Distributed Simulation of Peer-to-Peer Networks

Andrew Wilkins

This report is submitted as partial fulfilment of the requirements for the Honours Programme of the School of Computer Science and Software Engineering, The University of Western Australia, 2005
Abstract

Simulation is a valuable tool for testing ideas in a controllable environment. Many application domains have few other alternatives for testing. For example, peer-to-peer file-sharing attracts millions of users today, and orchestrating any useful analysis on such a huge, distributed network is a daunting, if not infeasible task. Simulation allows us to control every aspect of the system to more easily analyse the behaviour in which we are interested.

However, simulation is not a perfect solution. To make simulation a feasible solution, we must abstract away a system’s characteristics that have lesser or no impact on its behaviour – without doing this, simulations tend to require far too many resources and too much time to be useful. This is exacerbated in large systems, such as an aforementioned peer-to-peer network.

Distributed and parallel simulation can solve problems with resource requirements, but also create restrictions on the application domain, namely the transparency of mechanisms to the application. In this dissertation we will examine two distributed simulation schemes, using peer-to-peer network routing as a case-study. We have developed a distributed overlay network simulation framework called dnet, to demonstrate our approach to semi-transparent distributed simulation. We will discuss the challenges involved, and explain why distributed simulation is mismatched with our approach to peer-to-peer network routing simulation.

Keywords: Distributed discrete event simulation, peer-to-peer
CR Categories: I.6.8 Types of Simulation - Distributed; C.2.1 Network Architectures - Network topology.
Acknowledgements

Many thanks, to

my supervisor, Dr. Chris McDonald, for lending an ear when my project was ailing, and for keeping me from becoming disillusioned;

my family, for providing a roof, food, encouragement and love for 21 years;

Shannon, Aaron and James, for their years of friendship, and ensuring that I maintain a healthy diet of beer and late nights; and

my girlfriend, Michelle; without your comforting words and consistent encouragement I surely would have gone mad this year. Or is that madder?

~

“I get by with a little help from my friends” – The Beatles
## Contents

Abstract ....................................................... ii

Acknowledgements ........................................... iii

1 Introduction .............................................. 1

2 Literature Review ........................................ 3
   2.1 Overview .............................................. 3
   2.2 Distributed Discrete Event Simulation .............. 3
      2.2.1 Conservative Distributed Simulation ........... 5
      2.2.2 Optimistic Distributed Simulation ............. 6
      2.2.3 Global Virtual Time .............................. 8
   2.3 Summary ............................................... 9

3 Motivating Example: Peer-to-Peer Networks .......... 10
   3.1 Background ........................................... 10
   3.2 Simulation of Peer-to-peer Networks ............... 12
      3.2.1 Point-to-point Packet-level Simulation ....... 12
      3.2.2 Peer-behaviour and Evaluation Metrics ... 13
      3.2.3 High-level Modeling .......................... 13
   3.3 Summary ............................................... 14

4 Methodology .............................................. 15
   4.1 General Approach .................................... 15
   4.2 dnet: distributed overlay network simulator .... 16
      4.2.1 General Architecture and Terminology ..... 16
      4.2.2 Sequential Implementation .................. 19
      4.2.3 Chandy-Misra Implementation ............... 20
List of Figures

2.1 Causal Ordering in Distributed Simulations .................... 4
2.2 Chandy Misra Null-Message Driven Deadlock Avoidance ........ 7
3.1 A Simple Overlay Network ..................................... 11
4.1 General Architecture of dnet .................................... 17
4.2 Distributed Infrastructure ....................................... 19
4.3 Generating a Topology in dnet .................................. 31
5.1 Calendar Queue: Past-event Insertion ............................ 35
5.2 Speed comparison of Linear and Calendar Queues ............... 37
5.3 A Ring-Topology Overlay Network ................................ 38
5.4 Speedup for 100 Nodes ........................................... 40
5.5 Speedup for 500 Nodes ........................................... 40
5.6 Speedup for 1000 Nodes .......................................... 41
5.7 Speedup for 5000 Nodes .......................................... 41
5.8 Speedup for 10000 Nodes ......................................... 42
Listings

4.1 Example of DNET_on_undo usage ........................................ 27
5.1 Speedup Experiments: GT-ITM specification file ..................... 39
List of Algorithms

1. Sequential Simulation ........................................ 19
2. Entry-point to Chandy-Misra Algorithm ....................... 22
3. Chandy-Misra Derivative Algorithm ............................ 23
4. Entry-point to Time Warp Algorithm .......................... 24
5. Time Warp: process_event(event) ............................. 24
6. Time Warp: receive_event(msg_id) ............................ 25
7. Time Warp: rollback(to) ........................................ 26
8. Queue Hold-time Performance ................................... 36
CHAPTER 1

Introduction

Overlay networks – particularly peer-to-peer networks – have become increasingly popular in recent years, with file-sharing networks becoming so accessible to the masses. Like so many technologies, overlay networks typically suffer from scalability issues. For example, the popular peer-to-peer network routing protocol Gnutella reached a scalability limit with its initial network-flooding protocol, which was subsequently eliminated by a protocol revision. Scalability is a common issue in many distributed computing systems, and is often difficult to solve.

How can we verify claims of scalability in huge, dynamic and distributed systems? Do we ask hundreds of thousands of users to upgrade their software and participate in controlled tests? There are obvious issues of practicality here. To overcome them somewhat, we are able to employ the use of simulation. By simulating a system, one is able to control every aspect of its behaviour. This removes the impracticality of coordinating tests in the physical system. Furthermore, it opens up the option of analysing events which only rarely occur in the physical system, but have a significant affect on its stability. Finally, there are things which simply can’t (or shouldn’t) be tested in real systems, such as security – most people will object to their computer’s security being compromised, even in the name of research.

Whilst simulation is adequate for a very large number of applications, it is by no means a perfect solution. By its very definition, simulation requires that we abstract away certain characteristics of the physical system deemed to be insignificant to our interests. For all but the most trivial of systems, this is not an easy task. Frequently we will want to perform simulations which abstract away as little of the system as possible, whilst keeping resource-requirements at a practical level.

The most widely adopted way of reducing the resource-requirements – especially processing time – to perform a computation, is to distribute the simulation over a number of computers. This approach is known as distributed simulation, and was introduced independently by Chandy [7] and Bryant [5] in 1977.
In this dissertation we will study distributed simulation in the context of peer-to-peer network simulation. We will discuss our approach to modeling an overlay (peer-to-peer) network, and our work implementing our distributed overlay network simulator, \textit{dnet}. We developed \textit{dnet} to be transparent to the application level, and be able to scale to simulations of tens to hundreds of thousands of nodes, using end-to-end, rather than point-to-point, packet-level simulation. We will show that distributed simulation is highly dependent on the application domain and the level of abstraction of our model, and subsequently show how distributed simulation is not useful for simulating overlay networks with an end-to-end model.

The rest of the dissertation is structured as follows. In Chapter 2 we will review prior work on distributed simulation. In Chapter 3 we will present peer-to-peer network simulation as a motivating application for distributed simulation; we will cover fundamental background information, as well as present a review of prior work on peer-to-peer network simulation. In Chapter 4 we will discuss our approach to the simulation of peer-to-peer networks, and introduce our distributed overlay network simulator, \textit{dnet}. In Chapter 5 we will discuss the experiments we performed, and results obtained, to gauge the effectiveness of our simulator, and our approach in general. Finally, we will summarise our findings and offer ideas for future work in Chapter 6.

Throughout this dissertation we make references to the source-code of \textit{dnet}. Information on acquiring and building \textit{dnet} is available in Appendix B.
CHAPTER 2

Literature Review

2.1 Overview

In this chapter we will review prior work related to distributed discrete event simulation. We will focus on simulation scalability, and the transparency of mechanisms to the application level.

2.2 Distributed Discrete Event Simulation

Discrete event simulation of a system is a simulation wherein the events simulated are discretised in time. Discrete event (distributed or sequential) simulations are typically undertaken in one of two ways: event-driven, or time-driven. Time-driven simulations advance in single units of time, and execute all events having a time-stamp equal to the current simulated time. Event-driven simulations simply advance to the queued event with the lowest time. Time-driven simulation does not lend itself well to distribution or parallelisation, due to the fact that there must be some centralised facilitator to advance time. As a result, we will only be reviewing event-driven schemes.

Discrete event-driven simulation works quite simply. We have a start-time, a set of initial events, and completion criteria. The process is begun by executing events in order of scheduled time – processing events may add more events to the set of events. Once the completion criteria are met, the simulation finishes, and no more events are executed.

Distributed simulation is the simulation of a system in a distributed computing environment. Chandy [7] and Bryant [5] both independently defined this scheme of simulation in 1977. In distributed simulation, the system being modelled (the physical system) is made up of components, or physical processes. The simulation is composed of logical processes, with a one-to-one mapping to the physical processes. Each logical process executes events independently of oth-
Figure 2.1: Let A and B be two logical processes in our simulation, with the respective current simulated times of \( t_4 \) and \( t_1 \) (where \( \forall m, n \in N, m \leq n \iff t_m \preceq t_n \) (\( t_n \) does not precede \( t_m \))); dashed lines represent timelines for each process, arrows between them represent messages, and solid circles represent events. In this scenario, the event at \( t_1 \) in B generated an event in B at \( t_2 \), which in turn generated an event in A at \( t_3 \). If synchronisation were not accounted for, A would have immediately executed the event at \( t_4 \), and received the event at \( t_3 \) out of order.

Events; the problem here is how to ensure events are executed in the correct order. Events in a simulation must be causally ordered. Causal ordering is the ordering of events based on their dependence; given arbitrary events \( e_1 \) and \( e_2 \), where \( e_1 \) has the ability to cancel \( e_2 \), then \( e_1 \) must be executed prior to \( e_2 \). In many simulations the causal ordering is equivalent to \( \leq \text{time} \), which orders events by their time values. It is possible, however, for a causal ordering to have two events \( e_2 \) and \( e_1 \) where \( e_2 > e_1 \), but have \( e_2 \) independent of \( e_1 \). For the sake of simplicity, we will assume throughout the rest of the dissertation that events are ordered according to \( \leq \text{time} \). Figure 2.1 illustrates causal ordering in a distributed simulation.
To maintain correct time ordering (that is, to prevent causality errors – the processing of an event prior to one that it is dependent on), there exist two general classes of algorithms: conservative, and optimistic. Conservative algorithms ensure at all times that no events are performed out of order. Optimistic algorithms allow events to be processed out of order, but provide a correction mechanism known as rollback, which is executed when disorder is detected.

In the following subsections, we review some of the work done on the synchronisation problem – that of ensuring proper event ordering – and distributed simulation in general. Section 2.2.1 covers conservative distributed simulation, and Section 2.2.2 covers optimistic simulation. Finally, in Section 2.2.3, we will look at the general concept of Global Virtual Time, and some of the work based on it.

2.2.1 Conservative Distributed Simulation

The simplest, original approach to ensuring correct execution order is to disallow execution of an event unless it is known that no other event will occur at an earlier time. Generally this is non-trivial, as nodes in the distributed simulation operate independently of one another. Each simulator may be at different points in simulated time, unbeknownst to the other nodes.

Chandy and Misra [9] specify the scheme of asynchronous distributed simulation, and a means of detecting and recovering from deadlock. The scheme they propose runs in two phases: the parallel phase, and the phase interface. Respectively, these phases refer to the simulation of events until deadlock occurs, and the advancement of clock values. The latter phase resolves deadlock, and the system returns to simulating events.

Chandy and Misra formally prove the properties of their proposed scheme, and offer various examples of queuing networks to highlight the usefulness of the algorithm. Unfortunately, their work is highly theoretical, and offers no empirical data. Experimentation is specified as necessary in the future (which has been performed extensively since 1981).

To perform reasonably efficiently, conservative distributed simulations need to make use of lookahead [12]. Lookahead is the largest amount of time a logical process may jump forward without affecting causality constraints. Fujimoto [12] sees this as having a negative impact on transparency to applications, since they would typically need to tell the simulator the lookahead values. This is generalised, however; it is possible to base lookahead values on the lower-bounds of virtual propagation delays between logical processes.
Conservative distributed simulation schemes work best when logical processes are connected in a tandem (serial, acyclic) network topology \cite{25}. Deadlock occurs when there exists a cycle in the logical process topology. Consider a simple topology with only two logical processes, \(LP_1\) and \(LP_2\), and suppose each logical process wants to transmit a message to the other. Neither logical process may transmit their messages without knowing that the other is able to accept it, and so they deadlock. Under the Chandy-Misra deadlock-avoidance scheme, each would send the other a null-message with its current time incremented by the lookahead. This continues until each logical process knows the other’s local time to be greater than the receive-time of the message they want to transmit (Figure 2.2). Now, consider the same scenario with a tandem topology, where \(LP_1\) wants to transmit a message to \(LP_2\), but \(LP_2\) simply consumes and does not transmit anything. Since \(LP_1\) has no incoming channels, it may advance its clock without regard to any other logical processes; \(LP_1\) immediately sends its message to \(LP_2\).

2.2.2 Optimistic Distributed Simulation

Another, more complicated way to perform distributed simulation is called the optimistic approach. Jefferson \cite{17} proposes a theoretical mechanism for distributed synchronisation called virtual time, and an implementation of it called Time Warp. The general idea of virtual time and Time Warp is that events may occur out of order. When a process is observed to have executed an event out of order, that process is rolled back to a previous state, and special anti-messages are sent out to stop the propagation of the resulting incorrect events.

Jefferson informally proves the correctness of Time Warp, and offers conjectures on performance benefits over conservative methods algorithms proposed by Chandy and Misra. He covers two mechanisms in Time Warp: local control, and global control. Local control is necessary for all simulations using Time Warp; it entails the behaviour of processes with regard to rollback, and sending and receiving messages. Global control governs more general issues, such as fossil collection (memory reclamation), flow-control, I/O, error-handling and distributed snapshots. These latter issues are not common to all simulations, and so his coverage is very general.

Jefferson’s coverage of state restoration is limited to cases where the state of a simulation is static in size. Firstly, Time Warp is supposedly transparent from the application level. This holds true if users use only statically allocated memory, which can be saved and restored without help from the application. Dynamic memory on the other hand, may require the application to call special save and restore functions: this is not transparent. Secondly, as overhead in
Figure 2.2: A basic scenario illustrating how null-messages and lookahead are used to avoid deadlock in a cyclic simulation topology.

<table>
<thead>
<tr>
<th>Real Time</th>
<th>Simulated Time Begins at Zero in Both A and B. We Assume A and B Both Have a Lookahead of 1.</th>
</tr>
</thead>
<tbody>
<tr>
<td>A wants to send a message to B at time $t_2$</td>
<td></td>
</tr>
<tr>
<td>A doesn't know whether or not it will receive any events at $t_1$, and so can not yet send the message. Instead, A sends a null-message to B, stating it will not execute any events until at least $t_1$ (current time + lookahead).</td>
<td></td>
</tr>
<tr>
<td>B wants to send a message to A at time $t_2$</td>
<td></td>
</tr>
<tr>
<td>As above.</td>
<td></td>
</tr>
<tr>
<td>B receives null-message from A</td>
<td></td>
</tr>
<tr>
<td>B now knows that it will not receive any messages until at least $t_2$, which can not affect other events at $t_2$. B sends its message to A.</td>
<td></td>
</tr>
<tr>
<td>A receives null-message from B</td>
<td></td>
</tr>
<tr>
<td>As above.</td>
<td></td>
</tr>
<tr>
<td>A receives message from B</td>
<td></td>
</tr>
<tr>
<td>B receives message from A</td>
<td></td>
</tr>
</tbody>
</table>

$t_0$ $t_1$ $t_2$
state saving and restoration increases, the performance of optimistic algorithms
degrad. Dealing with dynamically allocated memory would certainly increase
the overhead of state saving and restoration.

2.2.3 Global Virtual Time

In a distributed simulation, each simulation process will have its own *local virtual
time*. It is desirable that logical processes are not synchronised at every moment
in time, so that they can operate in parallel. When events are processed, and
time advances, the logical processes must save their state to memory, so that if
an event is received out of order the process may rollback to that time and undo
all of the intervening events. This state saving accumulates over time, becoming
more problematic as state sizes increase, and eventually the simulator’s memory
may be exhausted.

To alleviate the problem of memory exhaustion, one needs to occasionally
synchronise each logical process’ local virtual clock. By doing this, the simulators
can ascertain the earliest time past which the simulation will not roll back; this
is known as the *Global Virtual Time* (GVT) [17]. Using GVT, one can perform
*fossil collection* (memory reclamation), by purging from memory all saved states
prior to GVT (except for one, since we may still roll back to GVT, just not
before it). Other uses of GVT are for inter-simulator message flow-control, error
handling, distributed snapshots, and committing irrevocable actions such as I/O.

There exist a number of algorithms for determining GVT, but they are typ-
ically targeted at specific types of simulation setups. For example, earlier algo-
rithms required reliable, FIFO communication between simulations, such as those
based on the work of Chandy and Lamport [8]. Mattern discusses a number of
algorithms for use in distributed simulators that use both FIFO and unordered,
but still reliable, communication [21].

Tomlinson and Garg present an algorithm for use on simulators with un-
ordered reliable communication [36]. Their algorithm is different to most others,
in that it detects when GVT has reached a specified point, rather than performing
an estimation. They claim their algorithm is well suited to interactive simul-
ations, since it is “minimally latent” – that is, there is minimal latency between
the request for the GVT detection, and the occurrence of the detection. One
problem, though, is that it must be used carefully; a process may have its local
time set to infinity, if its event queue is empty. If this process requests GVT
detection based on its local time (as suggested in the original paper), then it
would potentially request detection of GVT at time infinity, which is no better
than having no GVT detection at all.
There exist other well-used GVT estimation algorithms which are more restricted in their applicability. One is pGVT [11], which was designed specifically for simulators with unreliable, non-FIFO communication. Such a communication medium is not necessarily undesirable for Time Warp simulations, since it may become unnecessary to continue the sending of a message due to a rollback occurring. Fujimoto and Hybinette [13] defined what is the most popular GVT estimation algorithm for use in shared-memory Time Warp simulators. This has the obvious drawback that it is not useable in message-passing simulators. More recently, an algorithm called Seven-O’Clock was proposed by Bauer et al [3], which is based on the aforementioned algorithm by Fujimoto and Hybinette, but uses “network atomic operations” to extend the algorithm for use in a network-based simulator.

2.3 Summary

Distributed simulation techniques fall into several broad categories, but it is generally not possible to say one is better than another. The usefulness of the techniques is highly application-dependent. Conservative algorithms excel when we can determine look-ahead, otherwise they suffer due to their deadlocking. Optimistic algorithms exploit look-ahead transparently, but suffer as state saving and recovery costs increase. GVT detection may be used to reduce the cost of state saving and recovery, by periodically reclaiming memory from states that will not be rolled back to.
CHAPTER 3

Motivating Example: Peer-to-Peer Networks

In this chapter we will introduce the reader to peer-to-peer network concepts, and prior work on simulation of peer-to-peer networks. In Section 3.1 we will provide general background information on peer-to-peer networks, and in Section 3.2 a review of literature on simulating peer-to-peer networks.

3.1 Background

Peer-to-peer networks are a sort of overlay network; an overlay network is a logical network composed of a subset of an underlying network – typically the Internet (Figure 3.1).

In a peer-to-peer network, nodes undertake the role of both client and server. This is in contrast with the client-server architecture, where client and server roles are separate. Peer-to-peer networks come in three general architectures:

**Pure** Pure peer-to-peer networks are those that have absolutely no centralised facilities. An example of such a peer-to-peer architecture is the original Gnutella network [15], in which all peers acted as both a server and a client.

**Hybrid** Hybrid peer-to-peer networks incorporate some centralisation into their architecture. For example, Napster [27] uses a centralised server for searching for files. Once a file is located, peers connect to each other directly to perform the transfer.

**Partially Centralised** Kazaa [35] (which uses the FastTrack protocol [38]) is an example of a peer-to-peer network, which uses some centralisation to improve scalability by having peers with more processing power and bandwidth act as a proxy for slower peers to the network, handling some messages on their behalf.
Figure 3.1: The computers of Alice, Bob, Charlie, Dianne, Eliza and Frank are all connected to the Internet via their respective Internet Service Providers. Alice, Bob, Charlie and Dianne have formed an overlay network by connecting to each other via TCP/IP. Eliza and Frank are not part of this overlay network at the moment, but it is possible for them to join at a later point in time by connecting to one of the others.

There are various benefits for using each architecture, perhaps the most important being scalability and resilience to failure. Pure peer-to-peer networks, such as Gnutella [15], tend not to be scalable [32]. On the other hand, because pure peer-to-peer systems are completely distributed, it is more difficult to perform an attack that would disrupt the system. In contrast, hybrid peer-to-peer networks (as well as client-server architectures) are more susceptible to denial-of-service attacks, since attackers can focus on the centralised infrastructure.
3.2 Simulation of Peer-to-peer Networks

When simulating a system it is important to have an accurate model, so simulations are representative of the behaviour of the actual system. Models will cover a broad range of criteria, including peer (user) and protocol behaviour. In recent years there has been a substantial amount of research into defining models of peer-to-peer systems. In this section we review several pieces of work related to the modelling and simulation of peer-to-peer networks.

3.2.1 Point-to-point Packet-level Simulation

Peer-to-peer systems are influenced significantly by the underlay network on which it resides, but most studies have used simple models of the underlay network, reducing the accuracy of the analyses. He et al. [16] propose the use of packet-level simulation; they developed a peer-to-peer network simulator which sits atop low-level network simulators such as ns2 [37], GTNets [31] and pdns [29].

The model He et al. use is as follows: an Internet topology is generated, and a number of leaf nodes are attached to “stubs” in the topology. Some of these leaf nodes will be attached to “peer” nodes in the overlay network. Peer activity will be simulated, and communication between the peers will be simulated right down to the communication level, using TCP/IP simulation in one of the low-level network simulators.

He et al. have had some success in their method; their initial experiments simulated 1440 Gnutella servents over 16 physical machines, simulating the effects of the underlying network on the Gnutella network. The measurements included performance (number of queries, throughput, success ratio, and average hit time) versus background traffic. A significant problem with their results is that 1440 Gnutella servents is a relatively minuscule network when considering real file-sharing networks. They report their intention to use a parallel and distributed version of ns2 called pdns to extend their experiments to simulation of 10000 Gnutella servents on 100000 or more Internet nodes; this is more representative of actual networks, but it remains to be seen how viable packet-level simulation is of such large networks, given the obvious scalability concerns.
3.2.2 Peer-behaviour and Evaluation Metrics

To assess how good a protocol is, we must model peer-behaviour, and have suitable evaluation metrics. By modeling peer-behaviour we can put the protocol into context - i.e. examine how the protocol would operate in the environment we are modeling.

Merugu, Srinivasan and Zegura [24] developed a peer-to-peer network simulator used to analyse various metrics: query hit success rate, nearest query hit (on the underlay network), connectivity, and stress imposed on the underlay network. These metrics are far from exhaustive; further metrics could (and probably should) include hop-count, bandwidth usage, and time taken for queries.

Merugu, Srinivasan and Zegura state that the goals and capabilities of their simulator includes scalability. Their simulator (\textit{p-sim}) handles “thousands” of nodes: investigation of the source code reveals a maximum (albeit reconfigurable) number of 8192 nodes. This is insufficient for analysing macroscopic properties of peer-to-peer networks.

This work on \textit{p-sim} and peer-to-peer network modeling does, however, offer some useful information regarding peer-behaviour. Merugu, Srinivasan and Zegura model peer arrivals and departures, and file popularity and availability.

3.2.3 High-level Modeling

Whereas many simulators take a mid-to-low level approach to peer-to-peer network simulation, such as \textit{p-sim} [24] and the peer-to-peer framework simulator developed by He et al. [16], there exist higher level models and simulators. Schlosser, Condie and Kamvar [34] present one such simulator, called the \textit{Query-Cycle Simulator}.

Schlosser, Condie and Kamvar note that macroscopic behaviour of a peer-to-peer network is dependent on “local, peer-level parameters” - namely content distribution and peer behaviour. Of particular interest are the probability functions defined for modeling content distribution - what categories nodes are interested in, and the availability of files within that category - and of up-time and session duration, and query activity.

As Schlosser, Condie and Kamvar note, their model is lacking in several areas – particularly, how sources are selected for downloads, and how bandwidth is modeled. In their model, a node’s bandwidth is only affected by downloading and uploading files, and not by issuing, receiving of forwarding queries or other protocol messages. Their model is highly mathematical, but this is necessary to
model properties which would require user-interaction in the real system. Their work is in stark contrast to that of He et al.; whereas Schlosser, Condie and Kamvar focus on simulating high-level details such as the availability of files over time, most other work focuses on lower-level details such as query performance.

Finally, it should be noted that Schlosser, Condie and Kamvar’s work is only relevant to simulation of peer-to-peer file-sharing networks. Other simulators take a more general approach, focusing on the analysis of properties related to searching, topology generation, and the effects of overlay networks and the underlying network on one another.

3.3 Summary

Peer-to-peer networks are a sort of overlay network in which routing relies on no (or minimal) centralised infrastructure. Peers in a network play the role of both server and client, as opposed to traditional client-server architectures, where the roles are distinct. Finally, peer-to-peer networks are typically very large, and distributed, and thus are often adversely affected by scalability.

Peer-to-peer network simulators seem to vary widely in their scalability. There is a trade-off between scalability and accuracy - namely, modeling the underlying network behaviour. Packet-level simulation gives accurate results, but suffers through its highly intensive computation. On the other hand, high-level models require us to either significantly simplify our model of the underlying network, or ignore it entirely.

Most peer-to-peer network simulators to date have targeted simulation sizes of thousands of peer-nodes. For distributed search algorithms like Gnutella [15], thousands of nodes do not pose a problem. However, Gnutella (at least the older, but still most recent “stable” version, Gnutella 0.4) is widely known to not scale well, which led to semi-centralisation via the addition of ultrapeers. This indicates that thousands of nodes are insufficient for making claims about the scalability of peer-to-peer routing algorithms; for that, we would need hundreds of thousands, or even millions of nodes [32]. It is for this reason that we have chosen peer-to-peer networks as our application for investigating distributed simulation of large systems.
CHAPTER 4

Methodology

In this chapter we will discuss our approach to simulation of overlay (peer-to-peer) networks, and the architecture and history of our distributed simulator, dnet.

4.1 General Approach

There are various approaches one can take to simulation of peer-to-peer networks. In general terms, they come under packet-level, high-level, and a hybrid approach. He et al. have developed a packet-level simulator, which is run on top of standard network simulators [16]. High-level approaches tend to deal with statistical analysis, and modelling systems at the macroscopic level rather than simulating them down to the packet-level. Gil et al. have developed p2psim [14], which takes a hybrid approach, by simulating direct transmissions, and thus reducing the work required by the simulator.

There is no general answer to which approach to simulation is best; it differs with the users’ needs. For example, researchers studying the effects of peer-to-peer networks on Internet infrastructure will most likely want to look at the packet-level, as they can then have greater accuracy at the expense of time, memory and processing consumption. Similarly, researchers studying the effects of small-world network topologies in peer-to-peer networks will most likely want to use a high-level model, or hybrid approach.

We are most interested in simulation of peer-to-peer networks at a mid-level, and so have opted to take a hybrid approach. This means that we sacrifice accuracy in the Internet (i.e. the underlay network) layer to make the simulation more practical. While this means that we will never capture the exact behaviour of the Internet, we can still aim to provide a model that captures macroscopic details, such that we can reliably compare the macroscopic behaviour of our simulated peer-to-peer networks to those in the real-world.
4.2 \textit{dnet}: distributed overlay network simulator

We have developed a distributed overlay network simulator, called \textit{dnet}, for simulation of peer-to-peer network routing/search protocols. \textit{dnet} was developed with two major goals in mind:

- **Ease of use.** To be useful, we feel that a simulator should be as simple to learn as possible. Simulated applications should look similar to the real-world equivalent. The goal is to make the simulator accessible to a wide range of researchers, from university students to developers in the industry.

- **Efficiency and scalability.** The simulator should be powerful enough to simulate peer-to-peer network sizes similar to reality: at least tens of thousands of nodes, but preferably hundreds of thousands.

\textit{dnet} was developed in ISO C++ to be highly modular and efficient, and exports a simple, ISO C99 interface to applications. In section 4.2.1 we will discuss the general architecture of \textit{dnet}.

We have implemented a uniprocessor sequential event-scheduler, as well as two distributed event-schedulers: derivative implementations of the Chandy-Misra deadlock-avoidance scheme, and Jefferson’s Time Warp scheme. Section 4.2.2 describes the uniprocessor sequential implementation, Section 4.2.3 describes the Chandy-Misra implementation, and Section 4.2.4 describes the Time Warp implementation.

In accordance with our general approach, as described in Section 4.1, \textit{dnet} employs an end-to-end model of the internetwork that underlies our simulated overlay networks. Section 4.2.5 describes how we calculate latencies and hop-counts between overlay network nodes.

4.2.1 General Architecture and Terminology

There exist two distinct halves to \textit{dnet}: the kernel, and user-modules/applications. As with \textit{cnet}, the simulator upon which \textit{dnet} was initially based, we have used terminology similar to that used in operating systems. The kernel is responsible for scheduling events and managing the internals of the simulator. User-modules interact with the kernel via an API to specify handlers for events.

We have identified and decomposed the kernel into a number of components (Figure 4.1). The major components are the event manager and node manager. These two components also interface to other important sub-components, namely the event-queue and underlying network model.
The event manager is charged with the scheduling and processing of events; this is where the majority of the work is performed, and where the major differences between simulation schemes are. Event managers provide an interface to event-queues, which exist in an independent module in *dnet*. Our queue module is developed using C++ templates, so they can be reused by other projects.

Node managers create and destroy nodes in the overlay network, and provide an interface between the simulator and underlying network topology model. The underlying network topology model subsystem is completely independent of other subsystems in *dnet*, and provides a means of calculating lower-bound latencies and hop-counts between end-points in the network.

There is a one-to-one mapping between Node objects, which are created by node-managers, and simulated nodes in the overlay network. Node objects present an interface to the simulator for sending messages between overlay network nodes; doing this creates an event in the event-manager, which will be processed on behalf of the receiving node.

**Event-Handling**

All events are handled in user-modules, and the handler code is written in plain ISO C. User-modules are compiled as dynamic libraries, and linked into the kernel at run-time. Every simulated node will have its own user-module. To produce this effect, we use a concept borrowed from *cnet*; we copy the data-segments of the dynamic library into a different memory-buffer for each node. When a node is scheduled to handle an event, it is *swapped in*. Swapping in a node consists of copying that node’s memory buffer into the locations in memory where the dynamic library’s data-segments reside. Swapping a node *out* of memory consists of doing the reverse, copying the dynamic library’s data-segments into the node’s
buffer. We have also developed a new method for achieving the same functionality, but with much less overhead (depending on the sizes of the data segments). This new method involves manipulating the dynamic library’s Global Offset Table, to redirect data references at run-time. For example, 500,000 iterations of 4 lots of swapping in and out a user-module of roughly 512 bytes takes 1.4 seconds with the new method, as opposed to 2.4 seconds with the old method (see `src/dataset/test.cpp` for more details on the experiments). We will not go into the finer details of these methods, however, as that would be a departure from the main discussion.

When the simulation begins, the kernel executes a function, which resides in the dynamic library, called `boot`. This function will set up event-handlers for other events, such as `received a message`, or `timer expired`. In the `boot` function, or any other event-handler callbacks, the user-module may call API functions to create events. For example, a `send message` call will create a message event bound for a node with a specified address.

Distributed Simulation Architecture

We have implemented three pairs of event and node managers. The first one developed was the sequential scheduler, which does not require anything more than already discussed. The second and third schedulers developed are distributed, and so require additional infrastructure to perform necessary inter-simulator communication.

Our distributed simulation framework uses Message Passing Interface (MPI) [1]. MPI is the de facto standard for message passing in parallel/distributed computing. We have chosen MPI over alternatives, such as Parallel Virtual Machine (PVM) [2], mainly based on how well established it is, and how portable it is over heterogeneous environments. MPI provides transparent access to different methods of message-passing, such as standard Ethernet, and shared-memory. MPI enables the transparent loading and concurrent execution of our simulators in a network. By default, `dnet` uses MPI to perform inter-simulator communication. We have, however, provided an interface for extending `dnet` for use with alternative communication protocols, such as the User Datagram Protocol (UDP), or the Reliable Datagram Protocol (RDP). Figure 4.2 illustrates the distributed infrastructure of `dnet`.

18
4.2.2 Sequential Implementation

Sequential simulation is the traditional, and most widely used method of simulation. Sequential simulation is simply a simulation in which all events are processed in order of time, with no concurrency. Additionally, sequential simulation usually refers to uniprocessor simulations; there is not a lot to be gained from distributing a simulation and restricting it to sequential processing.

We have implemented a sequential event scheduler in *dnet* for the purpose of benchmarking our distributed schedulers, and also for testing auxiliary subsystems without the non-determinism of distributed/parallel computation. Algorithm 1 outlines the process. There is a simple loop of taking the earliest event in the future-event queue and processing the event. Processing the event involves invoking callbacks defined in application code.

**Algorithm 1** Sequential Simulation

```
1: time ← 0
2: while time < stopTime ∧ size(queue) > 0 do
3:    event ← pop(queue)
4:    time ← timestamp(event)
5:    process(event)
6: end while
```

There are two main deficiencies in sequential simulation, and they are both related to resource constraints. Firstly, as simulation sizes increase, so do memory
requirements. Each node in the simulation will consume some amount of memory, as will each event. Depending on usage, a single machine may eventually be unable to accommodate the memory requirements. Secondly, sequential simulations are restricted to processing events one at a time. As event-set sizes increase, so does the time taken to complete the simulation. It is possible to run some of the events concurrently; for example, two events with the same timestamp can not have an effect on each other, since we assume all newly created events will have a timestamp greater than the current time. Sequential simulations are restricted from taking advantage of this.

4.2.3 Chandy-Misra Implementation

The second event scheduler we developed was based on the conservative simulation scheme developed by Chandy and Misra [25]. This is a fairly basic scheme, which avoids deadlock rather than detecting and addressing it. This is accomplished by sending special control messages to tell other simulators that no event will be created up to a certain time.

Whilst there have not been any significant departures from the original algorithm proposed, we will still describe our implementation, since the work of Chandy and Misra on which our work is based was highly theoretical. We will point out some of the issues of practicality with our derivative algorithm, and also highlight any modifications and additions.

Sending Messages Between Peer-Nodes

In the sequential scheme described in Section 4.2.2, when a simulated node sends a message to another node an event is placed in the future-event queue. In our conservative scheme, we must alter this behaviour, since not all nodes reside on the same simulator.

Let $S$ and $D$ be two simulated nodes, and suppose $S$ calls the API routine for sending a message to another node, in this case $D$. If $S$ and $D$ both reside on the same simulator, then we immediately insert an event into the local future-event queue. If $S$ and $D$ reside on two separate simulators, then $S$ places the event in a queue in the outgoing channel to $D$’s simulator. The event will stay in the queue until it is determined that the destination simulator is ready to receive it.
Lookahead

Conservative distributed simulation relies on a property known as *lookahead* to perform efficiently. Lookahead is the amount of time in which a simulator can be certain that it will not send an event to another simulator. By sending this information to each other, the simulators avoid deadlock. Suppose simulators didn’t send this information, and let A and B be two simulators wanting to communicate with each other. Let $E_1$ and $E_2$ be the respective events that A and B want to send to each other, where $E_1$ has timestamp 100, and $E_2$ has timestamp 101. Now, given that lookahead is not used, A and B may not be able to ascertain whether or not the other will send an event to them that will affect the events they want to send. For example, suppose A knows that the last event B sent to it had a timestamp of 90. Without knowing B’s lookahead value, A cannot determine whether or not it is safe to send $E_1$ to B, given that it could potentially be canceled by receiving an earlier event. The same will be true of B; it will not be able to send $E_2$ to A, since it can’t determine whether or not it is safe. Therefore, both simulators enter deadlock. Using lookahead, the simulators can tell each other when it is safe to send events.

We use a very simple, yet elegant approach to calculating lookahead, based on the lower-bound latencies between simulated nodes on remote simulators. In this situation, latency is defined as the amount of simulated time that elapses between when the source sent the message, and when the resultant event is processed and the destination’s message reception callback invoked. We assign a lookahead value for each outgoing channel for every simulator, by determining the minimum lower-bound latency between any two nodes in the two simulators connected by the channel. Formally:

Let $S$ be the set of simulators,

Let $C(i, j)$ be an outgoing channel from simulator $i$ to simulator $j$,

Let $N_k$ be the set of nodes that reside on simulator $k$,

Let $L(m, n)$ be the lower bound on the latency of messages sent from $m$ to $n$.

$$(\forall i, j \in S) \ C(i, j) = \min((\forall m \in N_i, n \in N_j) \ L(m, n))$$
Algorithm 2 Entry-point to Chandy-Misra Algorithm

1: for all $C \in \text{Channels}$ do
2:   Set lookahead for C
3: end for
4: $\text{clock} \leftarrow 0$
5: $\text{moreevents} \leftarrow \text{true}$
6: while $\text{clock} < \text{stoptime} \land \text{moreevents} = \text{true}$ do
7:   $\text{moreevents} \leftarrow \text{run-phase}()$
8: end while

Main Algorithm

The bulk of the synchronisation algorithm in our Chandy-Misra derivative implementation is performed in a single loop, which is run until the local-time is equal to the stop-time. Algorithm 2 describes the process of the top-level loop. The routine run-phase referenced by Algorithm 2 is where most of the work is done. This is described in Algorithm 3.

Essentially, the algorithm does the following in a loop:

1. Process all events up until the current clock value.
2. Send all enqueued events to remote simulators up to the current clock value.
3. Receive messages from all simulators, until a NULL-message is received from each one. Once a NULL-message is received from a simulator, stop accepting messages from that simulator for this iteration (phase).
4. Set the local clock value to the minimum timestamp of all of the messages received.
5. Determine whether any events exist locally or remotely; if none exist, halt the simulation.

Note that one must send messages asynchronously. If they blocked, deadlock would occur when two simulators attempt to send to each other, since neither would reach the receiving state. Note also that the behaviour of the function call terminated is specified in a comment in Line 19 of Algorithm 3. Every NULL-message has a flag which, when the message is sent, is set to true iff there are no locally queued events. The function terminated simply returns the value of the flag of the last NULL-message received on a specified channel.
Algorithm 3 Chandy-Misra Derivative Algorithm

1: Process all locally queued events with timestamp \( \leq \) clock.
2: more-events \( \leftarrow (\max(\text{size(queue)}, \{\text{nqueued}(c) \mid \forall c \in \text{Channels}\}) > 0)\)
3: for all \( C_i \in \text{Channels} \) do
4: \( \{C_i \text{ is the channel from the local simulator to simulator } i.\} \)
5: Send all queued messages for \( i \) on \( C_i. \)
6: Send a NULL-message on \( C_i, \) with timestamp \( \text{clock} + \text{lookahead} \)
7: end for
8: lowest \( \leftarrow \infty \)
9: incomplete \( \leftarrow S \setminus \{\text{Local}\} \)
   \{Assign the set of remote simulators to “incomplete”\}
10: while incomplete \( \neq \emptyset \) do
11: Receive message \( m \) from any channel \( C \in \{C_i \mid \forall i \in \text{incomplete}\} \)
12: if \( m \) is an event-message then
13: Build and enqueue event based on message payload
14: else if \( m \) is a NULL-message then
15: incomplete \( \leftarrow \text{incomplete} \setminus \{C\} \)
16: end if
17: lowest \( = \min(\text{lowest}, \text{timestamp}(m)) \)
18: end while
19: more-events = more-events \( \lor (\{\text{terminated}(c) \mid \forall c \in \text{Channels}\} \subset \text{Channels}) \)
   \{More-events is true \( \iff \) at least one simulator has a local event;
   terminated\( (c) \) is true \( \iff \) we received a NULL-message on channel \( c \) which
   said the sending simulator has no more local events\}
20: \( \text{clock} \leftarrow \text{lowest} \)
21: return more-events

4.2.4 Time Warp Implementation

The third and final event scheduler we developed is based on Time Warp, which
was devised by Jefferson [17]. To improve our implementation, we have imple-
mented lazy cancellation [12], and Tomlinson and Garg’s GVT (Global Virtual
Time) detection algorithm [36]. We will first describe our initial implementation,
and subsequently describe these additions.

The fundamental events in Time Warp are:

- Event Processing,
- Sending/Receiving Event/Anti-messages,
- Checkpointing/Rollback.
Algorithm 4 Entry-point to Time Warp Algorithm

1: checkpoint()
   {We must have a checkpoint for time = $t_0$}
2: while not finished() do
3:     receive_messages($n$)
       {Receive up to $n$ messages, where $n$ is some constant. This may mean the
        creation of events, or receiving an anti-message and canceling an event (see
        Algorithm 6.)}
4:     process_batch($k$)
       {Process up to $k$ queued events, where $k$ is some constant.}
5: end while
6: fossil_collect()

Algorithm 5 Time Warp: process_event(event)

1: if time(event) $<$ TIMENOW then
2:     rollback(time(event))
       {TIMENOW is the time of the last successfully processed event.}
3: end if
4: process(event)
       {Call the standard event-processing procedure.}
5: checkpoint()
6: processed_events ← processed_events $\cup$ \{event\}

Event processing is much like the aforementioned simulation schemes, but has
two modifications. Firstly, if the timestamp of the event being processed is in the
past (has a value less than the local clock), then a rollback is initiated. After the
rollback has completed, the event is processed as normal. Secondly, local nodes’
states must be saved whenever local virtual time advances.

Algorithm 4 illustrates what occurs at the entry-point of the Time Warp
algorithm, and Algorithm 5 illustrates the processing of an event.

Sending and Receiving Event-messages

When event-messages are sent, they are issued a unique id, and this id is recorded
in the outgoing channel. When a rollback occurs, event-messages must be can-
celed by issuing anti-messages. To do this, we need the id of the original event-
message. When a simulator receives an event-message, it records the id of the
message, and creates a mapping from that id to the generated event. This allows
the simulator to later find the event if it receives an anti-message.
Algorithm 6 Time Warp: receive_anticomessage(msg_id)

1: event ← event_from_msgid(msg_id)
   \{event_from_msgid takes a message id, and returns the event which was created from the message with that id.\}
2: if event ∈ processed_events then
3:    processed_events ← processed_events \ \{event\}
4:    rollback(time(event))
5: else
6:    dequeue(event) \{Remove event from future event queue.\}
7: end if

Sending and Receiving Anti-messages

To cancel a message, a corresponding anti-message must be issued. Jefferson defines three cases for receiving an anti-message which corresponds to a message M. They are as follows:

1. M has been received, but not yet processed.
2. M has been received, and processed.
3. M has not yet been received.

Cases one and two must be covered in all implementations, but case three is only necessary when using a communication transport which does not preserve order. In dnet, we have only implemented a FIFO transport interface, and so we have only needed to cover the first two cases.

Algorithm 6 illustrates what happens when a simulator receives an anti-message.

Checkpointing and Rollback

Under a Time Warp scheme, a simulator must periodically save its state, so it can restore that state at a later time if a causality error occurs. The act of saving state is termed checkpointing. Since simulation states in dnet are dynamic (not discernible at compile time), these actions have become quite complex to implement. Checkpointing is simple enough: to save the state of the simulator, we simply store the data-segment for the node associated with the last event executed. Rolling back is more complex, and we will spend the next several paragraphs discussing how it is accomplished. Algorithm 7 illustrates the process of rolling back to a specified time.
Algorithm 7 Time Warp: rollback(to)

1: \text{cp} \leftarrow \text{the latest checkpoint made before time } to
2: revert(cp) \{Revert data-segments of local-nodes.\}
3: send antimesages \{Only if lazy-cancellation is not being used.\}
4: temp\_queue \leftarrow \emptyset
5: \textbf{while} size(event-queue) > 0 \textbf{do}
6: \quad \text{event} \leftarrow \text{pop(event-queue)}
7: \quad \text{time} \leftarrow \text{issue\_time(event)}
\quad \text{\{issue\_time is the simulated time when the event was created\}}
8: \quad \textbf{if} \text{issue\_time} < to \lor \text{is\_foreign(event)} \textbf{then}
9: \quad \quad \text{push(temp\_queue, event)}
\quad \text{\{is\_foreign is true } \iff \text{ the event was created as the result of a message from a remote simulator\}}
10: \quad \textbf{end if}
11: \textbf{end while}
12: \textbf{while} size(temp\_queue) > 0 \textbf{do}
13: \quad \text{push(event-queue, pop(temp\_queue))}
14: \textbf{end while}
15: \textbf{for all} \text{ event} \in \{e \mid \forall e \in \text{processed\_events, time}(e) \geq to\} \textbf{do}
16: \quad \text{processed\_events} \leftarrow \text{processed\_events} \setminus \{\text{event}\}
17: \quad \text{undo(event)} \{\text{call all of the registered undo-callbacks for this event}\}
18: \quad \textbf{if} \text{issue\_time(event)} < to \lor \text{is\_foreign(event)} \textbf{then}
19: \quad \quad \text{push(event-queue, event)} \{\text{We want to reprocess the event, so put it back in the future event queue.}\}
20: \quad \textbf{end if}
21: \textbf{end for}

Applications must explicitly tell the simulator how to undo modifications to data-structures allocated on the heap. This is in contradiction to our goal of transparency to the application, but is inevitable. In an application’s event handler callback, API calls may be made to notify the kernel of what to do when that event is undone. Specifically, the event-handler will present the kernel with a list of callbacks which will perform actions such as freeing memory. Additionally, events can specify what to do when it is committed. That is, what to do when the simulator determines that the event will never be rolled back, and its memory can be safely freed.

Take, for example, the C code in Listing 4.1. DNETX\_queue\_push is a function that may be called from within application event handlers, to add an item of data to a queue data-structure. If the event that calls DNETX\_queue\_push is undone by the simulator, then the memory allocated via malloc must be freed. To
void DNETX_queue_push(DNETX_Queue* q, dnet_data value)
{
    DNETX_QueueItem* new_item = (DNETX_QueueItem*) malloc(sizeof(DNETX_QueueItem));
    new_item->value = value;
    new_item->next = NULL;

    if (q->size++ == 0)
        q->head = q->tail = new_item;
    else
    {
        q->tail->next = new_item;
        q->tail = new_item;
    }

    DNET_on_undo(free_data, (dnet_data)(uint32_t)new_item);
}

Listing 4.1: Example of DNET_on_undo usage

ensure this is done, DNET_on_undo is called, passing in the address of a function called free_data, and the address of the memory that was allocated. The function free_data, not shown in the listing, is a simple callback function defined in the application that frees the memory which is pointed to by the argument free_data receives. Finally, note that we only need to call DNET_on_undo for actions that modify data on the heap; the application’s data-segments are automatically restored.

The reader may question why we did not choose to develop dnet in a language with automatic garbage collection, such as Java. Our primary reason for not doing this was the need for the highest performance possible. As opposed to our system, where we can determine precisely when memory can be freed, Java’s garbage collection is non-deterministic, and we felt such a system would be unsuitable for supporting very large simulations.

Whilst rollbacks must be carefully considered by users developing applications, we believe the simple API we have implemented makes the problem much easier to manage. To alleviate the problem further, we have implemented a small library of functions that can be linked with applications, called dnetx (dnet extensions). The purpose of dnetx is, amongst other things, to encapsulate some of this rollback handling behaviour. The function in Listing 4.1 is a part of a queue data-structure that is rollback-safe. Rollback-safe means that it will oper-
ate properly in light of rollbacks.

Events are committed once the scheduler has determined it is safe to do so. For the sequential and Chandy-Misra schedulers, an event is committed as soon as it has been processed. In Time Warp, an event is committed when the simulator can determine that it will not rollback before the event’s time. As with the undoing of events, event-handlers may create a callback to be invoked when the event is committed.

Lazy Cancellation

In our initial experiments, we noticed extremely poor performance in our Time Warp implementation. We ascertained that the cause of the deficiency was a behaviour known as cascading rollbacks [12]. Cascading rollbacks occur when a process rolls back and emits an anti-message, which eventuates in an anti-message being sent to the same process, causing another rollback. This behaviour can continue, and send the simulation into an unstable state. One method used in an attempt to eliminate cascading rollbacks is lazy cancellation. In this sub-section we shall describe what lazy cancellation is, and how it is implemented in dnet.

In the basic form of Time Warp, when a process rolls back and undoes an event, the process immediately sends anti-messages for all of the messages sent as a result of that event. Often, though, a process will send exactly the same message after the rollback, meaning the anti-message was unnecessary. This unnecessary anti-message could have caused the receiving process to roll back, and send out further unnecessary anti-messages. Lazy cancellation is a modification of Time Warp, which delays the propagation of anti-messages until it is determined that they are absolutely necessary. The following paragraph illustrates the way in which it works.

Suppose simulator $S_i$ just sent a message to simulator $S_j$ as a result of event $e$ with timestamp $t$, and that now simulator $S_i$ is rolling back to before time $t$. Prior to the rollback, $S_i$’s local clock was at time $t_s$. $S_i$ undoes all of the events, but does not send anti-messages for them. It then executes events up until $t_s$, and after each change in the local clock, checks if a message that was sent prior to the rollback was not sent since the rollback; if such a message exists, it should now be canceled by transmitting an anti-message.

Lazy cancellation is not without its own problems, however. There are two major considerations we must make when deciding to use it. Firstly, lazy cancellation imposes overhead in event-processing. After a rollback, whenever a message is to be sent to a remote simulator, it must first be compared against a list of messages sent prior to the rollback. Unfortunately this means comparing
the payloads of the messages, which can be potentially quite large (dnet imposes a maximum payload limit of 8192 bytes). This problem can be reduced by saving a hash of the original message (e.g. using MD5), which allows a faster comparison. The second problem is that we will inevitably delay the propagation of anti-messages that need to be sent anyway.

GVT Detection

Saving the state of the simulator will quickly exhaust memory as a simulation progresses if we can not identify which states can be safely discarded. To do this, the simulators need to be able to determine the Global Virtual Time. We can guarantee that no simulator will ever roll back to a time before the GVT, and so we can discard all but one state for the times before the GVT.

There exists a high degree of interdependence between GVT algorithms and the internal structures of the Time Warp algorithm. It might be possible to decouple them by providing callback-hooks for almost every event-scheduler function, but we felt this would be a bad idea for the following reasons. Firstly, it would increase the complexity of our code significantly. Secondly, the callback overhead would likely be significant, since the event-scheduler runs frequently and even the smallest performance hits accrue.

We have developed a simple GVT detection algorithm, based on the work of Tomlinson and Garg [36]. Their approach is to determine how many total messages have been sent and how many have been received. When the two values are equal, then GVT has been calculated. This is a simplification; there are issues related to rollbacks, as well. For a full definition of the algorithm, the reader is directed to Tomlinson and Garg’s original paper.

Whereas most other GVT algorithms work by calculating GVT, our chosen algorithm detects when it has occurred. The algorithm works by sending GVT target messages to all simulators. These target messages have a specified timestamp, which is the time for which we want to detect that GVT has passed. Suppose a GVT target with timestamp \( t \) is sent out; when GVT \( t \), then the detection will be complete. Simulators process these target messages like other events – they insert them into their event-queue, and process them when the local clock is equal to the target message’s timestamp. Processing them involves gathering some local information about the simulation, and sending the information in a GVT report to the initiator of the algorithm.

Each use of Tomlinson and Garg’s algorithm is independent of prior detections. In a typical simulation, we will want to calculate GVT continuously. To produce this effect, we simply create a timer in the root simulator that produces
a GVT target every so often. The root simulator is some unique simulator, typi-
cally the one with rank zero. The algorithm also only notifies the initiator when
the detection is complete. Since all simulators need to know GVT, rather than
replicating the algorithm over all simulators (which is possible), we simply push
the GVT from the root to all other simulators upon detection.

At any stage a simulator’s local time may be infinity, if the simulator’s event
queue is empty. If the simulator requests GVT detection for infinity, then no
detections will occur between when the request is made and the simulation ends.
In other words, this behaviour is potentially as bad as having no GVT detection
at all. In practice this has not occurred so frequently that it has been a great
problem, but it does happen occasionally, and introduces error in experimental
results.

4.2.5 Internet Topology Model

We have developed a topology subsystem in dnet which is responsible for rep-
representing the network underlying the simulated overlay network. Specifically, it
is responsible for calculating lower-bounds on latencies and hop-counts between
host nodes. In this section we will discuss the structure of the topology subsys-
tem, and how we calculate latencies and hop-counts.

Subsystem Structure

We have taken a strategy similar to that taken in p2psim [14] to calculating
latency and hop-count values. We first create an Internet-like substrate, using
an Internet-topology generator, such as GT-ITM [6], BRITE [23], or Inet [39].
We then select a set of nodes from this substrate, based on their degrees of
connectivity, and mark them as providers. Providers are analogous to Internet
Service Providers, in that they provide a point of connection for host nodes.
Host nodes are then attached to providers in some random manner. These host
nodes represent the computers which run peer-to-peer applications, and connect
to one another to form an overlay network. Figure 4.3 illustrates the creation of
a topology in dnet.

We have implemented an interface to the Stanford GraphBase [19] file-format,
for loading underlying network topologies from a file. This file-format is used
by GT-ITM. We generate Internet topologies externally to dnet, and they are
loaded at run-time. We decided to use GT-ITM for three main reasons: firstly,
it is widely used by other researchers in the field. For example, He. et al have
used GT-ITM, as does ns2. Secondly, GT-ITM produces a router-level topology
Figure 4.3: A step-by-step example of generating a full topology in dnet. Note that we’ve illustrated the process with an abnormally small network; generated topologies will typically have thousands to hundreds of thousands of nodes.
using the *Transit-Stub* method, as opposed to an *Autonomous-System* (AS) level network, which *Inet* produces. Router-level graphs can more readily be used for estimating latencies by calculating the Euclidean distances between nodes, whereas AS graphs represent connections between sub-networks, and would thus require modelling those sub-networks. Finally, we came to *BRITE*; this application is able to generate either topology representation, however we encountered erroneous behaviour with it in our experimentation.

**Approximating Latency and Hop-Counts**

Our latency and hop-count calculations assume the following: latencies and hop-counts will fluctuate over time, but not change significantly, disregarding catastrophic network failures. Of course, the fluctuation is difficult to model, but we have produced a framework which may be extended to factor in various attributes of a network. Our latency and hop-count calculations are based on *end-to-end* approximations. We calculate some lower-bound latency/hop-count values based on the shortest-path between two host nodes, and increase the values according to some formulae based on parameters such as link-bandwidths, and background traffic.

Presently, the lower-bound latency between two nodes is based on the sum of the Euclidean distances between the intermediate nodes on the shortest-path. The lower-bound hop-count is simply the number of links in the shortest-path. Fluctuation formulae are, at present, not implemented, but there exists a framework for implementing them.
CHAPTER 5

Experiments and Results

In this chapter we will discuss experiments performed, and results obtained, to determine the feasibility of using distributed simulation for simulating peer-to-peer networks. In Section 5.1 we will discuss our experiments related to event-queueing, and in Section 5.2 we will discuss our experiments related to the performance of the Chandy-Misra and Time Warp synchronisation schemes.

5.1 Event-Queueing

Given that simulators spend a non-trivial amount of their processing time on the enqueueing and dequeueing of events, we feel it is important to report on the performance of our event-queue data structures. In Section 5.1.1 we will describe the behaviour and complexity of our linear queue structure. In Section 5.1.2 we describe the behaviour and complexity of our implementation of Calendar Queue, and also describe a modification of the original definition to allow for the insertion of events in the past.

Note that we have not provided comprehensive empirical data for our basic queue implementations. This is not an oversight – our results reflect those of Jones [18] and Brown [4]. We have performed a single experiment, to inspire confidence in the validity of our implementation, and to highlight the performance of the queue implementations with a typical event distribution. This experiment is discussed in Section 5.1.3.

5.1.1 Linear Queue

Linear queues are the simplest data structure one can use for event queueing. As the name suggests, they have linear time complexity for their hold time (the hold time is the sum of enqueue and dequeue times). They typically consist of a simple linked-list which uses insertion sort when enqueueing items. Naïve
simulators will typically use linear queues, and will have poor performance when event sets increase in size.

5.1.2 Calendar Queue

Calendar Queues are a form of priority queue which exhibit an O(1) hold time [4]. They are representative of a desk-calendar, where items in the queue map to events on the calendar. Calendar Queues partition events into an array of “buckets”, which are sorted linked-lists. The enqueue algorithm calculates which bucket to use, based on the priority of the item, and expands and decreases the number of buckets according to the number of events in the queue.

\textit{dnet} uses Calendar Queues by default, since they outperform linear queues in all but the smallest of event-set sizes.

Modifications for Handling Past-Events

When inserting events into a calendar queue in the past, under certain conditions, a dequeue will not return the event in the past (which has the lower timestamp) (Figure 5.1). To enable the use of Calendar Queues with Time Warp, we made a very simple change: when enqueueing an item, if the item’s timestamp is less than the timestamp of the last event dequeued, then we should replace some internal attribute of the queue (“lasttime”, “lastbucket” and “buckettop” – see the original definition of Calendar Queue [4]) with those of the new item. This ensures that the next dequeue operation will begin its search in the newly enqueued item’s bucket. The resulting data structure supports enqueueing of past-events, with only 5 extra lines of code.

One should be concerned about the implications these modifications have on performance. When comparing the original and modified versions of the Calendar Queue for hold times when enqueueing events \textit{only in the future}, there was no performance degradation. This is to be expected, as if events are always in the future, the code modification will only incur a comparison and a jump instruction. It would not make sense to compare the original and modified versions’ performance when distributing past-events, since the original version will not handle them properly.
Initial state | enqueue(380) | dequeue() : 380 | enqueue(380) | enqueue(333)
---|---|---|---|---
lastbucket: 0 | lastbucket: 0 | lastbucket: 0 | lastbucket: 0 | lastbucket: 0
lasttime: 0 | lasttime: 0 | lasttime: 380 | lasttime: 380 | lasttime: 380
buckettop: 1 | buckettop: 1 | buckettop: 381 | buckettop: 381 | buckettop: 381

buckets | buckets | buckets | buckets | buckets
0 empty | 0 380, ... | 0 empty | 0 380, ... | 0 380, ...
1 empty | 1 empty | 1 empty | 1 empty | 1 333, ...

Figure 5.1: The boxes above depict the state of a calendar queue after several enqueue and dequeue operations. The next dequeue operation will return the event with timestamp 380, due to the fact that the event with timestamp 333 was inserted “in the past”.

### 5.1.3 Performance Comparison

We have performed a simple experiment to highlight the difference in performance of our linear and calendar queue implementations. Note that we are only supplying performance data for event distributions where all events are created in the future. See Section 5.1.2 above for details as to why.

Our experiment is based on the experiments in Jones’ [18] and Brown’s [4] papers. For each queue type, we did the following. We create a queue, and iteratively add elements to the queue, with random priorities (times) taken from a uniform distribution with an expected value of 100. After this, we perform 1000 hold operations – the removal of the earliest element from the queue, and reinsertion of that element into the queue. Prior to reinsertion, the element’s priority (time) is increased by a random value taken from an exponential distribution with an expected value of 100. The time taken to perform this series of hold operations is used for comparison. The comparison is repeated for varying queue sizes, but the number of hold operations is kept constant. To reduce the amount of error in our results, we performed each test 5 times, and took the mean average value, after removing outlying data.

Algorithm 8 formally defines the function used for one test-run. The C++ code is available in ‘src/queueing/hold_performance/hold.cpp’ within the dnet source distribution.
Algorithm 8 Queue Hold-time Performance

1: queue ← \{new queue\}
2: for \( i \leftarrow 1 \) to queue.size do
3: \( e \leftarrow \) new element
4: set.time(e, uniform.random.number)
5: push(queue, e)
6: end for
7: start.time ← get.time()
8: for \( i \leftarrow 1 \) to 1000 do
9: \( e \leftarrow \) pop(queue)
10: set.time(e, get.time(e) + exponential.random.number)
11: push(queue, e)
12: end for
13: total.time ← get.time() − start.time
14: return \( \frac{total.time}{1000} \)

Environment

This experiment was run on an otherwise quiescent single-user computer. The computer comprises a Pentium 4 3.0GHz (with Hyper-Threading) processor, and 512MB DDR RAM. The computer runs the Linux 2.6.13 kernel, and all programs used in testing were compiled with GCC 4.02, with the flags `-O3', `-Wl,-O1', and `-funroll-loops'.

For random-number generation, the GNU Scientific Library [30] version 1.6 was used. We used the default random-number generator { MT19937 (Mersenne-Twister). Each repeated test (i.e. for the same queue-size) used the same seed.

Results

Figure 5.2 shows the results of our performance comparison of our linear and calendar queue implementations. The crossover point where calendar queue becomes more efficient than linear queue is close to only 10 elements. There are very few practical simulations where such a small quantity of events would exist. A noteworthy result of the calendar queue’s performance analysis is that it becomes marginally faster as the number of events is increased. This is most likely due to the fact that at low numbers of events there are not many buckets in the queue, hence there will be more direct searching through the buckets.

36
5.2 Distributed Simulation Performance in \textit{dnet}

In Section 5.2.2 and Section 5.2.3 we will discuss the results of our experiments to analyse the performance of our Chandy-Misra and Time Warp scheduler implementations respectively. In particular, we focus on the \textit{speedup} of both schedulers compared to uniprocessor sequential simulation. Speedup is defined as follows: if \( A \) and \( B \) are two algorithms which take the same input and produce the same output, and \( A \) takes time \( T_A \) and \( B \) takes time \( T_B \), then the speedup of \( A \) over \( B \) is \( \frac{T_B}{T_A} \). eg. A speedup of 2 corresponds to \( A \) being twice as fast as \( B \). In Section 5.3 we will summarise our findings.

We have illustrated the best and worst case speedup for both of our distributed event schedulers. Both cases are simulated by forming a simple ring-topology overlay network (Figure 5.3), and having each node transmit a token to its successor, with a time-to-live of 1000. When a node receives a token, it decrements the time-to-live, and if it is still greater than zero, it is forwarded to its successor. We differentiate between best and worst case performance by specifying how many inter-node connections cross between simulators. The user-module which we wrote for our tests is distributed with \textit{dnet} – it is the file ‘test/test_queue.c’. See Appendix B for details on how to compile and run user modules.
Figure 5.3: A Ring-Topology Overlay Network. The illustration on the left depicts overlay network connectivity which will result in the best simulation performance, since there is minimal inter-simulator communication. The illustration on the right depicts the inverse case.

Each experiment is run with a different number of nodes (100, 500, 1000, 5000 and 10000). Best case simulations will incur $1002N_S$ inter-simulator communications, where $N_S$ is the number of simulators. Worst case simulations incur maximum of $1002N_N$ inter-simulator communications, where $N_N$ is the number of simulated nodes.

For the reasons stated in Section 5.2.1, we have run each experiment 10 times, and taken the mean of the results, after removing outliers.

5.2.1 Distributed Simulation Environment

Our simulation environment is a homogeneous Local Area Network (LAN) of computers. Each computer has an Intel Pentium 4 2.4GHz processor, 512MB RAM, and runs the 2.6.12.1 Linux kernel. GCC 3.4.3 was used to compile dnet, with ‘-O3’ and ‘-Wl,-O1’ optimisations enabled. LAM version 7.1.1 was used for MPI communication. Each computer is equipped with a standard 100Mb Ethernet NIC (Network Interface Card).

Since our simulation environment is shared with others, it is not possible to guarantee our results have been undisturbed by others’ use of the network. We ensured that none of the computers we ran our simulations on were in use by others, but the LAN was still in use, which could potentially have increased inter-simulator communication latency. Furthermore, we did not have exclusive access to the computers, and so there were always background processes active...
concurrently with our simulations. To reduce the effects of both of these possible disturbances, we ran each simulation multiple times and took the mean, after eliminating outlying results.

Virtual Underlying Network Topology

To simulate an overlay network protocol, we require an underlying network topology. Whilst the characteristics of underlying network topologies do have significant impact upon the behaviour of simulated nodes, they do not have a direct effect upon the simulator’s performance in which we are interested. Therefore, we have reduced our underlay network topology to be very small – 100 nodes.

We generated our topology using GT-ITM with the specification file whose contents are in Listing 5.1. The reader is directed to GT-ITM’s documentation for an explanation of each line.

### Listing 5.1: Speedup Experiments: GT-ITM specification file

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ts 1</td>
</tr>
<tr>
<td>2</td>
<td>3 0 0</td>
</tr>
<tr>
<td>3</td>
<td>1 20 2 1.0</td>
</tr>
<tr>
<td>4</td>
<td>4 20 2 0.6</td>
</tr>
<tr>
<td>5</td>
<td>8 10 2 0.42</td>
</tr>
</tbody>
</table>

Line 1 specifies that we are using the Transit-Stub method, and are creating a single graph. The lines after relate to the number of transit domains, and the average number of nodes per transit domain, and their connectivity. A graph with an average of 100 nodes will be generated.

5.2.2 Chandy-Misra implementation

In this section we discuss the results of our speedup comparisons of dnet’s Chandy-Misra scheduler implementation versus dnet’s uniprocessor sequential scheduler.

As can be seen in our results, dnet’s Chandy-Misra implementation exhibits a common trend in its time-performance. Our test with 500 nodes (Figure 5.7) best exemplifies the trend – speedup grows gradually, as the number of simulators is increased, and eventually begins to decay. The increasing speedup is due to the fact that multiple simulators can execute events in parallel. With increasing numbers of simulators involved comes an increased amount of communication and synchronisation. The simulation performance peaks and begins to decay when this increased overhead surpasses the benefit gained from executing more events
Figure 5.4: Speedup for 100 Nodes

Figure 5.5: Speedup for 500 Nodes
Figure 5.6: Speedup for 1000 Nodes

Figure 5.7: Speedup for 5000 Nodes
in parallel. The rates of gain and decay are related to the number of events in the simulation at any one time. For example, the tests for 1000 (Figure 5.6) and 5000 nodes (Figure 5.7) illustrate a similar shape to that of the above mentioned test, but peak at higher numbers of simulators, and have an overall improved speedup.

We have found that, due to the fact that deadlock occurs when logical process topologies contain cycles, the Chandy-Misra scheme is mismatched for applications of systems with dynamic topologies. Overlay networks are one such system, if the underlying network is abstracted away, in favour of an end-to-end network model. In such a scenario, we will obtain the lowest performance possible, since every single subset of the logical processes (of size greater than one) constitutes a cycle, and thus will cause deadlock. In such a situation, all logical processes will execute in lock-step with every other logical process. Nevertheless, there is still reasonable speedup in some cases – for example, in our test-case with 5000 (Figure 5.7) and 10000 nodes (Figure 5.8).
5.2.3 Time Warp implementation

In this section we discuss the results of our speedup comparisons of \texttt{dnet}'s Time Warp scheduler implementation versus \texttt{dnet}'s uniprocessor sequential scheduler. We compare \texttt{dnet}'s default Time Warp implementation, using lazy cancellation, against the uniprocessor sequential scheduler. In all experiments, the GVT detection interval is 0.5 seconds.

With few simulators, the performance of \texttt{dnet}'s Time Warp scheduler is very poor. With two simulators, the 100 node test takes more than twice the time taken by the uniprocessor sequential scheduler. There is an increase in speedup as the number of simulators is increased, which indicates that we can make the Time Warp scheduler perform at least as fast as the uniprocessor scheduler, given enough simulators. Figures 5.4 – 5.8 show similar correlations between the number of simulators and the speedup. Moreover, they show a pronounced correlation between the number of nodes and the speedup. Figure 5.4 shows a negative speedup (the speedup peaks at roughly 33\% of the speed of the uniprocessor sequential simulator) for 100 nodes. None of the tests showed positive speedup for Time Warp; the test with 10000 nodes (Figure 5.8) comes close, however, and suggests a continued upward trend.

At the time of writing, \texttt{dnet} exhibits erroneous behaviour related to GVT detection and message cancellation. This problem does not affect our above results, as we have used lazy-cancellation. With our experiments, no antimessages will be sent when using lazy-cancellation, as there is no randomness in the messages sent between simulated nodes. This is not true when we do not use lazy-cancellation, as messages are canceled regardless of whether or not they will be re-issued. We can still run \texttt{dnet} without lazy-cancellation, but GVT detection must limited to a single use, and that is to perform termination detection. We can still make useful comments, though. As previously mentioned, our experiments are fully non-random, and no inter-simulator messages will ever be canceled. This means that the only rollbacks that will occur are caused by out of order message reception, and never because of message cancellation. Lazy-cancellation and regular Time Warp differ only when (and if) they cancel a message, so we can infer that, in the tests we have performed, regular Time Warp would perform worse than when lazy-cancellation is used.

In the best-case test with 10000 nodes (Figure 5.8), we have only provided results for \( \geq 3 \) simulators for Time Warp. The test case with 2 simulators did not complete (we stopped it after approximately 7 minutes), which we surmise to be caused by excessive memory requirements. Time Warp requires substantial memory overheads – namely, the state must be saved frequently, to allow rollbacks. For the same reason, we have only been able to report limited results.
for the worst-case tests. We can see that in all worst-case tests (which completed successfully) the performance is significantly worse than in their best-case counterpart.

5.3 Summary

In summary, neither Chandy-Misra nor Time Warp is a suitable algorithm for the purposes of simulating peer-to-peer networks with an end-to-end underlay network model. Since simulations of peer-to-peer networks will typically be dominated by inter-node communication, distributed simulation in general is inefficient. The Chandy-Misra algorithm is most promising, but communication costs still render it ineffective. Time Warp is generally ineffective, due to its high memory requirements.
Conclusion

We have investigated the use of distributed simulation for testing and analysing peer-to-peer network protocols. To aid our investigations, we have developed a distributed overlay network simulator, \textit{dnet}, with a traditional uniprocessor sequential event-scheduler, and two different distributed event-schedulers – namely Chandy and Misra’s null-driven deadlock avoidance algorithm, and Jefferson’s Time Warp algorithm. We have shown that distributed simulation is mismatched to end-to-end packet-level simulation of peer-to-peer networks. This is due to the fact that there is a large amount of inter-simulator communication for even a small amount of computation.

We can safely say that distributed simulation of peer-to-peer networks with an end-to-end model is not viable when using a cluster of low-end computers on a standard 100Mb Ethernet network. However, we believe that with a reduced cost of communication, \textit{dnet}'s performance would drastically improve. We summarise in the Section 6.1 some avenues for future work.

6.1 Future Work

We are confident there is a mismatch between distributed simulation and simulation of overlay networks with an end-to-end model. If one were still inclined to continue work in this field, this section outlines some of the work necessary for advancement.

6.1.1 Improvement of \textit{dnet}

There are numerous improvements that can be made to \textit{dnet}. We will cover here the most noteworthy ones.
Global Virtual Time calculation

The GVT algorithm for \textit{dnet} should be replaced. The currently implemented algorithm is prone to erroneous behaviour, as mentioned in Section 4.2.4. A promising new algorithm, called Seven-O’Clock [3], would be a good candidate for examination.

Implementation of non-FIFO Transport for Time Warp

The only transport developed for \textit{dnet} is a simple wrapper around MPI’s communication facilities, which are strictly ordered and reliable. Given that Time Warp does not require ordered communications, nor necessarily reliable communications, using MPI for communication means we may not have optimal communication. Given that the cost of communication is the major impediment to \textit{dnet}’s performance, it is important that we reduce communication costs as much as possible.

We suggest investigation of the use of UDP or RDP. These implementations should be able to stop the retransmission of stale messages – messages sent as a result of events that have been rolled back and canceled.

Multi-threaded Simulators

At present, \textit{dnet} instances are single-threaded, meaning only one event can be executed at a time within one process. It is possible to execute some events in parallel (for example, all of those events with the same timestamp). Distributing the workload does this already, but this introduces significant overhead in terms of communication. Executing events in parallel could have potential gains for both uniprocessor and distributed schedulers.

Intelligent Memory-Management

The current memory-management in \textit{dnet} is naïve – when an event is created, a block of memory is allocated on the heap, and when the event is destroyed, the memory is freed. Dynamic memory allocation and deallocation are costly procedures. In simulation, it will frequently occur that two blocks of memory of the same size will be requested. If an event has recently been destroyed, then the freeing of its memory can be delayed. If, within the time that the memory is still in use by the simulator, another event is created which requires the same amount of memory as that recently destroyed event, the memory can be reused.
Since we are concerned with memory requirements for large simulations, such optimisations are worthy of investigation.

6.1.2 Further Experiments

Investigation of *dnet*'s Performance on High-Performance Hardware

The cost of inter-simulator communication is the major factor prohibiting *dnet* being useable in low-end clusters. We would like to investigate *dnet*'s performance on higher performance hardware. In particular, we would like to examine the benefits of using a high-speed interconnect such as Myrinet [26]. Standard 100Mbit Ethernet has a roughly 100-200 µs round-trip latency [33], whereas 10G-Myrinet has a reported round-trip latency of only 2 µs. These values depend on the size of frames sent on the physical medium, but we can still get an idea of how much the latency can be reduced.

As well as using Myrinet, one should investigate how *dnet* performs on Symmetric Multi-Processing (SMP) computers. Since *dnet* uses MPI for its communication, the mode of communication is handled transparently. That is, *dnet* can transparently run in either a cluster, or in an SMP environment.

6.1.3 Network Partitioning

Modern peer-to-peer network protocols typically try to connect nodes which have a low round-trip latency. In a simulation of such a protocol, a large portion of simulated communication would occur between nodes that have the least virtual round-trip latency.

To reduce the amount of inter-simulator communication, and consequently improve simulator performance, we can partition the peer-to-peer network prior to simulation, such that nodes who are close together (in terms of latency) are grouped together on the same simulator. When the simulated network stabilises, and optimal connections are made, a reduced amount of inter-simulator will result.

We do not know of anyone who has done this before, so it would interesting to see how much impact it would have on simulation performance.
Bibliography


APPENDIX A

Original Honours Proposal

Title: Distributed Simulation of Peer-to-Peer Networks

Author: Andrew Wilkins

Supervisor: Dr. Chris McDonald

Background

Peer-to-peer networks are a form of overlay network in which each node in the network acts as both a server and a client, communicating directly with each other without the need for any external dedicated infrastructure. Peer-to-peer networks are not particularly new; variations of the architecture have existed in Usenet (UUCP, NNTP) and DNS for many years now [28]. It has only been in recent years that peer-to-peer networks have become popular in the mainstream in the form of file-sharing networks, dealing predominantly in music and movies.

Napster was once the most popular music file-sharing peer-to-peer network; today, popular networks employ a variety of systems such as Gnutella [15] and BitTorrent[10]. One reason peer-to-peer networks are very popular for sharing files is because they allow users to take content distribution into their own hands, making use of otherwise wasted resources. Popular file-sharing networks attract from hundreds of thousands to over one million users at one time [20], and allow users to select from and download millions of files.

Implementing and testing new concepts in real-world peer-to-peer overlay networks is almost impossible [34]; the large number of participating nodes, and the lack of centralised infrastructure make it difficult to coordinate software updates and to gather and monitor statistics. Simulation allows us to remove these difficulties by taking complete control, and implement new ideas or test existing ones in an inexpensive, tractable manner. Distributed and parallel simulation becomes necessary as processing and memory requirements increase; it is infeasible
to simulate a properly modeled peer-to-peer network of hundreds of thousands of nodes on a single desktop machine. Various peer-to-peer network simulators exist, some distributed [16] and some not [14].

Aim

The aim of this project is to develop and explore the use of a distributed simulator for implementing and testing peer-to-peer network protocols, with a focus on routing and content location. The simulator will be used to analyse and compare two or more existing peer-to-peer routing schemes, and to make comparisons with previous attempts to simulate peer-to-peer networks. Ultimately, I aim to produce a distributed simulator that will be useful for researchers in implementing and testing novel protocols.

I aim to develop the simulator with two main priorities:

- Efficiency. The simulator should scale to hundreds of thousands of simulated nodes, whilst allowing for accurately modeled peer-to-peer networks. This will likely mean taking a high-level approach to modelling the underlying network, as opposed to packet-level.

- Ease of use. To make protocol implementation easier (and hence less error-prone), I will endeavour to implement the simulator such that it will compile and run protocol modules written in standard C/C++, which will communicate with the simulator via a well-defined interface, as in cnet [22].

My primary goal is the development of the distributed simulator. The application of the simulator, the simulation of peer-to-peer routing protocols, is secondary. If time permits, or if my motivation changes, I may seek an additional or alternative application for the simulator, such as routing in a large Internet-like network.
# Method

<table>
<thead>
<tr>
<th>Months</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>March–April:</td>
<td>Research peer-to-peer network protocols.</td>
</tr>
<tr>
<td></td>
<td>Research distributed and parallel simulation techniques.</td>
</tr>
<tr>
<td>March–May:</td>
<td>Research/develop network models.</td>
</tr>
<tr>
<td></td>
<td>Implement single-processor simulator.</td>
</tr>
<tr>
<td></td>
<td>Implement protocols.</td>
</tr>
<tr>
<td></td>
<td>Analyse protocol efficiency (for later comparison).</td>
</tr>
<tr>
<td>May–August:</td>
<td>Refine network model.</td>
</tr>
<tr>
<td></td>
<td>Implement distributed simulator.</td>
</tr>
<tr>
<td></td>
<td>Modify protocols for distributed simulator (if necessary).</td>
</tr>
<tr>
<td></td>
<td>Analyse efficiency and compare against single-processor simulator.</td>
</tr>
<tr>
<td>August–September:</td>
<td>Collate documentation, analyses, and produce dissertation.</td>
</tr>
</tbody>
</table>
Software and Hardware Requirements

The hardware requirements of this project are variable; 10–20 computers of the following specifications should be sufficient:

- CPU: 2GHz+
- RAM: 512MB
- 100Mb–1Gb network card

The accuracy of simulations will be partially reliant on the number of nodes in a simulated network; the higher the processing power available, the greater the number of nodes can be simulated. The requirements specified above are merely expected (minimum) requirements. Sufficient storage space for analysis data will also be necessary: it is expected that multiple gigabytes of storage space will be required for data logging.

Code will be written for running on Linux (hopefully POSIX compatible), with possibly some test scripts written in Python. LAM-MPI will be used for inter-simulator communication. Thus, the only software requirements I foresee are:

- GNU/Linux
- GCC
- Python
- LAM-MPI
APPENDIX B

Obtaining, Compiling and Running dnet

This section details how to obtain, compile, and run dnet. dnet is a distributed overlay network simulator, the result of 5-6 months of work. It consists of roughly 8000 lines of code, spanning three event schedulers: sequential, Chandy-Misra deadlock-avoidance, and Time Warp.

B.1 Obtaining dnet’s source-code

dnet is licensed under the GPL, a free open-source software license. At the time of writing of this dissertation, dnet can be obtained from the web at the following URL:

http://www.csse.uwa.edu.au/~wilkia07/dnet/

As of the end of 2005, this will no longer be accessible. One may then obtain the source by sending a request to Andrew Wilkins at axwalk@gmail.com.

B.2 Compiling dnet

dnet has the following dependencies:

- Boost C++ Library (http://www.boost.org/)
- Python 2.4 (http://www.python.org/) – optional
- GT-ITM (http://www.cc.gatech.edu/projects/gtitm/) – optional
- LAM-MPI (http://www.lam-mpi.org/)
- GNU Scientific Library (http://www.gnu.org/software/gsl/) – optional
dnet has been successfully compiled with GCC 3.4.3 and 4.0.2. To compile dnet, enter the top level directory, and perform the following:

<table>
<thead>
<tr>
<th>Scheduler</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>make dnetcore-seq</td>
</tr>
<tr>
<td>Chandy-Misra</td>
<td>make dnetcore-cm</td>
</tr>
<tr>
<td>Time Warp</td>
<td>make dnetcore-tw</td>
</tr>
</tbody>
</table>

You may need to alter locations or names of libraries. In particular, boost libraries have different names in different Linux distributions. In our primary development environment, `-lboost_signals-gcc` is used to link against the Boost Signals library. On Fedora Core, this should be changed (in `src/Makefile`) to `-lboost_signals`.

Optimisations and debugging parameters to GCC are specified in Makefile.base, in the top-level directory of dnet.

B.3 Running dnet

To run dnet, you require three things: a dnet binary for the scheduler you wish to use, a user-application, and a underlay network topology, in SGB format.

To compile a user-module, you must execute `dnet-compile <module input file names> <output filename>`. This will take in the specified input file names, compile them, and output a dynamic shared object with the specified output filename.

Generating an underlay network topology is not covered here – see the GT-ITM documentation, or use `test.gb` which comes with dnet. Table B.3 specifies how to run the dnet Python script in the top-level directory.
usage: [options] module_file #peers graph_file

options:
-h, --help                show this help message and exit
-s SCHEDULER, --scheduler=SCHEDULER
                        select which scheduler to use (sequential, timewarp
                        (default), chandy_misra)
-v, --verbose             print out extra status information
--degree-ceiling          Ceiling on the number of connections an ISP has
                        (default: 2)
--wrapper=WRAPPER          specify a program to wrap the simulator, such as a
                        debugger like gdb, or valgrind.
--log                     turn on logging to a file
--log-prefix=LOG_PREFIX   specify the prefix for log files; the sequential
                        simulator will output to 'prefix', while distributed
                        simulators will output to 'prefix.$LAMRANK'
-t, --stop-time           The simulated time at which the process should stop.
                        (inf (default), or an integer > 0)

Table B.1: dnet Python script usage