report ECE-TR-CCS-99-10-01, Department of Electrical and Computer Engineering, University of Massachusetts, 1999. Presented at Performance 99, October Istanbul

1999. Available at ftp://gaia.cs.umass.edu/pub/Misra00_AQM.ps.gz. 1, 2.3, A.4.3

[71] David M. Nicol. Parallel discrete-event simulation of fcfs stochastic queueing networks. In *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 124–137. ACM Press, 1988. 4.1

[72] David M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM*, 40(2):304–333, 1993. 4, 4.1

[73] David M. Nicol and Richard M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, 53:249–285, 1994. 2.4

[74] David M. Nicol, Michael Goldsby, and Michael Johnson. Fluid-based simulation of communication networks using SSF. In *Proceedings of the 1999 European Simulation Symposium*, October 1999. 2.6

[75] Peter O'Reilly and Joseph Hammond. Efficient simulation technique for performance studies of CSMA/CD local networks. *IEEE Journal on Selected Areas in Communications*, 2(1):238 – 249, 1984. 2.7

[76] Vern Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM Transactions on Networking*, 5(5):601–615, 1997. 3.5.2

[77] Vern Paxson and Sally Floyd. Wide-area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226 – 224, 1995. 2.2, 2.3, 2.8

[78] S. Raman, S. McCanne, and S. Shenker. Asymptotic scaling behavior of global recovery in SRM. In *Proceedings of SIGMETRICS/PERFORMANCE 98, Joint International Conference on Measurement and Modeling of Computer Systems*, 1998. 3.1

[79] D. M. Rao, N. V. Thondugulam, R Radhakrishnan, and P. A. Wilsey. Unsynchronized parallel discrete event simulation. In *Winter Simulation Conference*, pages 1563–1570, December 1998. 2.10

[80] D. M. Rao and P. A. Wilsey. An ultra-large scale simulation framework. *Journal of Parallel and Distributed Computing*, 10(1):18 – 38, 2000. 2.10, 4.2

[81] J. Reynolds, J. Postel, and Information Sciences Institute. RFC 1700, 1994. Edited by Jon Postel. Available at http://rfc.sunsite.dk/rfc/rfc1700.html. 2.5

[82] George F. Riley and Mostafa Ammar. Simulating large networks: How big is big enough? In *Proceedings of First International Conference on Grand Challenges for Modeling and Simulation*, Jan. 2002. 2.4, 2.10

[83] George F. Riley, Mostafa. Ammar, R. Fujimoto, D. Xu, and K. Perumalla. Distributed network simulations using the dynamic simulation backplane. In *Proceedings of International Conference on Distributed Computing Systems 2001 (ICDCS'01)*, 2001. 2.10

[84] George F. Riley, Mostafa H. Ammar, and Richard Fujimoto. Stateless routing in network simulations. In *MASCOTS*, pages 524–531, 2000. 2.5, 2.10

[85] George F. Riley, Richard Fujimoto, and Mostafa H. Ammar. A generic framework for parallelization of network simulations. In *MASCOTS*, pages 128–, 1999. 2.4, 2.10, 4.2

[86] George F. Riley, Ellen Zegura, and Mostafa Ammar. Efficient routing using Nix-Vectors. Technical Report GIT-CC-00-27, Georgia Tech, 2000. 2.5

[87] Franz Rothlauf, Juergen Gerstacker, and Armin Heinz. On the optimal communication spanning tree problem, 2003. Working paper: http://www.bwl.uni-mannheim.de/Heinzl/publications/working_paper_2003_10.pdf. 3.3

[88] Stefan Savage, Andy Collins, Eric Hoffman, John Snell, and Thomas E. Anderson. The end-to-end effects of internet path selection. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 289–299, Oct. 1999. 3.5.3

[89] D. Schwetman. Hybrid simulation models of computer systems. *Communications of the ACM*, 21(9):718–723, September 1978. 2.7

[90] D. D. Sleator and R. E. Tarjan. Self adjusting binary trees. In *Proceedings of the fifteenth annual ACM symposium on Theory of Computing*, pages 235 – 245, 1983. 4.3, 4.3.2

[91] Hongsuda Tangmunarunkit, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Network topology generators: degree-based vs. structural. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 147–159. ACM Press, 2002. 2.9, 3.5.1, 3.6, 4.7.2

[92] Hongsuda Tangmunarunkit, Ramesh Govindan, and Scott Shenker. Internet path inflation due to policy routing. In *Proceeding of SPIE ITCom 2001, Denver 19-24 August 2001*, pages 188–195, Aug. 2001. 3.5.3

[93] Hongsuda Tangmunarunkit, Ramesh Govindan, Scott Shenker, and Deborah Estrin. The impact of routing policy on internet paths. In *Proceedings of IEEE INFOCOM*, pages 736–742, 2001. 3.5.3, 3.6.1, 3.6

[94] U. Vishkin and B. Schieber. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, pages 17:1253–1262, 1988. 3.1.6

[95] Jared Winick and Sugih Jamin. Inet-3.0: Internet topology generator. Technical Report UM-CSE-TR-456-02, EECS, University of Michigan, 2002. 2.9, 3.5.1, 3.6, 4.2, 4.5.5

[96] B. Y. Wu, G. Lancia, V. Bafna, K. M. Chao, R. Ravi, and C. Y. Tang. A polynomial time approximation scheme for minimum routing cost spanning trees. *SIAM Journal on Computing*, 29(3):761–778, 2000. 3.3

[97] A. Yan and W. Gong. Fluid simulation for high speed networks. Technical Report TR-96-CCS-1, Dept. of ECE, Univ. of Massachusetts, Dec. 1999. 2.6

[98] E. Zegura, K. Calvert, and M. Donahoo. A quantitative comparison of graph-based models for internet topology. *IEEE/ACM Transactions on Networking*, 5(6), Dec. 1997. 2.9, 3.5.1

[99] Bernard P. Zeigler. *Theory of modelling and simulation*, pages 141 — 142. Wiley-Interscience, 1st edition, 1976. 2.2.1

[100] Xiang Zeng, Rajive Bagrodia, and Mario Gerla. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998. 2.10