Proportional Time Emulation and Simulation of ATM Networks

by

Sean B House

B.S. (Mathematics), University of Kansas, Lawrence, Kansas, 1998
B.S. (Computer Engineering), University of Kansas, Lawrence, Kansas, 1998

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

_____

Professor in Charge

_____

_____

Committee Members

_____

Date Thesis Accepted

I just have one rule: Everybody fights, no one quits.

Acknowledgments

I would like to express my sincere and utmost gratitude to Dr. Douglas Niehaus, my advisor and committee chairman, for his invaluable guidance, advice and inspiration throughout this research and my entire collegiate experience. I would like to thank Dr. Victor Frost for serving on my committee and for his direction and continuous encouragement, and Dr. David Petr for his instruction and insight on many subjects, both in the classroom and throughout this research. Thanks also to Dr. Jeremiah James for agreeing to serve on my committee and to Sprint Corporation for their support of this research.

I would like to thank the members of Team Niehaus that I have had an opportunity to work with at one point or another; Matt Peters, Shyam Pather, Robert Hill, Balaji Srinivasan, Shyam Murthy, Sachin Sheth, Sandeep Bhat, Yulia Wijata, Aaron Hoyt, Chanaka Liyanaarachchi, Sridharn Nimmagadda, Will Dinkel, Pramodh Mallipatna, Kamalesh Kalarickal, Gowri Dhandapani, Phongsak Prasithsangaree, Alejandro Parra-Briones, Anitha Rajesh, Aparna Ramkumar, Bhavani Shanmugam, Kevin Hunter, Scott Gelb, Xavier Stevens, Nurall Delon and anybody else that I have inadvertently forgotten. Everybody at Team Niehaus has made this an exceptional working environment and provided valued assistance, encouragement and friendship.

I would like to especially thank Shyam Murthy, whose preliminary work in Available Bit Rate service was a foundation on which I have based much of my work. I would also like to acknowledge Will Dinkel and Jason Chaffe for humoring me over the last couple of years and providing me much needed distractions, as well as assistance. Many thanks also to Roel Jonkman who always seemed to have the answers when I didn't.

I owe very special thanks to Ricardo Sanchez, from whose work much of this research was based and with whom I have worked closely to further the development and deployment of his creation, the virtual ATM software switch. I also owe a enormous debt of gratitude to Ming Chong who has been a great source of knowledge concerning Georgia Tech Time Warp and whose countless hours of dedication have provided the Time Warp results herein.

Perhaps the person I am indebted to the most is my girlfriend of seven years, Adriane Musuneggi, who continues to put up with me and has provided a great deal of encouragement and understanding throughout this process. Of course, my family has invariably been a profound source of support and inspiration, especially my parents, who have always instilled in me that I can accomplish anything that I set my mind to. Mom and Dad, while a simple thanks couldn't possibly recompense the support, guidance, and encouragement you've given me, nor the many sacrifices that you have made to provide me with the opportunities that you have, I'd like to say thanks anyway.

Abstract

The performance evaluation of Asynchronous Transfer Mode (ATM) networks has received considerable attention from the networking research community. Most of the relevant studies have been conducted using conventional methods such as discrete event simulation software. This research presents an application of real-time and embedded system techniques to create synchronized distributed proportional rate simulations of ATM networks which execute substantially faster than equivalent sequential discrete event simulations and show scaling tendencies superior to those of popular parallel discrete event simulation implementations. In this work, KU Real-Time modifications to the Linux operating system (KURT-Linux) were used to control proportional rate distributed ATM network simulations. The simulation results from this system are compared with simulation results for a specific set of experiments under alternative modeling methodologies and the speedup achieved using the Proportional Time Emulation and Simulation (ProTEuS) platform are discussed.

Contents

List of Tables

List of Figures

List of Programs

Chapter 1

Introduction

Asynchronous Transfer Mode (ATM) is now widely recognized as an important networking technology. By utilizing short (53 byte) fixed-length packets, called cells, ATM sufficiently reduces the delay variance making it an appropriate network for an array of integrated traffic types including voice, video and data. Because these various traffic types typically carry with them fluctuating requirements for bandwidth, delay and loss, unlike conventional IP networks, ATM provides quality of service (QoS) guarantees that the network must provide for and manage. ATM provides high-speed networking at very low error rates and allows the coexistence of dissimilar traffic types in a manner that is uniform, efficient and easily manageable.

It is our desire, as well as that of many others, to evaluate the performance of ATM networks of significant size and complexity under varying conditions and architectures. There are, of course, several approaches to solve this problem. One solution is the use of conventional techniques, such as sequential discrete event simulation (S-DES). Unfortunately, these simulations do not scale well in two important respects; the size of the network and the length of simulated time. Another method to model communication networks is through the use of parallel discrete event simulation (PDES); in particular, the Time Warp paradigm, which is an optimistic parallelism technique[29]. However, it is well-known that Time Warp is subject to poor performance when the computation is fine-grained, as it is in an ATM network simulation with small fixed-length cells. That is to say, in such applications, Time Warp is not able to fully exploit

the advantages to which it is accustomed, due to the high overhead to computation ratio.

The real problem is how to efficiently simulate large networks for long periods of virtual (simulated) time. It is our contention, and that of many others, that faster single processors are not adequate to offset the scaling of the simulation. Further, current methods for parallelizing such simulations are not entirely satisfactory for some simulated systems, including the particular networks which we want to study. Our approach to improving performance is an innovative application of real-time and embedded techniques producing a system that supports parallel simulations executing in proportional time, named ProTEuS: Proportional Time Emulation and Simulation.

1.1 Motivation

There have been several efforts to solve this and related problems because of the desire to evaluate systems, particularly large ones, without having to actually construct them. Our motivating interest lies in the evaluation of large communication networks, for which there are two principal simulation methods; sequential discrete event simulation and parallel discrete event simulation. Sequential discrete event simulation tools include BONeS, OPNET and Extend, just to name a few[10, 36, 27]. The foremost parallel discrete event simulation paradigm is the Time Warp mechanism, which has been enacted in various forms, but notably in Georgia Tech Time Warp (GTW), from the Georgia Institute of Technology[17].

We feel that there are four primary criteria for evaluating the performance of a simulation:

1. *Execution time*: The total wall-clock running time of the simulation.
2. *Verity*: The faithfulness of the simulation model to the system being simulated.
3. *Scalability*: The degree to which the simulation scales in both the size of simulated system and the length of simulated time.

4. *Accessibility*: The availability of the simulation environment and its associated cost.

Sequential discrete event simulations do not scale well with respect to the size of the network, nor with respect to the length of simulated time. Networks of significant size become increasingly difficult to *even model* in sequential discrete event systems and their size significantly affects the run-time of a simulation. They also carry with them non-trivial cost implications and are generally not open source, both of which limit their availability.

Time Warp techniques, which use optimistic concurrency control to parallelize simulations, are quite promising with regard to scalability. However, systems with significant feedback cause interactions among distributed components that can result in significant overhead due to frequent *rollback*, where the simulation regresses in time to dispel a conflict and then resumes. To take advantage of concurrency and accommodate the memory requirements imposed by its high-overhead optimistic philosophy, Time Warp systems generally require a high-end multiprocessor machine such as an UltraSPARC or SGI, which are also quite expensive and limit their availability on a practical level.

Moreover, neither discrete event simulators, nor Time Warp, are guaranteed to be particularly faithful to the real system being simulated. That is to say that they are, of necessity, an abstraction that requires the implementation of representations of system code in the simulation environment.

1.2 Proportional Time Emulation and Simulation (ProTEuS)

ProTEuS is a network simulation solution designed to combat many of the potential shortcomings of some of these alternative network simulation mechanisms. ProTEuS uses real-time, distributed system and embedded system techniques to create synchronized distributed proportional time simulations. In this work, the simulations of interest are of communication networks in general, and ATM networks specifically.

Since the simulated system is distributed across any number of physical machines,

it is many times faster than conventional sequential discrete event simulation experiments and shows scalability tendencies superior to those of Time Warp implementations. ProTEuS provides performance over a Network of Workstations (NOW) comparing favorably to that of GTW on a shared-memory multiprocessor. With the addition of virtual network device support, ProTEuS can map a simulated network onto a set of physical network capable hosts almost arbitrarily. This renders mapping a ProTEuS simulation onto a set of physical machines strongly analogous to mapping a GTW simulation onto the physical processors of a shared-memory multiprocessor.

Furthermore, we use the *real code*, both system and application, that networks and systems could, and often do, use. For example, ProTEuS ATM simulations use a real operating system protocol stack and the ATM signaling support is the same as that used in an off-board signaling architecture (Q.Port)[18, 19, 6]. Because it uses real system networking code, it does not require the implementation of system code abstractions into a software simulator as is required in other discrete event simulators. As a point of clarification, we are not trying to imply that ProTEuS abstracts nothing, but rather that ProTEuS significantly decreases the scale and scope of the abstraction required to create our ATM network simulations. This reduction in the extent of abstraction provides an environment that is significantly more faithful to the system being simulated.

Moreover, ProTEuS uses commercial off-the-shelf PC hardware running Linux at a modest cost to run proportional time simulations of ATM networks. Because ProTEuS runs on conventional PC hardware and software, when these resources are not being utilized for simulations, they can serve as everyday desktop PCs or support other research endeavors. Likewise, all of the necessary software is free and open source, including the real-time platform. ProTEuS maximizes accessibility through free software access and modest hardware cost.

The next chapter discusses some of the related work in the area of discrete event simulation, including the alternative solutions already mentioned. Chapter 3 is a detailed look at the design and implementation of the ProTEuS system in general, and for ATM simulation in particular. Chapter 4 presents an evaluation of ProTEuS as a viable

alternative for communication network modeling and evaluation. Finally, Chapter 5 provides some conclusions reached and some possible extensions to the system.

Chapter 2

Related Work

A considerable amount of research has been performed in the area of discrete event simulation, especially in the areas of logic circuits and communication networks, where many efforts have been made to produce a scalable, faithful, extendable and easy to use simulation platform. Many fundamentally different approaches to solving the same problem have been explored by these efforts, each resulting in varying degrees of overall success in each target area due to the vastly different properties of the simulation methods and driving applications.

This chapter discusses several other efforts at solving this problem. Each solution generally falls into one of two categories; sequential execution for single-processor workstations, or distributed execution aimed at shared- and distributed-memory multiprocessors. Although there are exceptions, in most cases of parallel execution, the systems employ optimistic synchronization in an effort to let the simulation progress as quickly as possible while detecting and correcting temporal violations, as opposed to the conservative synchronization protocol utilized by ProTEuS, which avoids these issues and their associated overheads by simply not allowing temporal violations to occur.

2.1 Sequential Discrete Event Simulation

Sequential discrete event simulators tend to be a simple and common solution for the simulation of small systems. In a typical implementation, the simulation has a single event queue (an event generally consists of some data, a timestamp and a corresponding action to perform) sorted in virtual timestamp order that is serviced by an event dispatcher. Events are added throughout the simulation and the dispatcher stops servicing events from the event queue when either the queue becomes empty, or when the virtual time clock reaches a specified stop time.

Unfortunately, because they run on a single processor, sequential discrete event simulations do not scale well as the size of the simulation increases, and it is generally accepted that faster single processors are not sufficient to offset the increase in computational demand as the simulation scales. Further, while the graphical nature of many of these systems is generally appealing, specifying a simulation is not made trivial by its nature. Nevertheless, sequential simulators certainly have their niche in the simulation space and are more than adequate, and even superior, in a fair share of circumstances.

2.1.1 BONeS

BONeS, the Block Oriented Network Simulator, is a sequential simulation platform produced by Cadence Design Systems, Inc[10]. BONeS employs a graphical user interface (GUI) that allows users to build networks using libraries of network elements in a block-oriented manner. The system architecture is an abstract model that treats the system as a collection of shared resources where users build hierarchical blocks representing network elements. BONeS also allows users to create new elements and optionally plug code into them, for example, to add Enhanced Proportional Rate Control Algorithm (EPRCA) support for ATM Available Bit Rate (ABR) service to an existing ATM switch model[48].

Cadence Design Systems, Inc. has recently ceased development of BONeS Modeler and has subsequently reached agreements with Mil3, Inc., the creators of OPNET

Modeler, to provide support for BONeS customers and assist in moving them over to OPNET Modeler.

Some information on BONeS can be found at *http://www.cadence.com*.

OPNET related information can be found at *http://www.mil3.com*.

2.1.2 OPNET

OPNET, the Optimized Network Engineering Tools, is a sequential discrete event simulation platform developed by Mil3, Inc[36]. OPNET provides detailed network modeling capabilities and includes sophisticated tools for the collection, evaluation and visualization of simulation data and performance metrics. Like many others in its class, it uses a GUI for model specification in a hierarchical, block-oriented structure. OPNET also allows users to import third-party libraries and provides APIs to standard modeling languages. Like BONeS, OPNET makes it easy for users to create new simulation entities and insert sophisticated implementations in a familiar programming environment.

More information on OPNET can be found at *http://www.mil3.com*.

2.1.3 Extend

Extend, developed by Imagine That!, Inc., is another graphically-oriented software simulation platform[27]. Unlike other simulation platforms mentioned herein, Extend that has the unique ability to model both discrete event and continuous time systems. Extend supports high user interaction with the simulation through its GUI, even allowing the modification of parameters and the viewing of results while the simulation is running. Extend is also hierarchical and block-oriented, which lends itself to easy model organization and component re-use among simulations. Like many other GUI based simulators, it also has excellent support for collection and visualization of simulation results.

More information on Extend can be found at *http://www.imaginethatinc.com*.

2.1.4 Ns/VINT

Ns is a sequential discrete event simulator developed at the University of California at Berkeley that is targeted specifically at network simulation, and is used most often to simulate network transport protocols and queueing disciplines[8]. The Virtual InterNet Testbed (VINT) is a program at Lawrence Berkeley National Labs, Cal-Berkeley and the University of Southern California aimed at producing improved simulation tools and techniques for the research and deployment of Internet protocols that has adopted Ns as its simulation platform and made many advances to it in the process.

Ns employs a split-level programming model; the simulation kernel, which is the core set of simulation primitives that requires the highest performance, is implemented in C++, and the definition, configuration and control of the simulation is implemented in OTcl, which is an object-oriented variation of the Tool Command Language (Tcl). Because of the frequency of their execution and stagnant nature, simulation primitives are well suited for a compiled implementation, while highly fluid, and often iterative simulation aspects such as setup and configuration, are much better served by a flexible and interactive scripting language, such as Tcl.

Ns includes a wide range of network protocols and queueing disciplines, including many flavors of the Transmission Control Protocol (TCP), the User Datagram Protocol (UDP), multicast protocols, Random Early Detection (RED), Explicit Congestion Notification (ECN), etc., and more are being contributed on a regular basis by the more than 200 estimated institutions world-wide using Ns. Further, Ns has an emulation mode, where stubs translate network packet contents and tunnel traffic between the simulation and a real network[20].

Rather than parallelizing execution to improve simulation performance, Ns chooses to tune the simulation implementation and provide multiple layers of abstraction. Of course, the potential cost of abstracting out detail is the accuracy of the simulation, but that cost often depends on the particular scenario being simulated. The position that Ns assumes is that abstraction can often result in significantly improved simulation performance with negligible effects on accuracy. It is also worth mentioning, however, that while native Ns is sequential, the Parallel and Distributed Systems (PADS)

group at Georgia Tech University has done some work to parallelize some of the Ns modules.

More information on Ns/VINT can be found at *http://www.isi.edu/nsnam/vint/*.

More information on PADS at Georgia Tech can be found at *http://www.cc.gatech.edu/computing/pads/*.

2.2 Parallel Discrete Event Simulation

Parallel discrete event simulation is one of several techniques to combat the scaling limitations inherent in a single-processor system[21, 42]. By distributing the simulation across multiple processors, one can very often achieve significant *speedup* in execution time. The magnitude of that speedup, however, depends on many factors, including the properties of simulated system itself, the number of processors over which the simulation is distributed, the inter-processor communication overhead and the inter-processor synchronization protocol.

One of the most fundamental rules of a discrete event simulation is that events *must be executed in order of non-decreasing virtual time*. In a sequential system, this can be enforced by a single event queue sorted in virtual timestamp order. In a parallel simulation, it is complicated by the fact that execution on distributed processors must be synchronized to prevent temporal violations. There are two fundamental methods by which to synchronize a parallel computation; conservative and optimistic.

In a conservative protocol, first proposed by Chandy and Misra, events are executed strictly in virtual timestamp order, without exception[11]. In order to attain perfect knowledge of pending events, each node in the system must have knowledge of every other node that can potentially send it messages. These messages are typically stored in FIFO queues; one for each point-to-point connection between two nodes. The queues are serviced one event at a time, always selecting the queue whose head message has the lowest logical virtual timestamp. This method prevents causality violations from occurring and is a very lightweight protocol, but may unnecessarily impede a simulation from progressing, which is a potential performance impact. Further, when the

selected queue is empty*, the protocol *must block*, waiting for a message from the associated node because the absence of a message could be due to any number of causes and the protocol has no way of knowing what timestamp the next arriving packet will have. If a cycle of empty queues with minimal virtual timestamps occurs, deadlock arises. One way to prevent deadlock, first suggested by Chandy and Misra, is through the use of NULL messages, where nodes periodically send out virtual time updates, even when no real messages are available. Of course, many other solutions have been suggested as well, including the detection of and recovery from deadlock using methods such as token passing and termination detection[12, 39].

In an optimistic protocol, made popular by Jefferson's Time Warp principle, events are executed at the time of their arrival[29]. Optimistic protocols take advantage of the fact that the most likely order of event arrival corresponds to non-decreasing order in virtual time. When events do arrive out of order, often called *straggler* events, the synchronization mechanism has to reconcile the violation, often through the *rollback* of the simulation to some previous known-good state. To support the ability to restore the simulation to a previous state, nodes keep lists of all messages sent and received over a virtual time period. When a straggler arrives, the protocol "rewinds" its execution to a time just previous to the straggler event's virtual time by walking these queues. In doing so, the node "undoes" all actions it performed erroneously and sends *anti-messages* to any nodes to which it sent an erroneous event previous to the arrival of the straggler. Upon completion, the node starts progressing forward in simulated time again. Of course, this mechanism involves potentially sophisticated and costly state-saving and conflict resolution implementations, which can have a significant performance impact in systems on which it is invoked frequently. However, due to its obvious performance impact, a great deal of research has been focused at optimizing these mechanisms. For example, not all straggler messages actually result in a change to the system state, so stragglers that do not alter the state of a node do not necessarily need to result in a rollback. Moreover, if a node that is a target of an anti-message has

*When a queue is empty, its associated virtual timestamp is the virtual timestamp of the last packet received on that queue, so an empty queue can be the selected queue even when there are non-empty queues present.

already processed the offending event, the result is *cascading* rollbacks that can propagate around the system and wreak havoc. Therefore, *lazy cancellation*, as opposed to *aggressive cancellation*, waits until the the computation re-completes and only sends an anti-message if the contents of the new message actually differs from the previous one, avoiding unnecessary cascading rollbacks.

In any case, most researchers agree that parallel simulation techniques are absolutely necessary to offset simulation scaling and a great deal of research effort has been placed in their development.

2.2.1 GTW

Georgia Tech Time Warp (GTW), an optimistic parallel discrete event simulator from the Parallel and Distributed Simulation (PADS) group at the Georgia Institute of Technology, is perhaps the most popular rendition of Jefferson's Time Warp principle to date[17, 23]. The primary design goal of GTW was to combat Time Warp's well-known performance issues when simulating fine-grained applications due to the overheads associated with optimistic synchronization. It is well-known that state-saving overhead and rollback costs dominate the performance of a Time Warp simulation. GTW was explicitly designed to minimize these overheads and support efficient execution of fine-grained discrete event simulations. While it is primarily intended for cache-coherent shared-memory multiprocessors, GTW has also been employed in simulations over a Network of Workstations (NOW).

A GTW simulation consists of a number of Logical Processes (LPs), generally one per simulation entity, that communicate via timestamped messages. Execution is strictly event-driven, such that all execution occurs in response to the reception of a message and the initialization of each LP is responsible for generating the first event(s). LPs also must inform the simulation kernel what portion of their state should be saved by the state-saving mechanism. In *automatic* state-saving, the state information is saved by GTW transparently to the user before each message is processed. In *incremental* state-saving, the act of saving state is explicitly initiated by the user when desired. Because this is a known performance bottleneck, a great deal of subsequent research has been

done to further decrease the state-saving overhead for many applications.

All Time Warp implementations, GTW included, maintain Global Virtual Time (GVT), which is essentially the minimum Local Virtual Time (LVT) of all LPs. The significance of GVT is that it presents a limit in past time over which no rollback can traverse, and therefore all state and data associated with events occurring before GVT can be reclaimed by the system. The calculation of GVT can be initiated by any processor and involves each processor finding the minimum timestamp of all unprocessed events and placing it into a global array. The last processor to do so calculates the minimum and updates GVT.

Contrary to Jefferson's initial intentions, each processor in a GTW simulation maintains a single pending event queue for all LPs on that processor, sorted in timestamp order, which allows GTW to find the unprocessed event with the smallest timestamp through a single dequeue operation. While this simplifies event processing and GVT computation, it makes dynamic LP load balancing much more difficult*. Each LP maintains its own queue of processed events, which includes the saved state information and messages sent as a result of the event processing (if any) for use in case of rollback. A process called *fossil collection* periodically cleans entries out this queue when GVT advances past the timestamp of the events.

Ironically enough, most of the issues with Time Warp implementations occur as a direct result of their optimistic nature. For instance, LPs that progress too fast in virtual time tend to run out of memory because their processed event queue grows larger as it progresses further past GVT. In an effort to reclaim memory, those processors will often initiate a GVT calculation to advance GVT and trigger fossil collection. Any memory reclaimed is typically exhausted quickly and a GVT calculation is initiated again. This frequent GVT calculation, known as GVT thrashing, keeps the other LPs from progressing, which perpetuates the problem. To combat this issue and others, work has actually been done to *limit* the optimism of LPs by blocking their advancement through several mechanisms, in sort of a hybrid optimistic/conservative implementation[44]. ProTEuS, of course, eliminates these concerns by implementing a conservative syn-

*Dynamic LP load balancing is *not* supported by GTW at this time.

chronization protocol where temporal violations are simply not allowed to occur, circumventing the need for optimizations associated with the high-overhead of optimism.

The Telecommunications Description language (TeD) from the PADS group at Georgia Tech University is a modeling framework for communication network simulation that utilizes a parallel simulation kernel, though not necessarily GTW, to perform distributed simulations of communication networks, including ATM[7].

More information on GTW can be found at *http://www.cc.gatech.edu/computing/pads/tech-parallel-gtw.html*.

2.2.2 WARPED

WARPED is an optimistic parallel discrete event simulation kernel based on Jefferson's Time Warp paradigm developed at the University of Cincinnati[33, 46]. It is an object-oriented Time Warp library written in C++ that runs on shared- and distributed-memory multiprocessors and has been ported to several operating systems and hardware variations. Unlike many of the other systems mentioned herein, the initial goal of WARPED was not to create a simulator for one or two specific driving applications, but rather to create a simple, extendable and portable platform specifically for the research of Time Warp mechanisms.

WARPED applications provide the simulation kernel class definitions of; (1) all Logical Processes (LPs), which are simulation entities that send and receive messages, (2) the state associated with those LPs, which is needed for state-saving purposes imposed by the optimistic synchronization, and (3) their associated events, which are communicated via inter-LP messaging services. Because of its object-oriented architecture, WARPED can mask nearly all details of the Time Warp implementation and synchronization protocol from the user by default, but users can change this behavior whenever necessary through the object-oriented mechanisms of inheritance and overload.

One area in which WARPED deviates from Jefferson's original Time Warp presentation is that it allows LPs to be grouped into clusters. The advantage of such a scheme is

that LPs in the same cluster can communicate directly, bypassing the message passing system*, which is much faster than communicating through it. Therefore, frequently communicating LPs benefit from being grouped into the same cluster, perhaps at the expense of load balancing in the simulation, which introduces one of many well-known trade-offs in parallel distributed computing. Moreover, it is the responsibility of a cluster to schedule the execution of the LPs within it, and although there may be some advantage to doing so, LPs within a cluster are independent and are not coerced into synchronizing with each other.

Some of the existing WARPED applications include KUE, a queueing model library, which is in turn utilized by SMMP, a shared-memory multiprocessor simulation application, and RAID, an IBM SCSI disk array simulation application. Much more sophisticated WARPED applications include TyVIS, a VHDL simulator, and USSF, a large scale simulation framework for communication network simulations[49].

As alluded to earlier, WARPED was created to facilitate research of the Time Warp paradigm itself and a lot of work has been done investigating some of the well-known Time Warp performance issues, such as state-saving and rollback[47]. This work includes lazy and dynamic cancellation, which address rollback and cascading rollback caused by anti-messages, and periodic and dynamic check-pointing, which address state-saving frequency and the associated overhead. Many of these schemes and others not mentioned here are available in WARPED as compile-time and run-time options.

Another interesting optimization investigated in WARPED research is dynamic message aggregation, which addresses the frequency of communication and its associated overhead[13]. WARPED experiments with aggregating the messages generated from a single Logical Process destined for the same recipient Logical Process over time using fixed and sliding virtual time windows; all messages from node A destined for node B within a virtual time window X are aggregated into a single message to reduce the overhead of communication. Serving the same purpose, but implemented in a seemingly orthogonal manner, ProTEuS aggregates its inter-LP messages, but does so across LPs, not time; all messages from nodes on host A destined for nodes on host

*WARPED uses the Message Passing Interface (MPI) for inter-cluster communication.

B during epoch X are aggregated into a single message.

More information on WARPED can be found at *http://www.ece.uc.edu/~paw/warped*.

2.2.3 ParSEC

The Parallel Simulation Environment for Complex Systems (ParSEC) is a parallel discrete event simulator for shared- and distributed-memory multiprocessors developed at the University of California at Los Angeles[4]. ParSEC is based on Maise, which is a parallel simulation language previously developed at UCLA, and has the flexibility to operate in sequential or parallel mode and with conservative or optimistic synchronization in the latter[3]. It also includes visualization tools for specifying topologies, simulation parameters, partitioning, etc.

One of the driving motivations in the design of ParSEC was the ability to minimize the effort and time to migrate code between the simulation and working real-world prototypes. Because ParSEC is built around a thread-based message passing kernel for general purpose parallel programming (MPC) such translations are rendered seemingly trivial. For instance, some ParSEC wireless protocol implementations have been directly refined and inserted into the protocol stack of a network operating system.

One of the most intriguing features of ParSEC is its Ideal Synchronization Protocol (ISP) support, which can be used to measure the overhead of a synchronization protocol. To calculate the ISP for a particular simulation, ParSEC runs the simulation twice. The first simulation is synchronized using any of the available synchronization protocols and is necessary only to create a detailed trace of the simulation execution. In the subsequent simulation, ParSEC uses the trace from the first run to make synchronization decisions. This gives ParSEC omnipotence regarding the order of event execution and so the computation involved in conventional synchronization protocols is bypassed, providing an "ideal" synchronization protocol measurement where the decision regarding whether or not to process a message degenerates to essentially a no-op. This measurement can be used to provide a straight-forward comparison of candidate synchronization protocols against the optimal, and clearly segregates the simulation time from the overhead involved solely in the synchronization protocol.

Other flavors of parallel simulators that have been created from the ParSEC base include Mirisim, which is a parallel switch-level circuit simulator, and GlomoSim, which is a parallel simulator for wireless and mobile networks[5].

More information on ParSEC can be found at *http://may.cs.ucla.edu/projects/parsec/*.

2.2.4 ParaSol

The Parallel Simulation Object Library (ParaSol) is an optimistic Time Warp based parallel discrete event simulator for shared- and distributed-memory multiprocessors developed at Purdue University[34]. While most of the PDES systems investigated herein adopt the active-resource simulation model, where resources are represented by objects and customers by messages passed between those objects, ParaSol embraces the more natural active-transition paradigm, where resources are objects and customers are processes acting on those objects. Further, while ParaSol is primarily a parallel simulator, it can also be modified to run in sequential mode, if so desired.

ParaSol uses the single-program multiple-data (SPMD) model to run simulations over multiple processors. The same process (executable) runs on all processors participating in the simulation, with each traversing different paths through it. A main program contains one or more Logical Processes (LPs) associated with one or more objects, and potentially many active-transition threads (customers). For example, each cell in an ATM network simulation may be represented by an individual thread that "moves" around the simulation between switches and hosts (objects).

ParaSol is based on, and depends on many of the features of, a run-time threads library, primarily leveraging the Ariadne user-level thread package, also developed at Purdue. Ariadne provides several interesting thread mechanisms, such as thread migration between processors, user-defined thread schedulers and thread check-pointing and restoration. Of course, thread migration between processors is necessary for ParaSol to route active-transition threads to the processor hosting the object that the thread is acting upon; i.e. getting a packet to the right queue in a IP router simulation, where the queue object may be (is likely to be) hosted on a processor other than the one the packet thread is currently executing on. Thread check-pointing and restoration is

functionality required by the Time Warp optimistic synchronization and its associated state-saving. The user-defined scheduling ability that Ariadne provides is key to the performance of ParaSol, because it allows the scheduling of simulation threads to be *simulation aware*; that is, the scheduler has unique knowledge of the simulation state, allowing it to "optimize" scheduling in order to assist in avoiding optimistic PDES pitfalls, such as temporal violation and the resulting rollbacks. ParaSol schedules threads *in virtual timestamp order* to keep LPs from falling too far behind in virtual time, increasing the rollback probability, or from getting too far ahead in virtual time, increasing the risk of a phenomenon known as Global Virtual Time (GVT) thrashing, both of which degrade overall simulation performance.

There are two interesting observations to be made here. First and foremost, we see further evidence of the desire, or necessity, to limit the optimization of a Time Warp simulation in many circumstances to avoid the consequences of optimism run rampant. Secondly, we are beginning to see forms of specialized support for simulation platforms; this time in the form of a tailored-to-fit user-level threads package. However, remember that user-level processes containing user-level threads continue to be scheduled at the whim of the system scheduler, while ProTEuS provides *system-level* real-time scheduling support to improve the performance of its simulations.

One of the drawbacks to the active-transition paradigm is the increased system complexity. However the onus of this complexity is on ParSol itself, and is largely hidden from the user. In fact, many researchers believe that writing applications for an active-transition model is actually *easier* than writing applications for the active-resource model. Some of the active-transition complications include the added state-saving overhead created by having to save both objects and threads, as opposed to just objects. This is an area that requires optimization effort due to the well-known effects of state-saving on performance. Thread migration itself causes most of the complicating issues, such as thread stack location consistency across processors, concerns regarding thread stack size due to the potentially large number of threads, and implications on resource allocation (threads cannot allocate on the heap because the heap is not transferred during migration)[43].

It is further worth noting that through some of its unique characteristics and those of Ariadne, ParaSol has the ability to dynamically load balance a simulation among participating entities through the migration of entire LPs (both data and threads) across processors; a quite useful mechanism in tuning performance.

More information on ParaSol can be found at *http://www.cs.purdue.edu/research/PaCS/parasol.html*.

2.2.5 The Scalable Simulation Framework

The Scalable Simulation Framework (SSF) is a distributed simulation platform aimed at, but not limited to, the simulation of large-scale communication networks that was developed collaboratively by Dartmouth College, Rutgers University, and Cooperating Systems Corporation[16, 15]. DaSSF, the Dartmouth SSF simulation core implementation, is based on a previous parallel simulator developed at Dartmouth called Nops, the Northern Parallel Simulator, both of which synchronize parallel execution conservatively[41].

In essence, SSF is a self-contained object-oriented design pattern. It defines five generic base classes; *Entity*, *Process*, *InChannel*, *OutChannel* and *Event*. *Entities* are simulation components and *Processes* implement the behavior of these components. *InChannels* and *OutChannels* are incoming and outgoing communication endpoints, respectively, that can support unicast, multicast or bus-style communication. *Events* are messages that implement information exchange among *Entities*, via *InChannels* and *OutChannels*. These base classes provide a simple, generic and flexible foundation for constructing many varied simulation models.

In order to take maximal control over the execution of its simulations, DaSSF implements its own user-level thread support at the source code level. That is, DaSSF models are transformed into programs that do not use a conventional user-level threads package, but rather imitate multi-threaded execution in a single-threaded process. This gives DaSSF complete control over the scheduling of its simulation entities and the thread state-saving overhead, which helps it tune performance. However, as alluded to earlier, this continues to be support at the user-level, leaving DaSSF subject to

the impulse of the system, while ProTEuS utilizes real-time scheduling support at the system-level.

One of the most intriguing aspects of DaSSF is its synchronization protocol, which very closely resembles that of ProTEuS. DaSSF implements a conservative synchronization protocol where processors synchronize periodically. Therefore, the synchronization overhead is not directly dependent on the simulation model itself, but rather on the chosen frequency of synchronization. DaSSF uses simulated transmission delays to determine the size of this synchronization window*. On each processor, by setting the synchronization period to the minimum of all simulated transmission delays on links whose destination is on another processor, it can be assured that any message sent from this processor in synchronization period N will not affect the destination processor until at least synchronization period N + 1. Therefore, large transmission delays allow DaSSF to synchronize less frequently, lowering overhead and generally improving performance. Conservative synchronization in ProTEuS operates in a nearly identical manner, also utilizing simulated network delays to allow a window of synchronization. However, ProTEuS continues to checkpoint synchronization during every epoch, while allowing violations within the limits of the window.

Because DaSSF is simple and generic, it is commonly augmented by adding layers of domain-specific standard components, called packages. For instance, SSF.NET, which includes common networking components such as routers, hosts and network links. Through the SSF.DBMS package, DaSSF provides self-configuration support for very large simulation models through queries to a parameter database.

More information on SSF can be found at *http://www.ssfnet.org*.

2.2.6 Large Scale Distributed Real-Time Computation

This time-driven distributed computing model targeted at large scale distributed computations was developed at the University of Kansas and is the logical predecessor to ProTEuS[35]. The time-driven framework runs on commercial off-the-shelf hardware that is essentially treated as an embedded system, where guaranteed resources are al-

*DaSSF *requires* a non-zero delay on all channels between processors.

located to distributed real-time tasks. To facilitate low latency communication between nodes, it uses an ATM network, which provides real-time transport services and quality of service (QoS) guarantees. The entire system runs on free software, including the Linux operating system.

Computation proceeds in rounds, where each round consists of a *task execution* and a *margin*. The *task execution* further consists of sub-tasks; receiving and processing messages from other nodes, computation and sending messages out to destination nodes. In ProTEuS, the *round* is called an *epoch*, the *task execution* is called the *computation time*, and the *margin* is called the *slack time*, but the only differences are in the terminology; their respective purposes are one and the same. The system uses quantized time and runs in lock-step, meaning that the duration of each *round* is the same on all nodes and all nodes are kept in close temporal synchronization in real-time. Even housekeeping tasks are scheduled in periodic *administrative rounds* where the nodes perform clock synchronization, out-of-band communication with other nodes and update state parameters based on measurements.

The system assumes that by the time the next *round* begins; (1) all nodes have completed the execution of the last *round*, (2) all nodes have sent any messages to their appropriate destinations, and (3) those messages have been received by the destination. Therefore, the time-driven approach guarantees causality for events separated in real-time by some minimal amount, *delta*, whose value is influenced by network delays, execution times and clock synchronization across nodes. For those events not separated in real-time by *delta*, ordering is not guaranteed. Therefore, the system sacrifices the "100% logically correct" requirement in lieu of the ability to scale to very large sizes and retain low latency on real-world interaction.

Further, because conventional clock synchronization implementations cannot sufficiently synchronize both the phase and frequency of PC timers, this time-driven system implements its own time-keeping, bypassing the standard RFC 1589 periodic time-keeping method in the Linux kernel. Through the use of a low latency real-time network such as ATM and by placing the time-keeping core mechanism as close to the PC hardware as possible, it can achieve clock synchronization orders of magnitude better

than conventional mechanisms.

To guarantee resources for tasks in the distributed computation, the system "tags" preferred processes and a modified Linux scheduler gives preferential treatment to those processes. This time-driven framework for distributed computation was in production at the same time as KU Real-Time, so it could not take advantage of the analogous functionality that KURT-Linux provides, but ProTEuS does.

One very interesting features of this system that has thus far eluded ProTEuS is that it can dynamically adjust both the length of the *round* and the time-line offsets of each node as a result of measurements such as loss (late message arrival), network delay, execution times, etc. However, it has been shown that the system is, for all practical purposes, limited to a *round* duration no smaller than 1ms, and whether or not this is sufficient depends entirely on the granularity of the application. Furthermore, while some applications will tolerate the less than 100% guarantee on correctness, others, such as the simulation of an ATM network, will not.

This time-driven framework for large scale distributed computation, in the form presented here, is no longer being actively developed and has been largely superseded by ProTEuS. More information on ProTEuS can be found at *http://hegel.ittc.ukans.edu/projects/proteus/*.

Chapter 3

Implementation

3.1 Overview

To increase the efficiency of our proportional time simulations, we use real-time and embedded system techniques. By essentially configuring a rack of Linux boxes as an embedded system, we make the rest of the system and its services available to ProTEuS simulations. Further, in attempting to execute fine-grained synchronous distributed calculations in the presence of ambient load, it becomes evident that a finer level of timing and priority control is necessary. Consequently, we use real-time support to achieve fine-grained scheduling control to better dictate what processes will be running and when. Real-time execution control helps us run the simulation at a significant speed advantage over other methods and also allows the use of real systems, thus avoiding the over-simplification suffered by some other simulation techniques.

We use KU Real-Time modifications to Linux (KURT-Linux), a firm real-time system with increased temporal resolution, to control our distributed computation[52, 53, 25]. With temporal resolution increased to the microsecond level, KURT-Linux adds simple real-time scheduling services and is capable of satisfying the firm real-time performance constraints of many applications, including our proportional time simulations[24].

The simulation is based on a basic virtual time interval which we represent in real-time by an *epoch* of execution. In the case of an ATM network, this virtual time interval

is the time required to transmit an ATM cell. In an IP network, it may be the time required to process an MTU-sized* IP packet. Elements of the network simulation are distributed across physical machines, which are essentially configured as embedded components of the simulation, and real-time techniques are used to ensure that all simulation elements execute each epoch at the same periodic rate and remain closely synchronized.

The approach described here implements firm real-time ATM traffic management and ATM cell switching at a rate proportional to the simulated line rate. As part of an earlier project at the University of Kansas, a Rapidly Deployable Radio Network (RDRN), a simple best-effort software switch capable of switching ATM cells was developed through Linux kernel-level modifications[18, 19, 51]. Extensions to the software switch for the work presented here have included Available Bit Rate (ABR) traffic management support in the form of both the Explicit Proportional Rate Control Algorithm (EPRCA) and more recently, the Explicit Rate Indication for Congestion Avoidance (ERICA)[40, 1, 50, 32]. The software switch has also been modified to allow both cell and packet switching and now employs several output queueing disciplines. Efforts are also underway to provide PNNI-capable signaling support with a modified version of the Q.Port signaling software from Bellcore[45, 6].

To allow for arbitrary scaling of the network, we introduce the concept of virtual network devices[51]. In virtual network devices, Virtual Circuits (VCs) and/or peer Media Access Control (MAC) addresses are used to represent network connections in a virtual network and connections in the virtual network are multiplexed across these virtual interfaces. Also a byproduct of the RDRN project at KU, virtual network devices were first utilized only as a very simple layer of "glue" to create continuity and compatibility from the Linux ATM protocol stack to the ethernet device drivers in an effort to provide transparent ATM over ethernet services. In our experience however, the scope of virtual network devices goes far beyond their initial vision and this work has made significant advancements in extending their utility in new contexts.

In a ProTEuS ATM simulation, all simulated network connections share virtual cir-

*MTU: Maximum Transfer Unit. The ethernet MTU is typically 1500 bytes.

cuits across a real ATM network, while the traffic for all VCs in the simulated network, Virtual Virtual Circuits (VVCs), is multiplexed onto the real VC supporting the network connection in the simulated network. The virtual time line is divided into cell times each of which is proportionally represented in real-time by an epoch, as can be seen in Figure 3.1. The epoch must be long enough for the busiest physical host to execute tasks for the portion of the simulated network that it supports.

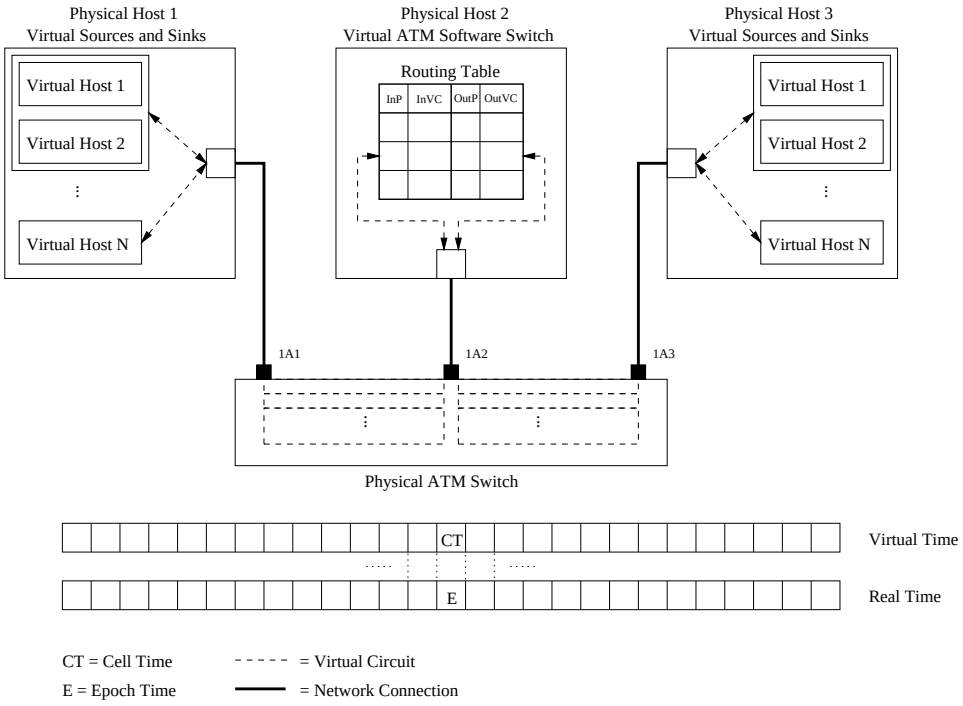

Figure 3.1: The ProTEuS Architecture

Our approach to mapping a simulated ATM network simulation onto a set of physical hosts is depicted in Figure 3.1 for a simulated network containing $2N$ virtual ATM hosts and one virtual ATM switch that is mapped onto three physical ATM-capable hosts. The principle, however, is the same for any number of switches or hosts, both virtual and physical. We certainly do not intend this figure to bias the reader into believing that sources, sinks, and switches must be mapped uniformly onto physical support machines as the figure suggests. The mapping of the simulated network onto the physical network is, in reality, *nearly arbitrary*. The only known restrictions are im-

posed by IP loopback when a virtual IP source and destination are on the same physical host and the monolithic properties of a virtual ATM software switch. In general, a single simulated entity cannot be split across physical hosts. The assistance of a real ATM switch is used to allow arbitrary mapping of simulation entities and facilitates the scaling of the simulation. Figure 3.1 also illustrates some of the more intimate subtleties of virtual ATM devices and virtual ATM switches, including virtual and physical traffic multiplexing and the impact of network interrupts, that will be discussed in detail in Sections 3.4 and 3.5.

3.2 Real-Time Distributed Synchronous Computations

In general, the implementation of synchronous distributed computations using conventional techniques limits their performance because conventional synchronization methods are not integrated with the system scheduling; they restrict the computations to a comparatively coarse temporal granularity and endure greater overhead because they reside at the user-level. Moreover, while many fine-grain synchronous computations can be implemented correctly without using real-time support, other application areas require it for correctness. For example, distributed virtual environments may require real-time control to adequately control user experience, while in a broad application area such as data collection, the necessity of real-time control will be determined by the semantics of the application itself; some will require it, others will not. However, even when real-time support is not required, our experience indicates that its use can improve performance. In all cases, precise resource allocation and scheduling control can make the distributed computations coexist with conventional computations more gracefully and maintain efficient performance in the presence of non-real-time load.

The periodic nature of ProTEuS computations supporting network simulation naturally leads to the notion of *epochs*, or periods, of execution. During epoch N, a computation with a single input and output consumes data produced by its supplier in some previous epoch, $N-\delta_1$, and produces data that will be consumed in some future epoch, $N+\delta_2$, by its consumer. As shown in Figure 3.2, each epoch consists of a computation

interval (CT) and some amount of slack time (ST), whose purpose is to allow the data expected from remote components to arrive before the subsequent epoch begins. Consequently, the epoch time (ET) is the sum of the computation time and the slack time: $ET = CT + ST$.

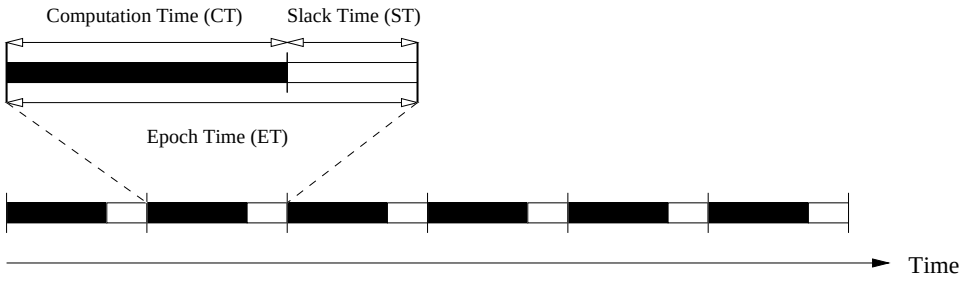

Figure 3.2: Structure of an Epoch

The computation interval in any given epoch N is itself partitioned into three segments as shown in Figure 3.3. The receive phase checks to see if all of the data that is necessary to continue with the current epoch has arrived, specifically, the data from epoch $N - \delta_1$. If it has not, then the epoch is skipped, or *missed* and will need to be reattempted at the next periodic invocation. If the necessary data has arrived, the computation phase produces data for consumption by downstream components during an ensuing epoch, specifically, epoch $N + \delta_2$. The third and final phase of the computation interval transmits the data produced to the destination component in preparation for the ensuing epoch on the destination hosts.

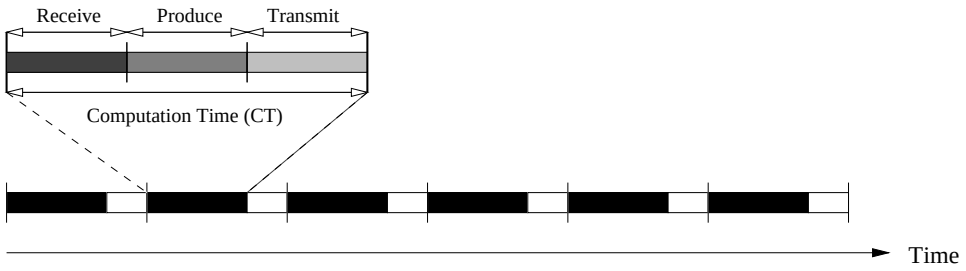

Figure 3.3: Structure of a Computation Interval

The δ_i represent potential delays between the production of data and its subsequent

consumption and the choice of the δ_i values is largely dependent on the semantics of the distributed application. Some applications will require the immediate consumption of data ($\delta_i = 1$). Others may use the δ_i as a method by which to buffer transmission delays and their variations on the network supporting the distributed computation. In that case, the δ_i may be set to maximize the probability that the necessary data has arrived in time to be consumed, and may, of course, be unique for each physical network link. Other applications may require specific δ_i values to serve a purpose in the simulated system. For instance, the ATM network simulations presented herein utilize δ_i to implement link delays in the simulated network.

The performance of distributed computations is largely dependent on the length of the epoch, and more specifically, the amount of slack time during which data from upstream nodes often arrive. Epochs with too little slack time will increase the number of missed epochs by increasing the probability that the necessary data has not yet arrived by the time it is needed at the beginning of the next epoch. Epochs with too much slack time will waste time unnecessarily waiting when they could be proceeding with the next epoch. Larger slack times decrease the probability of a missed epoch, but also increase the execution time of the distributed computation. The necessary slack time is strongly influenced by the transmission delay and the transmission delay variance of the supporting network over which the computation is distributed.

Further, different metrics have differing levels of significance for different driving applications. The best length of an epoch and its slack time component will depend on the relevant performance metrics of the application in question.

3.2.1 As a Basis for Distributed Simulation

At this point in the discussion, an obvious and necessary question arises, which is, *"Why implement network simulations as a synchronous distributed computation?"* The answer is manifest in the definition of a synchronous distributed computation. First and foremost, distributing the simulation across physical machines eases scaling; faster single processors are not sufficient to offset the scaling of the simulations. Further, the synchronization of distributed components ensures correctness, circumvents the rollback

problems associated with optimistic synchronization techniques such as Time Warp, and generally reduces the overhead involved in maintaining optimism (discussed in Section 2.2).

Of course, ProTEuS does not require real-time support to correctly simulate an ATM network, but real-time control makes the simulations more efficient. Because our periodic computations are fine-grained themselves, to achieve efficiency, we require scheduling granularities in the range of [1% - 10%] that of standard Linux. Consequently, we use KURT-Linux which can, in general, provide scheduling accuracy on the order of 10s of μs.

Real-time control affords ProTEuS the ability to more closely and easily maintain synchronization among distributed components. Although we have implemented an explicit synchronization method through the use of sequence numbers and simple handshaking, we are increasingly reassured through experimentation that such explicit synchronization is rarely necessary in ATM network simulations with sufficiently large δ_i, but its ultimate necessity will depend on the semantics of the particular application. While it is true that explicit synchronization ensures correctness, we can increase performance by instead using real-time control which provides a reliable and simple method for keeping distributed components executing at essentially the same rate. Furthermore, because one of the motivations in our ATM simulations is the ability to use the user and system code that real networks and applications use*, we require the ability to accurately and efficiently execute our simulation core in the presence of non-real-time processes.

In general, our goal is the just-in-time production of results from epoch $N-1$ in order to execute epoch N ($\delta_1 = \delta_2 = 1$). However, in the specific instance of communication network simulation, ProTEuS must simulate the transmission delay of each network link in virtual time. When data arrives, it will often be forced to wait a number of epochs before being consumed by the simulation entities in order to simulate a transmission delay in the simulated network. Therefore, the various δ_i associated

*Albeit properly virtualized. The code is the same, except in its notion of the passage of time (if any). See Section 3.6 for more details.

with each link in the simulated network will actually be functions of their simulated transmission delay.

3.2.2 Performance Metrics

We have identified three performance metrics that we believe usefully characterize a generic synchronous distributed computation and its efficiency:

1. *Execution time*: The total wall-clock execution time of the computation.
2. *Utilization*: The mean and variance of the CPU utilization of an epoch; that is, the percentage of the epoch occupied by the computation interval.
3. *Missed epochs*: The number of times an epoch is not allowed to proceed due to required data that is missing.

As alluded to previously, the importance of each of these metrics will, of course, vary with the application semantics.

For instance, in a distributed virtual environment, such as Internet multi-player games, the perceived realism of the game is the most important factor and the execution time is largely irrelevant; the game goes on as long as the players desire. In an environment such as this, a lower frame rate with lower error rate is likely to have a better look and feel than a faster frame rate with a high error rate. Therefore, maximizing the perceived frame rate, which is the actual frame rate minus the rate of missed frames, is a metric that involves striking a balance between the number of missed epochs and an acceptable epoch length.

Data acquisition, as another example, is typically driven by the desired accuracy of the data that it is collecting. Therefore, a data acquisition application has the freedom to vary the utilization and the number of missed epochs in order to meet some error tolerance restriction set by the application, but will also be influenced by both the data rate and the probability of loss. Therefore, in a data acquisition application, the design of the control system is also a factor in the relevance of the metrics.

With the simple explicit synchronization protocol present in ProTEuS, the correctness of its ATM simulations is not affected by the number of missed epochs. Rather,

we would like to choose an epoch length that minimizes the total simulation execution time, regardless of the number of missed epochs. We may choose to miss a greater number of shorter epochs to reduce the total execution time of the simulation. If the decrease in the epoch time ($\Delta ET = ET_{old} - ET_{new}$), multiplied by the initial, larger number of epochs is greater than the increase in the number of missed epochs ($\Delta ME = ME_{new} - ME_{old}$) times the new, shorter epoch time, then the net result is a decrease in total execution time, improving performance.

$$if\ ((\Delta ET * N_{old}) > (\Delta ME * ET_{new})) \rightarrow Performance\ Improves$$

An important item to note here is that in an application where we are trying to minimize the execution time, it is obvious that the system can move only as fast as its slowest component. Therefore, a key part of tuning such a system is *load balancing*, giving each distributed node supporting the distributed simulation, as far as possible, the same amount of work to do in each epoch. This minimizes the computation time component of the epoch which, in turn, reduces the epoch time.

Another item of consequence for any distributed computation is that the underlying network supporting it plays an important role in the selection of system parameter values and can significantly affect the performance of the computation. It is obvious that the speed and accuracy of the supporting network affects the missed epoch rate, which is one of several reasons that ProTEuS has chosen ATM as its initial support network. Dropped, corrupted and/or late messages are certain to increase the number of missed epochs and complicate the synchronization protocol. More reliable, faster supporting networks can reduce the epoch time by reducing the necessary slack time. Furthermore, the use of dual-processor hosts where one processor can be dedicated to servicing interrupts can further improve performance by preventing interrupt service from interfering with epoch progression.

3.3 An Overview of ATM Available Bit Rate (ABR) Service

ATM defines four traffic classes which receive different priorities and quality of service guarantees in the network. They are: Constant Bit Rate (CBR), Variable Bit Rate (VBR), Available Bit Rate (ABR) and Unspecified Bit Rate (UBR).

CBR connections are guaranteed a certain bandwidth that is negotiated upon connection establishment that remains firm for the duration of the connection and is typically used for real-time traffic with strict delay requirements. VBR connections are guaranteed a sustained cell rate, but their source rate is permitted to vary with time (burst) in compliance with a maximum burst size and is typically used for multimedia traffic. ABR uses the instantaneous *available* capacity in the network (bandwidth unused by CBR and VBR) which varies over time based on network feedback and is typically used for traffic such as WWW, or FTP sessions. UBR receives the remaining best-effort service and is used by applications that are insensitive to delay and loss, such as e-mail.

ABR connections can request a minimum bandwidth at connection establishment that will be guaranteed by the network. These connections can use more than the minimum bandwidth insofar as it is available from the network. This requires constant interaction between the network and the ABR sources in the form of feedback. Therefore, a closed loop flow control mechanism has been defined involving the source, the destination and the intermediate network (switches). As depicted in Figure 3.4, feedback information is conveyed to the source by means of special ATM cells designated as Resource Management (RM) cells. The network elements provide the source with information regarding congestion in the network and that information is used by the source to vary its transmission rate. The structure of an ATM RM cell is depicted in Figure 3.5 and the important fields are discussed in Sections 3.3.1 through 3.3.3.

It is these types of feedback loops, such as those present in ABR and TCP, that can potentially cause trouble for Time Warp systems by increasing the probability of rollback. For instance, ABR RM cell and TCP ACK feedback increases the number of messages exchanged between distributed components. This interaction consequently increases the probability of rollback which can, in turn, increase the simulation run-

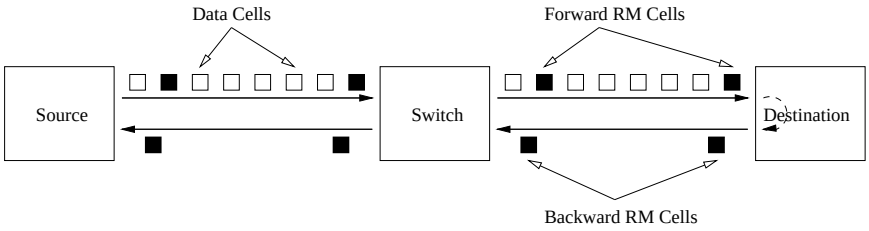

Figure 3.4: RM Cell Flow in ABR

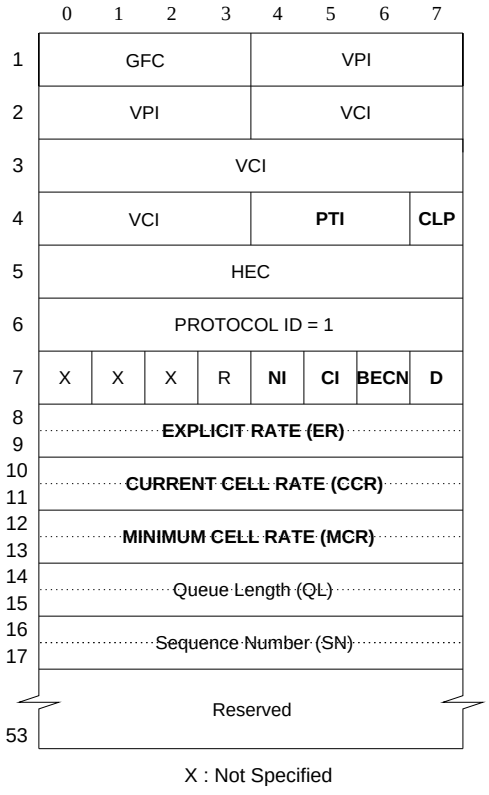

Figure 3.5: ABR RM Cell

time. Furthermore, in our experience, feedback loops can also affect discrete event simulation tools such OPNET and BONeS by encouraging users to introduce abstractions that can affect simulation results. For instance, we have encountered a series of BONeS ABR models in which RM cells were *not* circulated through the network, but rather RM cell feedback was accomplished through timer mechanisms based on round-trip delays. Upon timer expiration, results were relayed to the sources, but no RM cells actually circulated the simulated network[48, 40]. This caused significant disparity in some of the simulation results compared to simulations in which real RM cells circulate in the simulated network.

Table 3.1 contains a list of common ABR parameters, their meaning and default values for reference in Sections 3.3.1 through 3.3.3.

Parameter	Meaning	Default
ACR	Allowed Cell Rate	-
PCR	Peak Cell Rate	-
MCR	Minimum Cell Rate	0
ICR	Initial Cell Rate	PCR
NRM	Number of cells between forward RM cells	31
TRM	Maximum Time between forward RM cells	100ms
MRM	Minimum number of cells between forward RM cells	2
TBE	Transient Buffer Exposure	16777215
CRM	Number of missing, or in-flight, RM cells	$\lceil \frac{TBE}{NRM} \rceil$
CDF	Cutoff Decrease Factor	$\frac{1}{16}$
ADTF	ACR Decrease Time Factor	500ms
RIF	Rate Increase Factor	$\frac{1}{16}$
RDF	Rate Decrease Factor	$\frac{1}{16}$

Table 3.1: Common ABR Parameters and Meanings

3.3.1 Source Behavior

In order to take advantage of available bandwidth and to avoid cell loss due to congestion, ABR sources must periodically poll the network to retrieve information concerning the current network state. It is the duty of the ABR source to generate and dispatch RM cells. The following is an abbreviated list of ABR source behavior[1, 28].

1. ABR sources always send at a rate less than or equal to the **ACR**. Further, the **ACR** must lie between the **MCR** and the **PCR**.

2. ABR sources begin sending cells at the **ICR** and the first cell sent should be a Forward RM (**FRM**) cell.

3. ABR sources should send a **FRM** cell every **NRM** data cells, or if the time since the last **FRM** cell sent is greater than **TRM**. However, at least **MRM** other* cells must be sent between **FRM** cells.

4. If no **FRM** cells have been sent after a period of **ADTF**, then the **ACR** should be reset to the minimum of the **ACR** and the **ICR** and a **FRM** cell should be sent. This is a *use-it-or-lose-it* policy intended to keep ABR sources from retaining stale **ACR**s.

5. If **CRM FRM** cells have been sent without receiving any Backward RM (**BRM**) cells, then the source rate should be decreased by the **CDF**, subject to the **MCR**.

6. Upon reception of a **BRM** cell, ABR sources should reset the **ACR** based on the Congestion Indication (**CI**), No Increase (**NI**) and Explicit Rate (**ER**) information from the received **BRM** cell as shown in Table 3.2.

NI	CI	New Allowed Cell Rate
0	0	MIN(ER, ACR + PCR × RIF, PCR)
1	0	MIN(ER, ACR)
X	1	MIN(PCR, ACR − ACR × RDF)

Table 3.2: Setting ACR Upon Backward RM Cell Reception

3.3.2 Switch Behavior

The principal responsibility of ATM switches is to monitor congestion on the switch and provide feedback to ABR sources. Because of the bursty nature of VBR traffic, the

*Other cells are defined as either data cells or Backward RM cells

instantaneous bandwidth available to ABR connections varies constantly with time and may cause congestion and perhaps even cell loss. The goal of the network is to keep ABR cell loss to a minimum while efficiently allocating the network bandwidth.

At the reception of a **BRM** cell, switches invoke a congestion control scheme whose purpose is to place feedback information into the RM cell. However, congestion control schemes are vendor-specific. That is to say that the scheme used and its implementation is at the discretion of the ATM switch manufacturer. Several congestion control schemes have been proposed and they generally occur in three fundamental forms; binary, explicit-rate and credit-based.

Binary feedback is feedback based on a binary condition, such as *congested* or *not congested*. An early binary feedback mechanism used the Explicit Forward Congestion Indication (**EFCI**) bit in the ATM header. The **EFCI** is a method by which congested switches on the path from source to destination can notify the destination that congestion was experienced (by setting the bit). Switches monitored their queue lengths and set the **EFCI** when they exceeded a given threshold. Destinations monitored the **EFCI** bit and sent RM cells back to the sources periodically where the source used an additive increase, multiplicative decrease algorithm to adjust their source rates. Binary schemes can be unfair however, because the penalized source(s) may not be the one(s) causing the congestion.

It was also quickly noted that a binary feedback scheme would operate too slowly in high speed networks and that an explicit-rate scheme would be faster and allow for more flexibility. Rather than providing binary feedback, explicit-rate schemes provide sources with an express rate at which to send cells. Explicit-rate schemes converge faster than binary schemes and also make rate policing on the switch easier because the policer can monitor RM cells for the rate to use in policing algorithms; the explicit rate is one of the fields of an RM cell, which can be easily monitored by the policer. Further, they are robust in the sense that RM cell loss does not significantly degrade performance because the next RM cell to arrive will return the system to the correct operating point.

The credit-based approach consists of per-link, per-VC window flow control. The

receiver monitors queue lengths at each VC and determines the number of cells that can be sent on that VC. This parameter, known as the credit, is the number of cells that the sender may send. To prevent credits from being lost, bookkeeping is done to detect cell loss on the VC and the receiver then reissues credits to compensate. One problem with credit-based schemes is that each VC must reserve enough buffer space to fill the link capacity even though the link is potentially shared by many VCs. This can be resolved by adapting the number of credits issued in proportion to the activity of a VC, but consequently introduces ramp-up delays.

In short, it is the responsibility of the switch to fill in the the **ER**, **CI**, and **NI** fields of the **BRM** cells by some method, which is likely to vary from switch vendor to switch vendor. One should note that explicit-rate is the official congestion control scheme adopted by the ATM Forum, however, there have been many different explicit-rate schemes proposed, none of which is discussed in detail here. The specific congestion control schemes implemented in the Virtual ATM Software Switch are discussed briefly in Section 3.5.4.

3.3.3 Destination Behavior

The primary role of destinations in an ABR connection is to turn-around the **FRM** cells that it receives in an expeditious manner. This basically entails changing the direction bit in the **FRM** cell, making it a **BRM** cell, and sending it back out on the VC on which it was received. The following is an abbreviated list of ABR destination behavior[1, 28].

1. ABR destinations should monitor incoming data cells and store the **EFCI** bit from the ATM header.

2. When a **FRM** cell is received, the destination should set the direction bit, making it a **BRM** cell, and set the **CI** bit if the **EFCI** was set in the last data cell received.

3. In a fashion similar to that of switches, destinations may lower the **ER** in the RM cell or set the **CI** and/or **NI** bits if it is experiencing internal congestion.

4. ABR destinations may generate a **BRM** cell without having received a **FRM** cell in order to increase the responsiveness of the source. In this case, the destination

should set the Backward Explicit Congestion Notification (**BECN**) and **CI** bits in the RM cell.

3.4 Virtual Network Devices

Virtual network devices are a Linux kernel-level abstraction that creates what the socket implementation in the kernel *believes* is an actual network device. When created, virtual network devices are associated with physical network devices which can be ethernet or ATM and these physical network devices carry the traffic for their associated virtual network devices. In this way, with potentially only a single physical network device, the system can simulate traffic on an essentially unlimited number of virtual network devices. Current support includes virtual ATM over physical ATM and virtual ATM over physical ethernet, but efforts already exist to extend the concept to virtual ethernet devices and Proportional Time Emulation and Simulation of IP networks.

Network devices in the Linux kernel are an abstraction intended to provide broad coverage of both current and future networking devices. In that way, they are a slightly specific example of the familiar device driver abstraction which is designed to suit a vast array of PC devices. Their generic facade permits the ability to present the operating system with a pseudo device that has the familiar form of a network device, but whose implementation is completely dissimilar, acting as an intermediate network device driver between the network protocol stack and the physical network device drivers.

3.4.1 Architecture

Virtual network devices are implemented in the Linux kernel as a pseudo device driver, */dev/vdev*. This implementation technique is a popular one because it allows simple and uniform user-level access to kernel implementations. User-level programs written to create and configure virtual network devices do so chiefly through `ioctl` calls on the pseudo device. As shown in Figure 3.6, the virtual network device layer, which resides between the Linux socket implementation and the device drivers of the physical

network devices, consists of any number of configurable layers. This is an important feature because it allows per-instance user-level configuration of virtual network device functionality and makes adding functionality to existing virtual network devices simple, modular and unobtrusive.

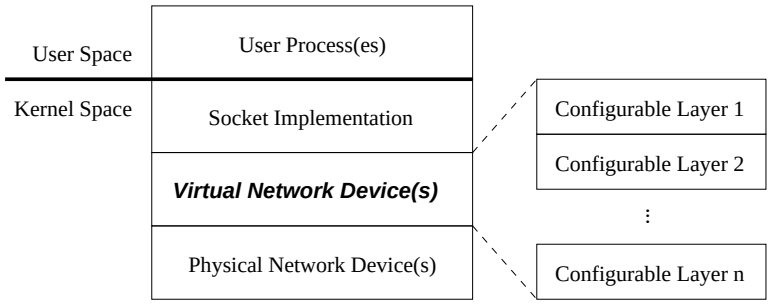

Figure 3.6: The Virtual Network Device Stack

The method by which virtual network devices insert themselves into the kernel protocol stack is exceedingly simple. To all parts of the Linux kernel, with the exception of the device drivers, a device is just an abstract object. Only the device drivers really know the details of a particular device. Therefore, creating what the kernel will believe is a network device simply requires instantiating the kernel structure used for the type of network device being virtualized and registering the device with the kernel through a provided API. This is similar to the way that physical network devices are registered as well. During kernel boot, device driver initialization functions probe the PCI (or ISA) bus looking for devices. When and if they find one, they, among other things, allocate one of these kernel device structures, initialize it, and call a function to register it with the appropriate kernel entity.

However, for virtual network devices, it is also necessary to ”insert” the virtual network device layer functionality between the socket implementation and the physical device drivers unobtrusively, which is accomplished by correctly setting some appropriate function pointers. Most types of device structures in the Linux kernel have an operations structure that contains function pointers for common device operations such as *open*, *send*, *close*, etc. Rather than pointing these function pointers to the physi-

cal device driver entry points, as is done for physical network devices, virtual network devices point them into the analogous entry points in the virtual network device implementation. Since each device has its own operations function pointers, virtual network devices can insert the virtual network device stack into their kernel stack without affecting the operation of the physical network hardware.

Likewise, the Linux kernel also provides the ability to intercept incoming traffic on physical network devices destined for virtual network devices through the use of function pointers. Ethernet packets can be tagged with a packet type to identify them and ATM cells and packets can be identified by the virtual circuit on which they are received, which allows virtual network device traffic to be routed to the necessary virtual network device receive entry points and segregated from traffic on the physical network devices.

The potentially complicated part of a virtual network device is the bookkeeping associated with multiplexing and demultiplexing virtual device traffic on the physical network devices and the emulation of physical device functionality, such as ethernet fragmentation, ATM segmentation and ATM traffic shaping. Although for many applications, a truly abstract virtual network device implementation may be acceptable, for ProTEuS network simulations, it is important that virtual network devices faithfully perform many of the necessary aspects of a real network device to avoid the oversimplification suffered by many other alternative network simulation techniques.

To the protocol stack, virtual network devices look like network devices. To the physical network devices, virtual network devices look like protocol implementations. The real power of virtual network devices lies in the fact that from the protocol stack entry point up, kernel and user-level entities are oblivious to the fact that the devices are virtual; *they do not know or care*. Similarly, from the physical device driver entry points and down, the physical devices have *no idea* that the devices are not actually protocols.

3.4.2 Virtual ATM Devices

Virtual ATM devices are a form of virtual network device that creates a pseudo ATM interface that can be utilized as any physical ATM device, including supporting native ATM applications and Classical IP (CLIP) over ATM. Further, virtual ATM devices are also eligible to become Virtual ATM Software Switch ports, as discussed in Section 3.5, and support exists for using either a physical ethernet device or a physical ATM device as the physical support network. Although these are the only currently supported physical network device types for virtual ATM, there is nothing magical about them and virtual network devices in general can be associated with any physical network device, or any physical communication device for that matter, for which the Linux kernel has support.

Figure 3.7 depicts the virtual ATM device stack in the Linux kernel used by ProTEuS ATM network simulations, which utilize both Segmentation and Reassembly (SAR) and Proportional Time (PT) layers. However, as mentioned earlier, virtual devices, ATM or not, can utilize any combination of available layers; we show this configuration only as a common example. The ATM Adaptation Layer (AAL) layer is a non-configurable layer that exists on *all* virtual ATM devices. The individual virtual ATM layers are discussed in more detail in Sections 3.4.2.2 and 3.4.3.

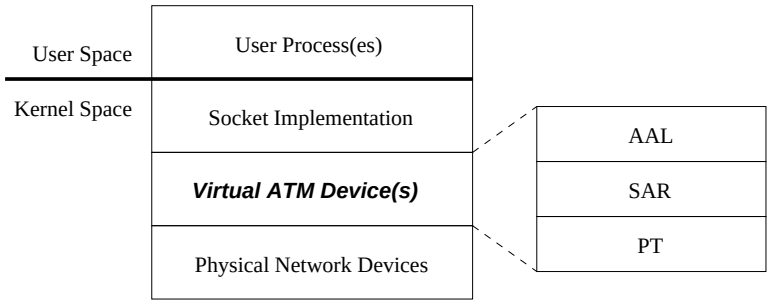

Figure 3.7: The Virtual ATM Device Stack

When an application opens a virtual circuit on a virtual ATM interface, what is actually created is what can only be termed a Virtual Virtual Circuit (VVC). No resources are allocated and no connection is established on the physical network in the conven-

tional ATM sense. All reads and writes on these VVCs are multiplexed at the cell level onto the virtual ATM interface.

3.4.2.1 Physical Network Device Support

When the physical network device is an ATM device, virtual ATM creates an AAL5 virtual circuit over which it will multiplex all traffic for the virtual device. Multiple virtual flows (VVCs) among multiple virtual devices are multiplexed onto these physical flows (VCs). Point-to-point connections in the ATM sense are created by VC pairings; two virtual devices communicate point-to-point over a VC, emulating a physical ATM connection (fiber).

As part of the virtual device configuration, the *push* function pointer for this virtual circuit is set to the virtual ATM over ATM receive function rather than the default Linux-ATM receive function. This is the hook from the receive side of the physical device driver into the virtual device implementation. The *push* function is a Linux-ATM specific implementation that is invoked when a cell on an AAL0 VC or packet on an AAL5 VC is received. Each virtual circuit has its own *push* function pointer, allowing virtual ATM devices the redirection they need without disturbing normal Linux-ATM functionality on other VCs.

When an AAL5 packet is received on a VC supporting a virtual network device, the *push* function is executed, which passes the packet up the configured virtual device layers. Each layer propagates the packet up the stack after any necessary processing is done. For example, the SAR layer will hold cells on AAL5 VVCs until a completed AAL5 packet has been successfully reassembled, at which time it will pass the reassembled packet up to the next layer (See Section 3.4.2.2). Eventually, upon exiting the top of the AAL layer shown in Figure 3.7, the *push* function for the *virtual virtual circuit* is invoked, which is typically the default Linux-ATM receive function, which routes the data back into the normal ATM protocol stack and socket implementation as if it was received on a conventional virtual circuit.

If the virtual ATM device is configured with a Proportional Time layer (See Section 3.4.3), then multiple virtual ATM devices may share a VC on the physical ATM device.

This option was implemented in an effort to reduce the number of network interrupts suffered on a machine supporting many virtual ATM over ATM devices. The PT layer performs the necessary multiplexing and demultiplexing based on peer Media Access Control (MAC) addresses that are user-supplied upon virtual ATM device configuration.

Likewise, when the physical device is an ethernet device, because ethernet and IP are not connection-oriented, Virtual ATM creates a point-to-point connection in the ATM sense by an association between peer MAC addresses. As with proportional time virtual ATM devices, the destination MAC address for virtual ATM data is located in the packet payload in order for the destination to demultiplex the cells and route them to their appropriate destination virtual ATM devices.

Further, a new ethernet packet type is created that is used to identify ethernet packets generated by and destined for virtual ATM device implementations. Similar to the ATM *push* function, packet type definitions in Linux contain a *func* function pointer, which virtual ATM over ethernet devices set to the virtual ATM over ethernet receive function. When a packet of this type is received on an ethernet device supporting a virtual ATM device, the *func* function is executed, which hooks into the virtual ATM device implementation and passes the packet up the configured virtual device layers.

At the point where the packet is passed from the particular virtual device receive function up to the bottom layer of the virtual device stack, the implementation does not depend on the physical device type. That is, the physical network supporting virtual devices is transparent to the implementation of the configurable layers* and they should have no need to know its intimate details.

3.4.2.2 Existing Configurable Layers

There are two virtual ATM layers that have already been implemented as part of previous work that provide important functionality that is utilized by ProTEuS ATM network simulations[51]. They are the ATM Adaptation Layer (AAL) layer and the Seg-

*With the temporary exception of the Proportional Time layer which currently requires the physical support network to be ATM.

mentation and Reassembly (SAR) layer.

ATM Adaptation Layer (AAL)

The AAL layer is a layer that must be configured on all virtual ATM devices. In short, it is the "glue" between the Linux-ATM protocol stack and the virtual device implementation. It provides the entry points into virtual devices from the socket implementation on the send side and out of the virtual devices back into the socket implementation on the receive side. Aside from cementing the virtual device layer into the kernel, the AAL layer also provides some clandestine housekeeping functions to keep the Linux-ATM implementation happy.

Segmentation and Reassembly (SAR)

The SAR layer performs ATM segmentation and reassembly that is an essential point of functionality in trying to faithfully emulate ATM interface card semantics, which typically support this operation in hardware. The virtual ATM SAR layer supports the two most common ATM adaptation layers, AAL0 and AAL5.

In an AAL0 connection, raw, 53 byte (48 byte payload) cells are sent and received, but no other processing is done. In Linux-ATM, applications that open AAL0 connections can only read and write 52 byte buffers (an entire ATM cell, minus the Header Error Check (HEC)). The application is responsible for filling in the entire ATM header except for the HEC. This includes the Virtual Circuit Identifier (VCI), the Virtual Path Identifier (VPI), the Cell Loss Priority (CLP), the Generic Flow Control (GFC), and the Payload Type Indicator (PTI). The virtual ATM SAR layer inserts the one byte HEC and passes 53 byte cells down the virtual ATM layer stack. When a ATM cell is received on an AAL0 connection, the SAR layer checks the HEC and discards the cell if an error is detected. If not, it removes the HEC and passes the remaining 52 bytes up to the application.

In AAL5 connections, applications read and write packets of variable length. The SAR layer segments the packets into 48 byte payloads and attaches full ATM headers, sending 53 byte ATM cells down the virtual ATM device stack. The last cell of an

AAL5 packet is marked as such in the PTI and contains an AAL5 trailer at the end of the payload that includes, among other things, a packet length indicator. When cells arrive on an AAL5 connection, the HECs are checked and, if no errors are found, the payloads are queued until the last cell of the packet is received. If an error is found, the entire packet is discarded, including the cells that have yet to arrive that belong to the erred packet. The length indication from the AAL5 trailer permits the SAR layer to reassemble the error-free payloads into a variable length buffer matching the one sent on the transmit side that is passed up to the application.

3.4.3 Proportional Time

Proportional Time is a configurable layer on a virtual ATM device that provides the control structure for ProTEuS synchronous distributed proportional time ATM network simulations. Current support is available for virtual ATM devices over physical ATM devices only; efforts are ongoing for virtual PT ethernet and virtual PT ATM over physical ethernet.

The Proportional Time layer is controlled by a real-time kernel thread that is invoked under KURT-Linux control on a static explicit real-time schedule that establishes the epoch time. The period of the real-time thread's execution is predetermined and set by the user when the kernel is switched into real-time mode. In general, the length of the epoch should be long enough for the busiest physical host to complete all of the tasks in each cell-slot for the portion of the virtual ATM network that it supports.

3.4.3.1 Synchronization

ProTEuS synchronization is accomplished through a *very* simple mechanism. During each epoch, each host sends out a packet on every configured virtual circuit supporting the simulation whether or not any data (ATM cells) needs to be transmitted or not. All ProTEuS packets are timestamped with the epoch during which they were generated and each physical host knows, through the setup of ProTEuS, how many packets to expect per epoch (one per configured supporting virtual circuit).

If all of the packets from epoch $N - \delta_1$ have arrived, then the epoch will proceed. If

not, the epoch will be *missed* and it will be re-attempted during the next KURT-Linux periodic invocation, presuming that the missing data has arrived in the meantime. As alluded to in Section 3.2.1, in ProTEuS network simulations, there is a need to model simulated transmission delays, so cells are often forced to wait in queues after they have arrived. This circumstance allows ProTEuS the opportunity to raise the value of δ up to the minimum of the simulated link delays imposed on this physical host. In other words, we can introduce a synchronization *tolerance*, where we will allow epochs to proceed having not yet received all expected messages. However, because the tolerance is the *minimum* of all simulated link delays, it does guarantee having received all messages with a potential bearing on the current epoch.

This phenomenon is depicted in Figure 3.8 where a packet is generated by a remote host in epoch $N - 1$. The packet is generally expected to arrive in epoch N, but a simulated link delay of four epochs, allows the packet a window of valid epochs in which to arrive. In this example, the packet actually arrives in epoch $N + 2$ and is subsequently forced to wait in a queue until epoch $N + 5$, simulating its network link delay*, at which time it is finally processed.

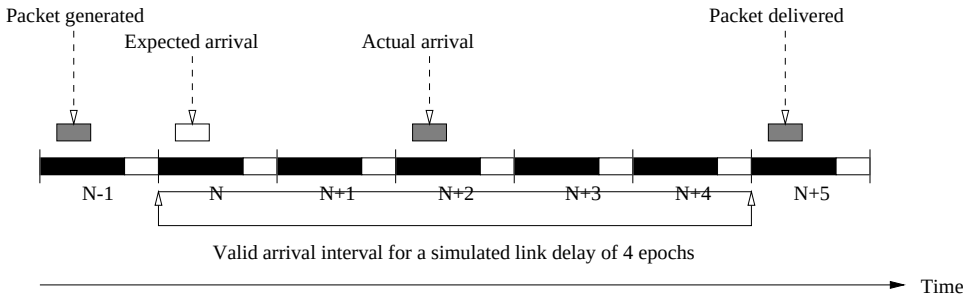

Figure 3.8: Simulated Link Delays and Their Impact on Synchronization

The simplicity of the synchronization protocol is the principal reason that ProTEuS does not currently support ethernet as the physical support network; the protocol contains no contingencies whatsoever for lost packets. To support ethernet, the synchronization mechanism in the Proportional Time layer would have to be extended

*Those with a keen eye will notice that a link delay of four would actually have the packet being processed in epoch $N + 4$. However, one epoch is added to all link delays to simulate clocking delay.

to utilize a simple fixed-window windowing scheme to detect lost packets or out-of-sequence packet arrivals. Further, an acknowledgement or retransmit request mechanism would need to be implemented to recover from such failures and transmitting hosts would need to retain packets until they are acknowledged or fall outside the sliding epoch window. ATM, on the other hand, is reliable, low delay and essentially loss-free. The ProTEuS synchronization protocol contains a simple mechanism to detect the loss of a proportional time packet in the physical network. Through hundreds of ProTEuS experiments lasting millions of epochs each, resulting in physical ATM cell transmissions in the billions, not a single ATM cells has ever been lost.

3.4.3.2 Epoch Phases

The real-time thread controlling proportional time executes a conventional suspend-work loop as shown in Program 3.1. At the top of the loop, the thread is suspended by KURT-Linux if either the kernel is not in real-time mode, or until the thread is next scheduled. When the thread is awakened, the work phase of an epoch ensues.

Program 3.1 Proportional Time Thread Suspend-Work Loop

```
1  while (!rt_suspend(SUSPEND_IF_NRT)) {
2
3      if (receive() == SUCCESS) {
4          produce();
5          transmit();
6      }
7
8      if (stop_time()) break;
9  }
```

Each real-time epoch consists of three fundamental phases as previously alluded to in Section 3.2; receive, produce and transmit. The iteration of the loop is terminated by a stop condition that signals the end of the simulation, at which time the real-time thread breaks out of the loop and exits.

The receive phase ensures that all data necessary to proceed with the epoch has arrived as enforced by the synchronization protocol.

If the receive phase succeeds, the Proportional Time layer then moves to the pro-

duce phase. This phase entails first unpacking and demultiplexing the received packets and imposing the simulated link delays on all cells in the packets. It then pushes those cells whose simulated delay has expired up the virtual device stack and into the software switch or the kernel socket implementation. At this point, any necessary cell-switching is accomplished by invoking the virtual ATM software switch switching function, followed by servicing all open virtual virtual circuits on each proportional time virtual ATM device through performing cell-level traffic shaping. As preparation for the transmit phase, each virtual ATM device is permitted to select a single cell to send during this epoch if one is so enabled.

This is followed by the transmit phase, during which the enabled cells sharing each supporting physical virtual circuit are multiplexed into AAL5 packets, one per physical support VC, that are relayed to the destination host.

3.4.3.3 Sending and Receiving

When sources send data on an open virtual virtual circuit on a virtual ATM device, the buffer propagates its way down to the virtual device layers, where the data is eventually sent out onto the network. Those configured with a SAR layer segment the buffer into 48 byte payloads and propagate a *train* of 53 byte ATM cells to the layers below. On a virtual device void of a PT layer, the cell train is passed down to the lowest virtual device layer, which encapsulates the ATM cells into a network packet and sends it out on the physical network device.

When the virtual ATM device is configured with a PT layer, which resides below the SAR*, the cells are buffered by the layer and are *not* immediately sent out onto the physical support network. As depicted in Figure 3.9, cells entering the Proportional Time layer are queued into a transmit queue on a per-VVC basis where they wait to be enabled by the traffic shaper.

As part of the work during each epoch, cell-level traffic shaping is performed on every open VVC with pending cell transmission. Traffic shaping applies a decision function to each VVC, whose purpose is to decided whether or not the VVC should

*Proportional time virtual ATM devices *must* be configured with a SAR layer above the PT layer.

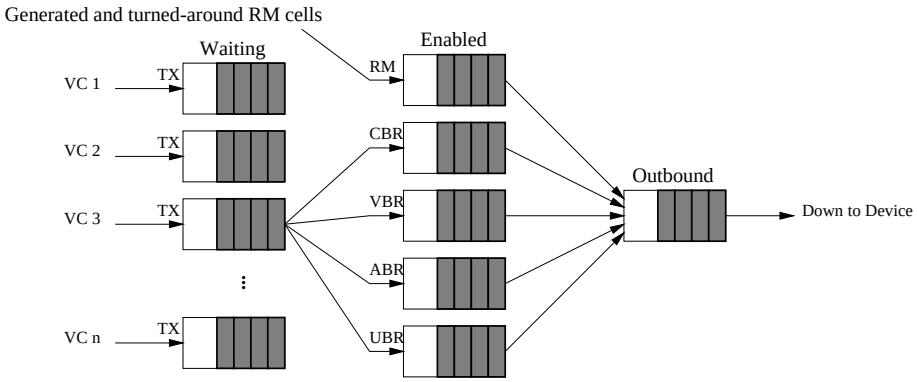

Figure 3.9: Transmit Side Queueing in the Proportional Time Layer

be allowed to send a cell in this epoch, based on its current cell rate (`CCR`). Program 3.2 shows the algorithm used to shape ATM traffic, which guarantees that a source sending at rate `CCR` cells per second on a simulated link of speed `linerate` cells per second will send exactly `CCR` cells in `linerate` cell-slots. The algorithm itself is very simple and spaces cell transmission on the link as evenly as possible, utilizing remainder accumulation to approximate periodic transmission in accordance with the `CCR`.

Program 3.2 Proportional Time Traffic Shaping

```
1   /* Initialization. */
2   send = linerate / CCR;
3   remainder = linerate % CCR;
4
5   /* During each epoch. */
6   if (!--send) {
7       /* Send a cell. */
8       send = linerate / CCR;
9       remainder += linerate % CCR;
10      send += remainder / CCR;
11      remainder = remainder % CCR;
12  } else {
13      /* Do not send a cell. */
14  }
```

CBR and ABR are shaped by this algorithm, while VBR and UBR are not. Of course, in a CBR connection, `CCR` never changes, while in an ABR connection, it has the potential for change at the reception of each **BRM** cell.

VBR traffic is implemented through the ATM Reference Traffic Source (ARTS) in ProTEuS which dispenses cells based on an explicit time line schedule. This is a very interesting and intriguing feature because it essentially means that ProTEuS can replicate *any* ATM stream whose cell transmission schedule can be explicitly captured. For instance, some of the ARTS VBR sources used in Chapter 4 were cell traces of an MPEG clip from the movie *Star Wars* streamed across a real ATM network. ARTS can also be used to replicate other fundamental source cell or packet patterns such as on-off, or statistically significant ones such as Markovian. All that is required are statistical random number generators, such as those contained in the GNU Scientific Library, and a simple program to generate the schedule.

UBR traffic is shaped using a token bucket algorithm where user-level programs can specify a token replenishment rate and a maximum bucket size, which default to the simulated line rate and one, respectively. In ProTEuS, by default, UBR cells are *always* allowed to transmit, however, they find themselves behind all other traffic classes in priority for the link once they are enabled. When UBR cells are waiting to be transmitted, the shaper hands out tokens, each of which allow a cell to be enabled, as long as they are available. When the bucket is emptied, cells are blocked and the bucket is replenished with tokens at the specified rate.

When cells are enabled by the shaper, they are transferred from their VVC queue to the device queues, where cells are separated by class. Each device has five queues as shown in Figure 3.9; one each for RM, CBR, VBR, ABR and UBR. At the conclusion of each epoch, each virtual ATM device is allowed to send *one* cell if there are any enabled. This is, of course, because we specifically chose the ATM cell time as the basic virtual time unit. Therefore, each epoch can result in at most one cell being sent on each simulated network link. The current mechanism used to choose a cell is a strict priority criterion which services traffic classes in the order; RM, CBR, VBR, ABR and UBR. Therefore, if there is an RM cell waiting, it will be sent regardless of the occupancy of other queues. If no RM cell is waiting, but a CBR cell is, then the CBR cell will be chosen, etc. Chosen cells, at most one from each proportional time virtual ATM device, are then enqueued onto an outbound queue shared by all PT devices being

multiplexed onto a particular virtual circuit in the physical network. The contents of each of these multiplexing queues will be bundled and transmitted during the last phase of the epoch.

Figure 3.10 depicts the structure of a proportional time packet. The first four bytes is a long integer that conveys the epoch number in which the packet was created. Following the epoch information, which is primarily used for synchronization purposes, is a four byte long integer containing the status flags. The status flags can be used for any purpose, but currently contain information about the state of the node generating the packet. State values include `PT_INACTIVE`, which indicates the the host has no more active sources or sinks, `PT_STOP`, indicating that the host is prepared to stop the simulation, and `PT_HALT`, which means that the host is in the process of halting its support of the simulation. These flags are used when the criteria for ending a simulation is based on a number of data cells sent and/or received as opposed to a set stop time. Following the status flags are any number of demultiplex information - ATM cell pairs, including zero; synchronization information is sent in every epoch regardless of the traffic, or lack thereof, in the virtual network. The demultiplexing information for each cell is a unique six byte MAC address* in the simulated network assigned to each virtual ATM device, by the user, at configuration time. Each ATM cell is preceded by a MAC address, allowing the receiving PT layer to identify the destination PT virtual ATM device by associating virtual point-to-point connections between pairs of peer MAC addresses in the simulated network.

When an AAL5 packet is successfully received and reassembled on the physical network supporting ProTEuS, the Interrupt Service Routine (ISR) merely queues the packet onto a receive queue for the shared physical VC on which the packet was received as shown in Figure 3.11. When the next epoch begins, under KURT-Linux control, the receive phase of the epoch determines whether or not the epoch may proceed by checking the synchronization information in the packets that have arrived thus far. If the epoch is allowed to proceed, all packets timestamped with an epoch previous to

*The MAC addresses *should* be unique, but recall that they are user-assigned upon virtual ATM device configuration, so no guarantees can be made. Non-unique MAC address certainly can, and often will, lead to incorrect demultiplexing of ATM cells.

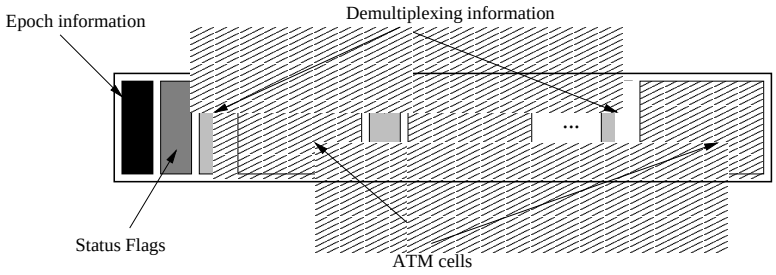

Figure 3.10: The Structure of a Proportional Time Packet

the current epoch are demultiplexed. This entails iterating through each packet, using the demultiplexing information to locate the destination virtual ATM device for each cell, and moving those cells into the receive queue on the destination virtual ATM device.

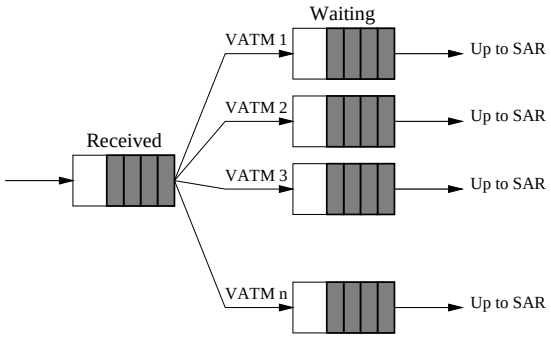

Figure 3.11: Receive Side Queueing in the Proportional Time Layer

All PT virtual ATM devices have the option of specifying a simulated network transmission delay. When cells are moved into the receive queues on their respective virtual devices, they may be forced to wait to simulate this delay. While all cells are forced to wait at least one epoch to simulate clocking delay, the simulated link delay may be zero. The wait queue is implemented as a timeout queue, such that only the head of the queue ever needs to be checked to determine cell release. When a cell is released from the queue, it is passed up the virtual device stack to the SAR layer.

A recent modification of ProTEuS ATM simulation was the ability to *loopback* virtual

devices so that both endpoints of a simulated connection can be mapped onto the same physical machine, eliminating altogether the need for a VC in the physical network. Furthermore, this permits a more flexible, almost arbitrary, mapping of the simulated domain onto the physical. When virtual ATM devices with PT layers are configured, they attempt to find their peer on the local machine by MAC address association. If they are successful, they are configured as *loopback* devices. When cells from a loopback device are selected for transmission, they are not multiplexed into a network packet, but rather queued directly onto the receive queue of their peer, where they too will endure any simulated network transmission delay.

It should also be noted that there are several minor exceptions to the implementation described here. For instance, when **FRM** cells are turned-around by destination PT virtual ATM devices, they are not queued in the transmission queue for the VC on which they were received (and will be subsequently transmitted). Instead, they are queued directly onto the RM cell queue on the virtual ATM device for immediate transmission. Further, when PT virtual ATM devices are configured as virtual ATM software switch ports, traffic shaping is *not* performed on any of the VCs, regardless of traffic class. Any shaping, i.e. weighted-round robin cell service, is done at the software switch level, such that cells are unconditionally enabled for transmission when available at the PT layer.

3.4.3.4 ABR Support

Support for ABR service is present in the Proportional Time layer in both the source and destination behavior whose ATM Forum specification is summarized in Sections 3.3.1 and 3.3.3. Although the current Proportional Time platform implementation supports ABR cell service in the PT layer, the layered architecture of virtual network devices allows a cleaner implementation, separating ATM traffic shaping, including that of ABR, into a separate ATM Quality of Service (QoS) layer residing logically above PT. This is part of the future work that will generalize the use of proportional time for other applications.

As alluded to earlier, ABR sources are shaped at the cell-level by the fundamental

algorithm in Program 3.2. However, for ABR streams, the shaper is complicated by two other factors. First, if the shaper determines that an ABR source may send a cell, it must decide whether it will be a **FRM** cell or a data cell in the current cell-slot. Further, if the source rate is instructed to fall to zero, **FRM** cells must be periodically sent to probe the network, or the rate may never again be non-zero.

Below is an abbreviated list of the supported ABR source behavior in proportional time virtual ATM devices.

1. ABR sources always send at a rate less than or equal to the **ACR**, which lies between the **MCR** and the **PCR**.

2. ABR sources begin sending cells at the **ICR** and the first cell sent is a **FRM** cell.

3. ABR sources send a **FRM** cell every **NRM** data cells.

4. Upon reception of a **BRM** cell, ABR sources reset the **ACR** based on the **CI**, **NI**, and **ER** from the received **BRM** cell as shown in Table 3.2 on page 35. If the new **ACR** is zero, a timer is set to probe the network with a **FRM** cell every 100ms (in virtual time) and is not turned off until a returning **BRM** contains a non-zero **ER**.

ABR destinations have an extremely simplified implementation in PT virtual ATM devices and support only one destination behavior, albeit the most important one. When a **FRM** cell is received, the destination sets the direction bit, making it a **BRM** cell, and sends it back out on the virtual virtual circuit on which it was received.

3.5 The Virtual ATM Software Switch

The virtual ATM software switch was initially created as a method by which to provide simple software supported ATM cell switching capabilities at a fraction of the cost of a real ATM switch[51]. Although the software implementation restricts the achievable real-time throughput, it also provides flexibility for experimentation and extension. As was the case with virtual network devices, the potential uses for the software switch has far transcended its intended purpose. For ProTEuS network simulation, the need

emerged to more closely mimic the operation of a real ATM switch. As part of this work, the software switch has been extended to support both AAL0 cell and AAL5 packet switching and employs several queueing disciplines. Further, the switch has been modified to operate in a proportional time mode compatible with ProTEuS.

3.5.1 Architecture

Virtual ATM switching is implemented in the Linux kernel as a pseudo device driver, */dev/vswitch*. Again, this implementation technique is a common one because it allows simple and uniform user-level access to kernel facilities. User-level programs written to control the software switch configuration and operation do so through `ioctl` calls on the pseudo device.

Figure 3.12 shows the virtual ATM software switch architecture. In this example, the switch contains four ports; two are virtual ATM devices configured over a physical ethernet Network Interface Card (NIC), one is a virtual ATM device configured over a physical ATM NIC and one is itself a physical ATM NIC. This example demonstrates the versatility of the software switch, which can be configured to support *any* number of ATM interfaces as switch ports, both physical and virtual. This illustrates an important realization; the software switch *does not know or care that the ATM devices may be virtual*.

The virtual ATM switching layer, as the name suggests, accomplishes the task of switching cells from an incoming (Port, VC) onto an outgoing (Port, VC) based on the contents of a routing table. The switch provides backlog queueing or output queuing in multiple forms as discussed in Section 3.5.2 and can easily be extended to support others. Further, Q.Port has previously been modified to operate on the virtual ATM software switch fabric and can be used to provide signaling support[6].

Routing table entries are created implicitly full-duplex through a user-level `ioctl` call specifying an input (Port, VC) and output (Port, VC). This opens the incoming and outgoing virtual circuits* on the specified ports and makes an association between the

*Remember that if the port is associated with a virtual ATM device, then these are actually *virtual virtual circuits*.

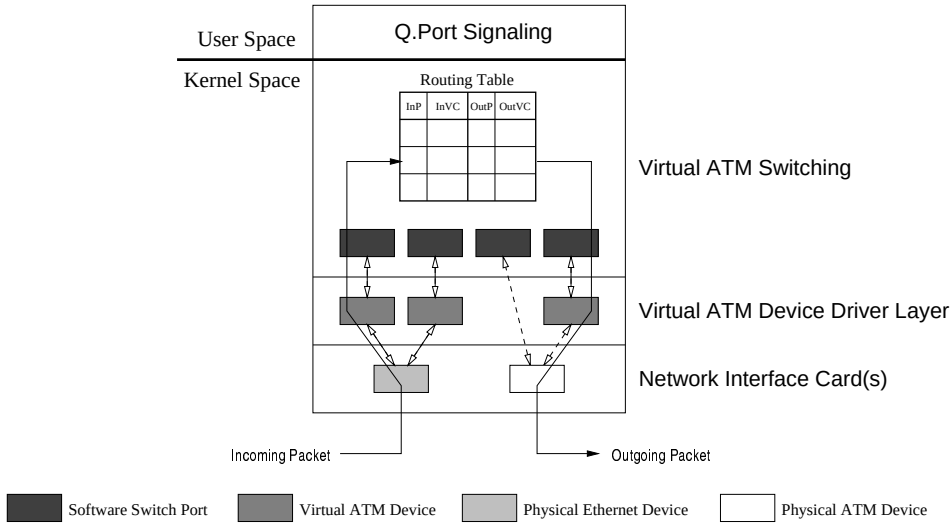

Figure 3.12: Virtual ATM Switching

two in the software switch routing tables. As part of the creation of a routing table entry, the *push* function pointer for each VC is set to the ATM switch receive function rather than the default Linux-ATM receive function. This is the hook that bypasses the Linux-ATM protocol stack and kernel implementation and allows the software switch to queue and switch ATM cells and packets.

3.5.2 Queueing

The virtual ATM software switch has been modified to support several queueing disciplines in an effort to provide another level of realism to the switch. As mentioned previously, the architecture of the switch is quite flexible and can be extended *ad infinitum* to mimic a real ATM switch as closely as the user desires. Such embellishments may include input queueing, crossbar operation, multicast routing, etc. However, the more complicated that queueing, routing and switching become, the slower the switch is likely to run, elongating the necessary real-time epoch.

The available queueing disciplines, as well as the maximum lengths of the related queues can be configured from user-level applications on a per-port basis, individually and independently, providing further flexibility. As will be discussed in the ensuing

sections, however, different disciplines may require different types of additional support to provide the quality of service for which their implementation was intended.

3.5.2.1 Shared Backlog Queueing

The original software switch was extremely primitive because the purpose for which it was initially intended did not require it to be overly faithful to real ATM switching. Furthermore, one of its driving motivations was the maximization of the achievable throughput, which, as mentioned earlier, is theoretically maximized when the switch software is at its simplest, including the queueing mechanisms[51].

Therefore, the first queueing discipline implemented, and still the default queueing discipline, is a single shared backlog queue whose maximum length is configurable, but which defaults to 300 cells (or packets). As depicted in Figure 3.13, all incoming cells from all ports are queued into a single First In First Out (FIFO) queue. When the queue is serviced, cells are switched from the queue in FIFO order and routed to their respective output ports.

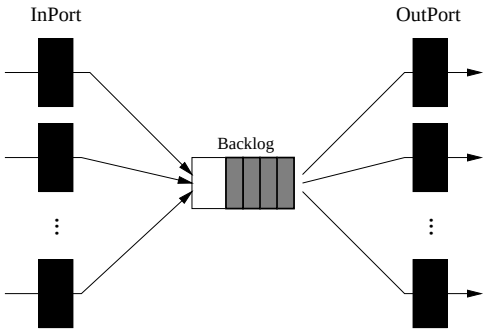

Figure 3.13: Shared Backlog Queueing in the Software Switch

3.5.2.2 Per-VC Queueing

Another queueing discipline implemented in the software switch is per-VC queueing in which every route across the switch is allocated its own FIFO queue at each port associated with the connection, whose maximum lengths are configurable, but which

default to 5000 cells (or packets). As cells arrive, they are routed to the output port of their connection and queued in FIFO order on the queue for that connection, as shown in Figure 3.14.

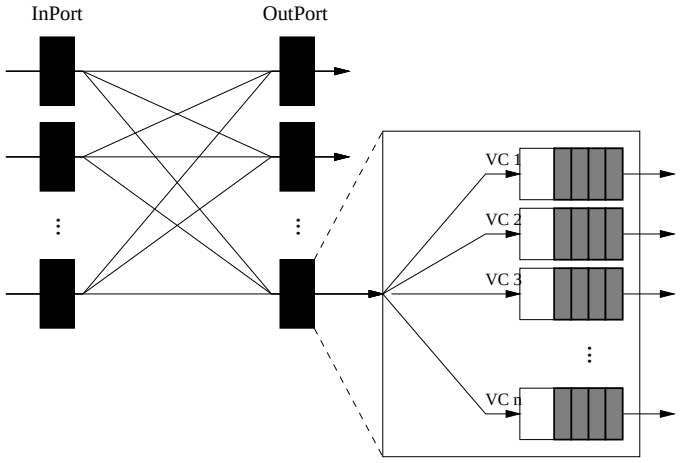

Figure 3.14: Per-VC Queueing in the Software Switch

This is a very complicated queueing mechanism, and was one of the major reasons that the credit-based congestion control mechanism discussed briefly in Section 3.3.2 was not adopted by the ATM forum; it *requires* per-VC queueing, while the explicit-rate approach does not necessitate any particular queueing discipline. In general, per-VC queueing can unnecessarily consume large amounts of memory and complicates both the queueing and switching mechanisms. When a particular queue is serviced, cells are switched from the queue in FIFO order. However, on a given port with many open VCs, switching between the queues for each VC is usually accomplished in a fashion reminiscent of round-robin service.

To accomplish the effect that this queueing discipline is intended to have, it is important that only one cell is switched per switching invocation, which is not the default software switch behavior. For instance, if instead, the switching mechanism switches all cells from each per-VC queue in each invocation, the behavior will be nearly equivalent to that of the shared backlog; the order of cell departure would differ, but all cells, in effect, would be switched at essentially the same time. Therefore, real-time opera-

tion of the switch, in which only one cell is switched per switching invocation, is the recommended implementation for this discipline.

3.5.2.3 Per-class Queueing

Per-class queueing is a common and practical queueing mechanism for several reasons. First of all, it provides per-port queueing, unlike the shared backlog, but is far less complicated than per-VC queueing. Further, it segregates the traffic by class which facilitates fulfilling some quality of service guarantees.

In per-class queueing, each port has a queue for each class of traffic; VBR, CBR, UBR and ABR. In addition, the software switch also creates a separate RM cell queue* to segregate resource management cells. As shown in Figure 3.14, incoming cells are routed to the output port of their connection where they are queued FIFO on the particular queue for their traffic class.

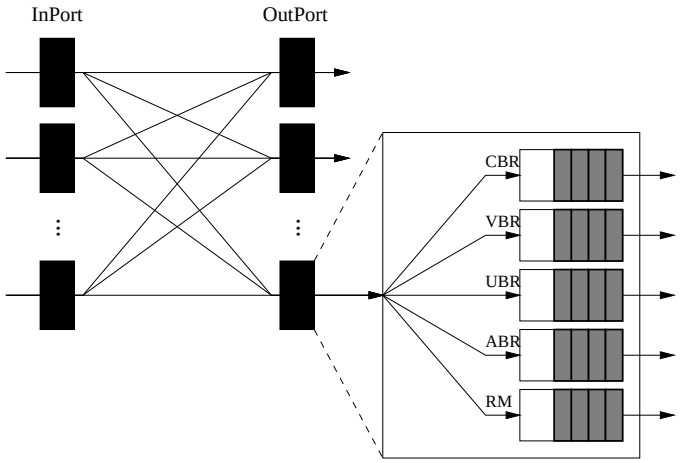

Figure 3.15: Per-Port Per-Class Queueing in the Software Switch

When the queue of a particular traffic class is serviced, cells are switched from the queue in FIFO order. To service cells between classes on each port, a weighted-round robin scheduling mechanism has been implemented, but because the focus of this work

*More generally, it is an Operation, Administration and Maintenance (OAM) queue, to which RM cells belong.

and its predecessor has been Available Bit Rate (ABR) service, current scheduling support is only available for VBR and ABR; cells of all classes are queued by the switch, but only the VBR and ABR (and RM) queues are serviced. The VBR and ABR queues are serviced with a VBR:ABR weight ratio of 200:1. The RM cells in the separate RM cell queue are serviced with the highest priority.

Again, real-time operation of the switch is the recommended mode of operation for this queueing discipline because best-effort service will, in effect, negate any quality of service imposed by the queueing and cell-switching mechanisms by simply switching cells as quickly as possible.

3.5.3 Switching a Cell or Packet

The process of switching a cell or packet in the virtual ATM software switch is typically driven by interrupts issued by the physical network devices on which the software switch ports are configured. However, with proportional time virtual ATM devices, queueing and switching are invoked by the KURT-Linux proportional time thread. The four fundamental interrupts issued by the network device are fairly standard and whether the physical devices are ethernet or ATM is largely inconsequential in this regard. As an example, Figure 3.12 on page 56 depicts a scenario that we will refer to during the following explanation.

Here, an incoming packet that is tagged as as a virtual network device packet arrives on a physical ethernet card causing the card to issue an interrupt to the operating system. In this interrupt service routine, the device driver is supposed to arrange for the DMA of the packet contents into system memory and relay the DMA information to the NIC.

When the NIC completes the packet transfer from card memory to system memory, it interrupts the system again to let it know that the transfer is complete. If the physical device is an ethernet device, as it is in Figure 3.12, the interrupt service routine in the device driver queues the cell and marks the network bottom half which services the packet at the discretion of the system*. When the network bottom half runs, it

*Bottom halves are discussed in slightly more detail in Section 3.7.1.

checks the ethernet packet type and recognizes the packet as one destined for a virtual network device, at which time it will invoke the packet type *func* function, passing it to the appropriate virtual ATM device.

If the physical device is an ATM NIC, the packet is passed directly to the virtual ATM device implementation or to the software switch queueing mechanism through the *push* function on the virtual circuit, or virtual virtual circuit as the case may be, on which the packet was received, bypassing the bottom half mechanism which does not exist for ATM devices in Linux.

In Figure 3.12, the virtual ATM device processes the packet and passes the cells up the virtual device stack, eventually invoking the *push* function on the virtual VC. This *push* function is the software switch receive function that routes the cells to their output ports and queues them. As mentioned, however, depending on the configuration of the virtual ATM device, the invocation of this *push* function may, or may not, be the result of the physical network interrupt.

When the switching routine is executed, cells that are ready to be switched, as determined by the queueing and switching mechanisms, are switched out by calling the *send* routine on the ATM device configured as the switch port. If the output port is a physical ATM device, then this is the conventional physical device driver *send* function. If the port is associated with a virtual ATM device, then this is the virtual ATM device *send* function which propagates the cells down the virtual ATM device stack and eventually to the device driver of either a physical ATM or ethernet device.

At this point, the device driver arranges for the DMA of the packet back to the memory of the NIC and informs it of the pending transfer. When the card completes the transfer of the packet into card memory, it interrupts the system making it aware that the packet has been copied from system memory, at which time, in the interrupt service routine, the device driver frees the memory that the packet occupied. When the packet has been successfully sent out on the network by the NIC, it interrupts the system one last time to let it know that the packet has been sent.

3.5.4 ABR Support

Because Available Bit Rate service is still, in many ways, in its infantile stages, support for it is yet absent in hardware or premature at best. Therefore, in order to study its behavior, it was necessary to model ABR service in software. As discussed in Section 3.4.3, ABR support is provided in end systems as part of the Proportional Time layer implementation on virtual ATM devices. Support for ABR on the virtual ATM software switch is present in two forms; the Explicit Proportional Rate Control Algorithm (EPRCA), and more recently, the Explicit Rate Indication for Congestion Avoidance (ERICA).

3.5.4.1 EPRCA

The Enhanced Proportional Rate Control Algorithm (EPRCA) is generally a *congestion reaction* scheme[50]. EPRCA operates at each output port of the software switch where it computes a Mean Allowed Cell Rate (**MACR**) using exponential weighted averaging and a **FairShare** as a fraction of the **MACR**:

$$MACR = \frac{15}{16}MACR + \frac{1}{16}CCR$$

$$FairShare = \frac{7}{8}MACR$$

Program 3.3 shows the fundamental algorithm executed at the reception of a **BRM** cell. EPRCA sets the **CI** and **NI** bits of the **BRM** cells based on the length of the ABR queue on the port on which the **BRM** cell was received. In the software switch, `LOW` is defined as 200 cells and `HIGH` is defined as 300 cells.

If the computed **FairShare** is greater the the **ER** in the **BRM** cell, then the **ER** is unchanged, otherwise the **ER** is set to the **FairShare**. This ensures that if **BRM** cells traverse more than one switch, the minimum of the **FairShare**s reaches the source, avoiding congestion at a bottleneck switch.

The advantages of EPRCA are that it is conceptually and computationally simple and allows both explicit *and* binary feedback. However its congestion detection algorithm, which is based on queue length, has been shown to be unfair. Sources starting

Program 3.3 ERPCA: At the Reception of a Backward RM Cell

```
1   MACR = (15 * MACR + CCR) >> 4;
2   FairShare = (MACR * 7) >> 3;
3
4   if (length >= HIGH) {
5       CI = 1;
6       NI = 0;
7   } else if ((length < HIGH) && (length > LOW)) {
8       CI = 0;
9       NI = 1;
10  } else if (length <= LOW) {
11      CI = 0;
12      NI = 0;
13  }
14
15  ER = MIN(ERInRMCell, FairShare)
```

late are treated differently than those that started early and are not allocated equal bandwidth.

3.5.4.2 ERICA

The Explicit Congestion Indication for Congestion Avoidance (ERICA) algorithm is generally a *congestion avoidance* scheme[32]. ERICA operates at each output port of the software switch where it periodically measures the total load, total ABR load, and the number of currently active ABR flows on the port. Using those, ERICA calculates a **FairShare** that can be allocated to each ABR stream, which is the target ABR capacity (usually 85 - 90% of the instantaneous available ABR capacity) divided by the number of active ABR VCs. Further, it calculates a load factor, **z**, to control the source rates. The load factor is the ABR input rate divided by the target ABR capacity.

$$FairShare = \frac{TargetABRCapacity}{NumActiveABRVCs}$$

$$z = \frac{ABRInputRate}{TargetABRCapacity}$$

Program 3.4 is the fundamental algorithm executed at the reception of a **BRM** cell. The **VCShare** is the current cell rate on the connection divided by the load factor,

MaxAllocPrevious is the **FairShare** from the last averaging interval and `DELTA` is 0.1.

As in EPRCA, if the **ER** in the **BRM** cell is less than the calculated **ER**, then the **ER** is unchanged ensuring that **BRM** cells traversing more than one switch relay the minimum **ER** to the source.

Program 3.4 ERICA: At the Reception of a Backward RM Cell

```
1   VCShare = CCR / z;
2
3   if (z > 1 + DELTA)
4       ER = MAX(FairShare, VCShare);
5   else
6       ER = MAX(MaxAllocPrevious, VCShare);
7
8   if ((ER > FairShare) && (CCR < FairShare))
9       ER = FairShare;
10
11  ER = MIN(ERInRMCell, ER, TargetABRCapacity);
```

ERICA possesses several distinguishing features. First, to minimize the feedback latency, ERICA monitors **FRM** cells for traffic metrics, but feedback is provided in the **BRM** cells. Therefore, the feedback is based on recent traffic and is more timely than if it were provided in the forward direction or based on current cell rate content from **BRM** cells. Further, the characteristics of the flows are measured over averaging intervals and ERICA gives only one feedback value per interval. By doing so, ERICA prevents the switch from giving multiple conflicting feedback indications in a short time interval. ERICA has a high throughput, exhibits low delay, keeps queue lengths near unity and exhibits a short time to reach steady-state.

3.5.5 Real-Time Switching

In its original form, the Virtual ATM Software Switch performed the switching of cells in a Linux bottom half. When cells arrived, the software switch marked the ATM software switch bottom half. When the bottom halves were run at the discretion of the system, the ATM switch bottom half switched out cells until the queues were emptied. While this mode of operation still exists in the switch and is adequate for some applications, a second mode, a real-time implementation, was added both to better provide

quality of service guarantees in the switch and to support ProTEuS ATM simulations.

The switch has been modified to operate under KURT-Linux control, which provides real-time support and the necessary accurate scheduling and timer mechanisms to service the output queues and provide the expected quality of service[52, 25, 53]. A real-time kernel thread operates the software switch switching functions, looping through and servicing each port, and is scheduled by an executive process through an explicit time line schedule, which is typically, but need not be, periodic. The switch operates in firm real-time mode and the cell service rate is specified by the executive process at the time the system is switched into real-time mode.

The obvious benefits of real-time switching are two-fold. First, real-time scheduling can provide the accurate periodic queue service that simulating a real ATM switch requires. Second, real-time guarantees can ensure an accurate, reliable switching rate that *cannot* be guaranteed by the Linux bottom half mechanism.

In ProTEuS ATM simulations, the switching function is invoked as part of the real-time thread controlling the proportional time simulation. During the computation interval of each real-time epoch, the PT layer calls the real-time switching function to provide the opportunity to switch out a cell on each port.

3.6 TCP/IP Timing Changes to Linux

As alluded to previously, the only changes necessary to utilize real system and application code in a ProTEuS simulation is to make it aware of virtual time, as opposed to real time. A protocol stack such as ATM or UDP requires little to no changes because the protocols are not heavily time-dependent. In ATM, for instance, all of the time sensitive operation, such as traffic shaping resides in the network hardware, not in the protocol stack.

TCP, however, includes several timing mechanisms to ensure reliable delivery of data. These include delayed acknowledgements and packet retransmission, just to name a couple. If these mechanisms are not modified to operate in virtual time, then the behavior of the protocol will be largely unpredictable and its operation is not likely

to be faithful to that of a real-world implementation. In such a case, the link layer of the system is operating in virtual time while the protocol layer is operating in real time. Therefore, because we wanted to be able to simulate IP over ATM sources in ProTEuS ATM simulations, it was necessary to virtualize the TCP/IP stack in Linux.

The TCP/IP stack in Linux uses Linux timers to control its time-sensitive implementation. For instance, when TCP packets are sent, a retransmit timeout timer is set that will cause TCP to retransmit the associated packet(s) when the timer expires, meaning that the corresponding ACK did not return in time. If the ACK does arrive before the timer expires, the timer is removed from the queue. These timers are are serviced by the Linux scheduler from a FIFO queue and are controlled by the PC timer chip. The fundamental unit in time in Linux is the *jiffy*, which represents 10ms of real time, and all timers are multiples of a jiffy. Once per jiffy, the Linux scheduler runs the timer queue and services all expired timers.

It was therefore necessary for ProTEuS to intercept the timer routines in the TCP/IP stack associated with virtual TCP flows and allow ProTEuS to service them in virtual time. First and foremost, a method was contrived to distinguish a virtual flow from a non-virtual flow so that normal TCP sessions are not disrupted by this modification to the protocol stack. Then, through the use of a simple C macro, all timer routine calls in the TCP/IP stack were modified to use the ProTEuS timer mechanisms when the flow is across a proportional time virtual ATM device and to use the conventional Linux timer mechanisms when the flow is non-virtual. ProTEuS implements its own timer queue identical to that of standard Linux, and services it each *virtual jiffy*. Further, using a similar macro, several places in the stack where TCP was capturing and operating on timestamps* were modified to use the virtual timestamp value, in units of virtual jiffies, as opposed to the real timestamp value, in units of Linux jiffies. This converts the timing control of virtual TCP flows to think in virtual time instead of real time.

The changes to the TCP/IP protocol stack encompassed probably less than 50 lines of source code and were completed in less than a week. Most of that time was spent

*Such as in the estimation of round-trip time which TCP uses to vary its retransmission timeout value.

simply traipsing through the TCP/IP stack trying to find the calls to the Linux timer routines and uses of the Linux jiffy counter, some of which were less than apparent.

This metamorphosis is a significant example of the strength of ProTEuS. With fewer than 50 changed lines of source code and with minimal effort persisting less than a week, ProTEuS simulations were able to use the *real TCP/IP stack from the Linux operating system*.

3.7 Remaining Challenges

Once the major performance constraints of ProTEuS were addressed and corrected, it was still clear that there were additional issues affecting the performance of our distributed computations. However, these newly identified factors constraining performance are at the system level and manifest in the form of scheduling jitter. Because KURT-Linux is a *firm* real-time system, guarantees cannot always be made for *every* invocation of real-time processes. KURT-Linux is, at times, subject to the greedy nature of other parts of the system and can occasionally experience latencies of up to several hundred μs; that is, a few times per second. Of course, if the period of a computation is in the range of several milliseconds, this is unlikely to have much impact (depending on the utilization of the epoch). However, if the period is on the order of hundreds of microseconds, it can have a significant impact on efficiency.

3.7.1 Linux Bottom Halves

The largest source for jitter is the bottom half execution in Linux[2, 24]. In an effort to minimize the amount of work done in an Interrupt Service Routine (ISR), many interrupt handlers will employ a bottom half, which they "mark" during the ISR. Only critical work is done during the ISR and marking the bottom half is a notification to the system that it needs to run the bottom half for this interrupt, whose duty it is to complete the work. Bottom halves are run at the discretion of the system and are executed all-or-nothing. That is, when they are run, they are all run in succession regardless of the length of time it takes to complete.

Some bottom halves, such as the network bottom half and the SCSI disk bottom half can introduce significant jitter because they are allowed to run for significant lengths of time, much of it with interrupts disabled. This is because in a standard Linux system which uses a 10ms periodic interrupt to keep time, occasional distortions of a few hundred microseconds are largely insignificant. Some work has been done to reduce the execution times of specific bottom halves, but it is clear that a more general approach is necessary to achieve significant improvement in the reduction of scheduling jitter[24]. One method currently being considered is to introduce a kernel thread whose sole purpose is to run bottom halves. That thread could then be scheduled along with all other threads, explicitly controlling when bottom halves execute, thus reducing the jitter they currently introduce into the scheduling of real-time processes.

3.7.2 Clock Rate and Phase Synchronization

Another issue with synchronized distributed computation is that KURT-Linux and UTIME depend on somewhat inaccurate timer chips to keep time, as does conventional Linux, that are sensitive to environmental changes which affect the rate at which they increment. This is a known problem and is a motivation for the Network Time Protocol (NTP)[37, 38]. The problem is that if each component of a synchronized distributed computation has a different notion of either relative or absolute time it will affect the synchronization of the distributed computation. We assist the machines in incrementing time at the same rate as addressed by Hill where the NTP standard is used to calibrate each machine's notion of the number of CPU cycles per second[24]. This has been shown to be a very effective method to keep clocks incrementing at very close to the same rate.

However, it does not affect their synchronization on the absolute time line, which is subject to the granularity of NTP. On a local area network (LAN), NTP claims to be able to synchronize to roughly within a millisecond. Whether or not this is adequate absolute synchronization will depend on the length of the epoch and its slack time component. Relatively simple and notably accurate clock synchronization, both in phase and frequency, has been shown by Menon to be achievable when the physical

network is reliable and low latency and the time-keeping mechanism is kept as close to the network hardware as possible[35].

Figure 3.16 shows a scenario with two distributed components where the absolute time references are off by approximately half the length of an epoch. Depending on the epoch length and the ratio between the computation time and the slack time (epoch utilization), it is easy to visualize how this interleaving of epochs can have unfortunate consequences, even the *propagation* of missed epochs around the system. This is depicted in Figure 3.16 where each distributed component misses roughly every third epoch, all of which, except the first, are the direct result of the other component missing a previous epoch. Because our driving application often requires very fine temporal granularity, we are considering various ways of improving the synchronization of distributed components on the absolute time line, including re-enacting the simple master-slave synchronization method investigated by Menon[35].

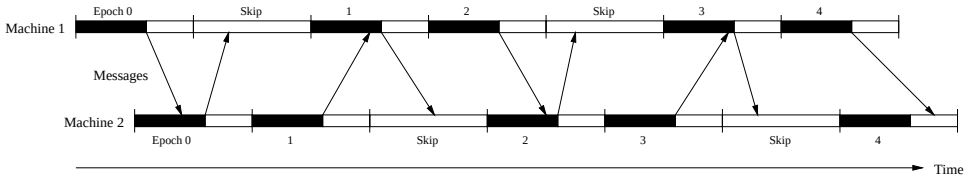

Figure 3.16: Epoch Interleaving Due to Insufficiently Synchronized Clocks

3.8 NetSpec

As the size of ATM networks grow, so does the complexity of the simulations necessary to effectively evaluate them. This brings about a need for a simple, effective, reproducible and extensible way to describe an experiment and have the setup, execution and the subsequent tear-down be automated as much as possible.

NetSpec is a distributed network performance evaluation tool developed at the University of Kansas that uses a simple, block-oriented scripting language to describe an experiment[30, 31]. The general architecture of the NetSpec system is depicted in Figure 3.17. The input script is parsed and blocks of the experiment are distributed to

NetSpec *daemons* as directed by the script. The daemons are processes that execute on target hosts to perform the tasks outlined in the script. These daemons, which can be written to perform just about any task desired, catch and execute the instructions that NetSpec sends them.

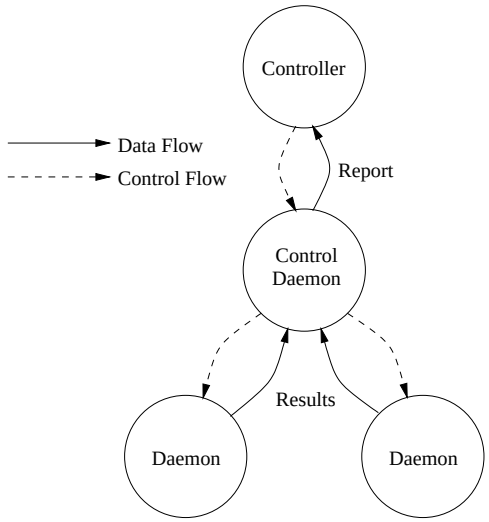

Figure 3.17: The NetSpec Architecture

Each NetSpec daemon traverses several phases of execution. The NetSpec control daemon, *nscntld*, invokes these functions on each of the daemons in the input script. All of the phases must be defined in the daemon, but none is actually required to do anything except send an acknowledgement back to the control daemon to tell it that the execution of the particular phase has completed; not doing so runs the risk of NetSpec hanging, waiting for an acknowledgement that never arrives. Other than the acknowledgements, the daemons have complete autonomy regarding what they do during each phase. Each phase is listed below with its intended use:

- *Initialize*: Initialize the daemon.
- *Setup*: Allocate any necessary resources.
- *Open*: Establish any necessary connections.
- *Run*: Perform the requested actions.
- *Close*: Close any open connections.

- *Finish*: Finish any unfinished portion of the requested work.
- *Report*: Generate a report of the results.
- *Teardown*: Release any allocated resources.
- *Kill*: Exit.

Users provide NetSpec an input script that describes an experiment in a manner that is meaningful to the appropriate NetSpec daemons. NetSpec parses the script and contacts the daemons on hosts, typically remote ones, as the script dictates. The daemons listen on TCP ports waiting to receive NetSpec commands and an instruction block from the input script, which they receive and parse between the *initialize* and *setup* phases. Before each daemon exits, it is granted the opportunity to send a report back to the user during the *report* phase. The report can contain anything the daemon wishes to relay back to the user, if anything. Upon completion, the daemon processes are killed and before it exits, NetSpec presents the information reported from the various daemons to the user.

Furthermore, the daemons need not be started by hand and left running like conventional daemon processes. NetSpec is set up such that *inetd*, the Internet superserver daemon, will activate the NetSpec daemon, *netspecd*, on a remote host when a connection is established on a recognized NetSpec port*. Ironically, *netspecd* is essentially the NetSpec equivalent of *inetd* and receives information about the NetSpec daemon to invoke and the port on which to invoke it. If *netspecd* can locate information about the requested daemon in its configuration file, */etc/netspec.conf*, it will execute it, requiring no user intervention.

3.8.1 NetSpec Scripts

Program 3.5 contains an example NetSpec script used to relay ATM traffic between two nodes across a virtual ATM software switch in a ProTEuS ATM network simulation. The script contains a super-block that describes the general manner in which the

*This operates much in the same way, for instance, that *telnetd* is activated by *inetd* when a connection is established on port 23.

experiment should proceed; in this example, *cluster*. The valid super-block types are listed below:

serial The individual blocks of the super-block are executed in the order in which they appear in the script, serially; i.e., the first block is executed in its entirety before the second even begins execution, and so on.

parallel The individual blocks of the super-block are run in parallel asynchronously; i.e., they all begin execution roughly at the same time, but no guarantees are made that their execution will remain synchronized in any way.

cluster The individual blocks of the super-block are executed in parallel synchronously; i.e., they all begin execution of each NetSpec phase at the same time. For example, no daemon will begin the *run* phase before *all* daemons have finished the *open* phase, and so on.

Inside the super-block are blocks that each describe some portion of the overall experiment. Each block in a super-block specifies the name of the NetSpec daemon for which the block is destined and an optional host name and port number on which to contact the daemon. The target machine will default to *localhost* if not specified and a port number for communication will be assigned if not provided.

Inside each block is the list of commands to be passed to the specified daemon. Although each daemon can define its own semantics and syntax, it must satisfy certain restrictions imposed by NetSpec. NetSpec parses each block, not for content, but to ensure that it conforms to its language syntax, which basically consist of:

```
ID = ID [(ID[=value] [, ID[=value] ...])];
```

where the square brackets indicate optional specification.

The argument list in parentheses is optional, but if it exists, the arguments should be identifier or identifier=value pairs separated by commas, and each statement *must* end in a semicolon. Identifiers are compelled to start with a letter and may consist of letters, numbers and underscores exclusively. Values have many different allowed

Program 3.5 An Example NetSpec Script

```
1   cluster {
2       vdev testbed18 {
3           vdev = atm (V1, physical=atm, itf=0, vc=60, localmac="0020ea000776",
4               remotemac="0020ea000777", stack=sarpt, delay=10);
5           pt = config (linerate=8000, wait=0, stop=50000);
6           kurt = pt (repeat=5000, schedule="pt_sched_150",
7               path="/users/shouse/ProTEuS/KURT/user_progs");
8       }
9
10      test testbed18 {
11          type = full (blocksize=2048, duration=5);
12          protocol = atm (PVC, vc=V1.0.50, class=ABR, pcr=8000, ccr=1000);
13          own = testbed18;
14          peer = testbed20;
15      }
16
17      vdev testbed19 {
18          vdev = atm (V1, physical=atm, itf=0, vc=60, localmac="0020ea000777",
19              remotemac="0020ea000776", stack=sarpt, delay=10);
20          vdev = atm (V2, physical=atm, itf=0, vc=65, localmac="0020ea000779",
21              remotemac="0020ea000778", stack=sarpt, delay=10);
22          pt = config (linerate=8000, wait=0, stop=50000);
23          kurt = pt (repeat=5000, schedule="pt_sched_150",
24              path="/users/shouse/ProTEuS/KURT/user_progs");
25      }
26
27      vswitch testbed19 {
28          vswitch = port (P1, itf=V1, queueing=perclass);
29          vswitch = port (P2, itf=V2, queueing=perclass);
30          vswitch = connection (invc=P1.0.50, outvc=P2.0.50, aal=0, class=ABR);
31      }
32
33      vdev testbed20 {
34          vdev = atm (V1, physical=atm, itf=0, vc=65, localmac="0020ea000778",
35              remotemac="0020ea000779", stack=sarpt, delay=10);
36          pt = config (linerate=8000, wait=0, stop=50000);
37          kurt = pt (repeat=5000, schedule="pt_sched_150",
38              path="/users/shouse/ProTEuS/KURT/user_progs");
39      }
40
41      test testbed20 {
42          type = full (blocksize=2048, duration=5);
43          protocol = atm (PVC, vc=V1.0.50, class=ABR, pcr=8000, ccr=1000);
44          own = testbed20;
45          peer = testbed18;
46      }
47  }
```

types including integer, string, character, real, IP address, ATM address, identifier, boolean, etc.

3.8.2 NetSpec Daemons

One of the most attractive features of NetSpec is that it is easily extended. One only needs to write a new daemon or extend an existing one to get new functionality, and it requires *no* changes to NetSpec itself. The daemons link with a library of common NetSpec daemon routines that serves as the glue between the daemon and NetSpec. Since the language that each daemon utilizes to describe the functions that it will perform can be, and probably should be, unique for each daemon, each daemon requires a Lex & Yacc parser to parse the instruction block from the input script that NetSpec provides it. Furthermore, each daemon has to implement each of the phases described in Section 3.8, but what the daemon does in each of these phases, if anything, *is completely up to the daemon*. This feature makes it almost infinitely extensible; you can write a daemon to do pretty much anything.

3.8.2.1 The Virtual Network Device Daemon: *nsvdevd*

The primary method for configuring virtual network devices is command line based. This is acceptable, perhaps even convenient, when the experiment is small to moderately sized. However, in large-scale experiments, it becomes oppressive and impractical. Therefore, the virtual network device NetSpec daemon, *nsvdevd*, was created as part of this work to ease the burden of experiment specification, rendering it as simple, automated and reproducible as possible.

There are currently four accepted commands; one for configuring virtual network devices*, one for proportional time configuration, one for real-time support for proportional time and one for proportional time ATM Reference Traffic Source (ARTS) support. The commands may come in any order and inconsistencies will be detected

*Because the development of virtual ethernet devices is as yet ongoing, *nsvdevd* does not yet support them, but the syntax of the language has been designed with their eventual matriculation in mind.

during the setup. Program 3.6 shows some example NetSpec commands for configuring devices, ARTS streams, and proportional time and real-time operation.

Program 3.6 Examples of Virtual Network Device NetSpec Commands

```
1   vdev = atm (V1, physical=atm, itf=0, vc=50, localmac="0020ea000778",
2       remotemac="0020ea000779", stack=SARPT, delay=50);
3
4   pt = config (linerate=365000, wait=50, tolerance=100, stop=365000);
5
6   kurt = pt (repeat=5000, schedule="pt_sched_250", start=952059105,
7       path="/users/shouse/ProTEuS/KURT/user_progs");
8
9   pt = arts (repeat=5000, schedule="StarWars-8000.sched", vc=V2.0.40,
10      path = "/users/shouse/ProTEuS/atm/vatm");
```

Creating a virtual network device creates what the Linux kernel *believes* is an actual network device, and requires numerous parameters, both to configure the behavior and structure of the virtual device and to specify the physical network resources that will be used to support it. Below is a brief explanation of each of the virtual network device options.

V*num* Assigns the virtual network device a temporary identifier. This is necessary to identify the device throughout the script because the kernel assigns interface numbers as devices are registered. Therefore, to allow the virtual network devices to be subsequently identified in this script (and by other NetSpec daemons), we assign a temporary identifier to each virtual network device. This *must* be the first specification in the list of options. (*Required*)

Physical Specifies the physical support network [eth, atm] for the virtual network device. That is, the type of physical device with which it will be associated. Because virtual network devices have different options depending on the physical network, this specification *must* be the second specification in the list of options so that the list of permissible options can be identified by the parser. (*Required*)

ITF This is the interface number of the physical network device. Ethernet devices are identified by a name, *ethX* where *X* is an integer, while ATM devices are identified by a integer ATM interface ID. (*Required*)

Type Applies to virtual network devices over ethernet only and should always be MAC. MACRCA is used specifically in the RDRN project at the University of Kansas and is essentially useless elsewhere. (*Required if physical ethernet*)

VC Applies to virtual devices over ATM only, and is the ATM virtual circuit over which traffic for this virtual network device will be multiplexed. (*Required if physical ATM*)

LocalMAC A fictional MAC address that will be assigned to this virtual network device. (*Required*)

RemoteMAC A fictional MAC address that this virtual network device associates a point-to-point connection with. This MAC address should be the local MAC address on another virtual network device in the experiment. (*Required*)

Stack This is the stack that will be created on the virtual device. Components currently include Segmentation and Reassembly (SAR), Data Link Control (DLC) and Proportional Time (PT) that can be combined in particular supported configurations [SAR, DLC, SARDLC, SARPT]. (*Required*)

Delay Sets the simulated link delay on proportional time virtual ATM devices. This option is only valid on virtual ATM devices configured with a Proportional Time layer. The default simulated delay is zero. (*Optional*)

Support for ProTEuS resides in the virtual network device implementation and is available whenever a virtual network device is configured with a Proportional Time layer. Several parameters of the proportional time simulation can be configured as briefly explained below.

LineRate The simulated line rate for the simulation in cells per second. The default line rate is 365000 cells per second (~OC-3). (*Optional*)

Wait The time to wait (in μs) for packets that have not yet arrived before bailing out of the receive phase. The default wait time is zero. (*Optional*)

Tolerance The distributed synchronization tolerance. This parameter is automatically calculated by proportional time virtual ATM devices as the minimum of the local simulated link delays on this physical host, however a specific tolerance can be set through this mechanism. *Note: A configured tolerance greater than the minimum simulated delay can cause erroneous ATM simulation results because ProTEuS will no longer be able to ensure that all data with a bearing on the current epoch being executed has actually arrived. See Section 3.4.3.1, page 45 for more information.* (*Optional*)

Stop The stop time of the simulation in epochs. The default stop time essentially amounts to never (~12.5 days at a period of 250μs). (*Optional*)

Of course, any virtual network devices on a host that are configured with a Proportional Time layer require real-time support in the form of KU Real-Time modifications to Linux (KURT-Linux) to execute the simulation. KURT-Linux invokes the proportional time implementation on a static periodic schedule provided in the real-time parameters described briefly below.

Schedule The filename of the real-time schedule. Schedules are usually, but need not be, periodic and are created by a utility provided in the KURT-Linux distribution. (*Required*)

Repeat The number of times to repeat the schedule. The default is one. (*Optional*)

Start The time at which to begin the real-time schedule, as expressed in UNIX time format (seconds only). The default behavior is to start as soon as possible. (*Optional*)

Path The path to the KURT-Linux executables and schedules. The default path is */usr/local/bin*. Schedules are expected to live in a subdirectory of **path** called *schedules/*. (*Optional*)

Virtual ATM devices configured with a Proportional Time layer also contain support for ARTS sources used to mimic or reproduce specific ATM traffic sources based on an explicit cell schedule. Cells are dispatched by the PT layer in accordance with a

schedule that specifies the epochs in which to send cells. Below is a brief explanation of each of the ARTS parameters.

Schedule The filename of the ARTS schedule. Schedules are created by utilities provided with the ProTEuS distribution. (*Required*)

Repeat The number of times to repeat the schedule. The default is one. (*Optional*)

VC The VITF.VPI.VCI on which to send the ARTS traffic. Because ARTS functionality is provided by proportional time virtual ATM devices, this *must* be a VC on a PT virtual ATM device (a VVC). The interface should therefore begin with a leading *V* to signify a virtual ATM interface, followed by the virtual ATM device ID. (*Required*)

Path The path to the ARTS executables and schedules. The default path is */usr/local/bin*. Schedules are expected to live in a subdirectory of **path** called *schedules/*. (*Optional*)

When the *nsvdevd* parser is invoked on the script that NetSpec provides, it fills in data structures that it will need to accomplish the requested tasks. After parsing is complete, it loops through all virtual network device specifications and creates them trough an `ioctl`. If proportional time configuration has been provided, it is also configured during this phase. During the *run* phase, all specified ARTS sources are configured. Because the ATM interfaces are virtual, the daemon must map the virtual device ID to the kernel-assigned ATM interface number by querying the kernel. This is also the phase where real-time control of proportional time is initiated, if it has been specified.

Upon reaching the *teardown* phase, the daemon again loops through the same data structures created while parsing the script to restore the system to its original state. By the time the daemon exits, the machine state is exactly as it was before the experiment ran, allowing another experiment to run on it at any time without requiring a re-boot.

The *nsvdevd* report consists of the state of the hosts network devices at the end of the experiment and the ProTEuS status. This includes the device statistics, and the

proportional time configuration and results. The report is obtained through both the Linux */proc* interface and a system call in the PT implementation.

/proc/atm/devices Shows the status of the host ATM devices; cells transmitted, received, erred, etc.

sys_pt_get_config() Retrieves the PT configuration; the simulation duration, total number of epochs and missed epochs, configuration information, etc.

3.8.2.2 The Virtual ATM Software Switch Daemon: *nsvswitchd*

For users of the virtual ATM software switch, setting up an experiment can be an arduous and monotonous task when the size of the experiment is very large. The primary method for configuring the software switch is command line based, which is fine for small to moderately sized experiments, but cumbersome and impractical for large ones. Therefore, the virtual ATM software switch NetSpec daemon, *nsvswitchd*, was created as part of this work to employ a simple, automated and reproducible way to configure the software switch and ease the burden of experiment specification.

There are three commands; one for configuring virtual ATM software switch ports, one for creating connections across the switch and one for specifying the real-time operation of the switch. The commands may come in any order and inconsistencies will be detected during the setup. Program 3.7 shows some example NetSpec commands for configuring ports, connections and real-time support.

Program 3.7 Examples of Virtual ATM Software Switch NetSpec Commands

```
1  vswitch = port (P3, itf=V2, queueing=perclass, class=CBR, length=300);
2
3  vswitch = connection (invc=P1.0.50, outvc=P3.0.40, aal=0, class=ABR);
4
5  kurt = switch(repeat=500, schedule="rt_sched_125", start=952059001,
6      path="/users/shouse/ProTEuS/KURT/user_progs");
```

In previous versions of the software switch, it discovered the ATM devices on the host machine during boot and automatically configured them as software switch ports. However, with the advent of virtual ATM devices, this is no longer a desired feature

because not all ATM devices will be available to be automatically discovered by the switch at boot time. Therefore, the need has arisen to explicitly configure devices as ports. Below is a brief explanation of each of the port parameters.

P*num* Assigns the switch port a temporary identifier. This is necessary to identify the port throughout the script because the software switch assigns port numbers as devices are configured. Therefore, to allow the port specifications to appear in any order in the script, we assign a temporary identifier to each port. This *must* be the first specification in the list of parameters. (*Required*)

ITF Specifies the ATM interface that will support the switch port. Virtual ATM interfaces are identified by a leading *V* followed by the virtual ATM ID assigned in the *nsvdev* script. This will cause the switch daemon to map from the virtual ATM ID to the ATM interface. (*Required*)

Queueing Specifies the queueing discipline [backlog, perVC, perclass] to be assigned to this port. The default queueing discipline on all ports is the shared backlog. (*Optional*)

Class Specifies the ATM traffic class [CBR, UBR, ABR, VBR] to which the length is applied if the discipline is per-class. The default is that the length is applied to the queues of *all* traffic classes on the port. (*Optional*)

Length Specifies the length of the associated queue(s). The default length of the shared backlog is 300 cells and all other queues default to 5000 cells. (*Optional*)

Connections across the software switch must be set up manually through an `ioctl` call to make sure that the VC becomes associated with the software switch and not the normal Linux-ATM stack (through the ATM *push* function, as described in Section 3.5). Any VCs not created this way, even those on ATM devices configured as software switch ports, pass through the normal Linux-ATM stack and bypass the switch. SVCs can be setup across the software switch using Q.Port software that has been modified to work with the virtual ATM software switch fabric and need not be specified here. Further, connections are implicitly full-duplex, so there is no need to create the same

connection in both directions*. Below is a brief explanation of each of the connection parameters.

InVC The incoming [port.VPI.VCI]. Ports are identified by their IDs, beginning with a *P*, assigned by the port configuration command. (*Required*)

OutVC The outgoing [port.VPI.VCI]. Ports are identified by their IDs, beginning with a *P*, assigned by the port configuration command. (*Required*)

AAL The AAL [0, 5] for this connection. The software switch can switch packets (AAL5) or cells (AAL0). If the connection is AAL0, packets will not be reassembled by the ATM device. The default AAL is AAL0. (*Optional*)

Class The traffic class [CBR, UBR, ABR, VBR] for this connection. The default traffic class is UBR. (*Optional*)

The software switch operates in one of two modes, which is a kernel configuration option set at compile time. The default mode of operation is what we term best-effort, where switching is invoked through the Linux bottom-half implementation. A more recent mode of operation is real-time switching, where the switching routine is a real-time kernel thread invoked periodically by KURT-Linux. Below is a brief explanation of each of the real-time switching parameters that are available if the real-time mode of operation has been built into the kernel.

Schedule The filename of the real-time schedule. Schedules are usually, but need not be, periodic and are created by a utility provided in the KURT-Linux distribution. (*Required*)

Repeat The number of times to repeat the schedule. The default is one. (*Optional*)

Start The time at which to begin the real-time schedule, as expressed in UNIX time format (seconds only). The default behavior is to start as soon as possible. (*Optional*)

*The second call would fail when it finds that the VC is already in use.

Path The path to the KURT-Linux executables and schedules. The default path is */usr/local/bin*. Schedules are expected to live in a subdirectory of **path** called *schedules/*. (*Optional*)

When the *nsvswitchd* parser is invoked on the block that NetSpec passes it, it fills in data structures that it will need to accomplish the requested tasks. In the *setup* phase, it loops through all port specifications and associates the ATM devices with software switch ports through an `ioctl`. If the interfaces are virtual, the daemon must map the virtual device ID to the kernel-assigned ATM interface number by querying the kernel. It maintains a mapping from user specified port identifiers to software switch assigned port numbers. Finally, it loops through all connections in the script and creates each one using the port-ID mapping. During the *run* phase, if specified, real-time switching is initiated.

Upon reaching the *finish* phase, the daemon loops through the same data structures to restore the system to its original state. By the time the daemon exits, the virtual ATM software switch setup is exactly as it was before the experiment ran, allowing another experiment to run on it at any time without requiring a re-boot.

The *nsvswitchd* report consists of the state of the switch at the end of the experiment. It includes the switch configuration, statistics and the current status of the queues and is obtained through the Linux */proc* interface.

/proc/atm/vswitch Shows the status of the switch ports and connections; the number of cells switched, dropped, peak cell rates, etc.

/proc/atm/backlog Shows the status of the backlog queue; the current length of the queue, number of cells dropped, etc.

/proc/atm/pervc Shows the status of any configured per-VC queues; the current lengths of the queues, number of cells dropped, etc.

/proc/atm/perclass Shows the status of any configured per-class queues; the current lengths of the queues, number of cells dropped, etc.

3.8.2.3 The Test Daemon: *nstestd*

People trying to evaluate the performance of a network, regardless of the particular protocol, need a simple way to specify and generate network traffic. For small networks, simple techniques, such as `ttcp` at the command line may do. However, this limits the characteristics of the generated traffic to a greedy source and bounds its verity in practical application. Users will typically want to emulate real traffic sources such as bursty, telnet, WWW, etc. The NetSpec Test daemon is a candidate application capable of assuming such a role, however, it does not support the ATM protocol that is necessary for ProTEuS ATM network simulations. As part of this work, modifications were made to the existing daemon to add support for ATM and Classical IP over ATM (CLIP).

The Test daemon was implemented as part of the original NetSpec work, however, it did not support the ATM protocol[30, 31]. One reason for this is that not all operating systems that NetSpec runs on have ATM support, where TCP and UDP are omnipresent. In the case of Linux, the ProTEuS platform, ATM support is in many ways still in its infantile stages, and therefore NetSpec support for the ATM protocol in general has not been a high priority. Rather, the Test daemon concentrated on the UDP and TCP protocols and support for the aforementioned emulated traffic types.

There are five accepted commands; one for setting the test type, one for specifying the protocol, two for setting the peer and local IP addresses, and one for configuring CLIP over ATM. The commands may come in any order and inconsistencies will be detected during the setup. Program 3.8 shows some example NetSpec commands for configuring the ATM protocol and CLIP over ATM*.

Program 3.8 Examples of ATM Test NetSpec Commands

```
1  protocol = atm (PVC, vc=V2.0.70, class=abr, ccr=1000, pcr=15000);
2
3  clip = PVC (itf=atm0, own=10.0.0.1, peer=10.0.0.2, vc=0.0.60,
4      path="~/ProTEuS/atm/arpd");
```

*The specifics of each preexisting test type and protocol can be found in [30, 31] and will not be discussed here. Instead, the focus is on the added ATM functionality.

The protocol line informs the daemon that the protocol is ATM and sets up the VC on which the traffic will be transmitted as well as the traffic class for the connection and its parameters. TCP and UDP have a similar specification, but with different sets of protocol options. Below is a brief explanation of each of the ATM protocol parameters.

PVC/SVC Specifies whether the traffic should traverse a Permanent Virtual Circuit (PVC) or a Switched Virtual Circuit (SVC). Although the language syntax includes SVCs, currently, only PVCs are supported. Because the protocol options differ depending on the type of VC, this specification *must* be the first specification in the list of options so that the list of permissible options can be identified by the parser. (*Required*)

VC The ITF.VPI.VCI on which to perform the test. Virtual ATM interfaces are identified by a leading *V*, followed by the virtual ATM ID set upon virtual ATM device configuration in the *nsvdev* script. (*Required*)

Class The ATM traffic class of the VC [CBR, UBR, ABR]. The traffic class defaults to UBR. ABR is available only if the ATM interface is a proportional time virtual ATM device. The default ATM traffic class is UBR. (*Optional*)

AAL The AAL of the connection. Currently, only AAL5 connections are allowed. (*Optional*)

SDU The maximum AAL5 packet length. The SDU defaults to the ATM maximum AAL5 packet length, 65535 bytes. (*Optional*)

PCR The Peak Cell Rate (**PCR**) of the connection. (*Required if CBR or ABR*)

MCR The Minimum Cell Rate (**MCR**) of an ABR connection. The default **MCR** is zero. (*Optional*)

CCR The Initial Cell Rate (**CCR**) of an ABR connection. (*Required if ABR*)

NRM The number of data cells between **FRM** cells on an ABR connection. The default **NRM** is 31. (*Optional and only if ABR*)

Classical IP over ATM allows the use of conventional IP services over ATM interfaces. From the user-level, the differences are transparent; *ssh*, *telnet*, *ping*, etc. all function in the same fashion from the user perspective, but the actual data transport is over an ATM network. However, some setup and configuration is necessary to allow ATM Address Resolution Protocol (ARP) to resolve IP over ATM mappings. The available configuration options and parameters are briefly explained below.

ITF The unique* ATM-o-IP interface. ATM-o-IP interfaces are of the form *atmX* where *X* is an integer. (*Required*)

Own The IP address to be assigned to the ATM-o-IP interface. (*Required*)

VC If configuring CLIP using PVCs, this is the PVC through which to contact the peer IP entity. Virtual ATM interfaces are identified by a leading *V*, followed by the virtual ATM ID set upon virtual ATM device configuration in the *nsvdev* script. (*Required if PVC*)

Peer If configuring CLIP using PVCs, this is the peer IP address reachable through the specified PVC. (*Required if PVC*)

Netmask If configuring CLIP using SVCs, this is the netmask used by ATM ARP to determine which ARP server to contact. (*Required if SVC*)

ARPsrv If configuring CLIP using SVCs, this is the ATM address of the ARP server. (*Required if SVC*)

When the *nstestd* parser is invoked on the block that NetSpec passes it, it fills in data structures that it will need to accomplish the requested tasks.

In the *setup* phase, it first configures CLIP support if it has been specified. It then opens an ATM socket and sets the specified Quality of Service (Qos) parameters, including the traffic class and associated cell rates and the *open* phase results in the binding of the virtual circuit (or virtual virtual circuit). If the interfaces are virtual in either the CLIP VC or the ATM protocol VC, the daemon maps the virtual device ID to the

*The same ATM interface cannot exist twice simultaneously on the same physical host.

kernel-assigned ATM interface number by querying the kernel. During the *run* phase, the actual data transmission is conducted and the subsequent *close* phase severs all open connections.

The *nstestd* report consists of both the setup and results of the test, including the test and protocol configuration, test duration, blocks transferred, and the throughput achieved.

3.8.2.4 Summary

The increasing size of network performance evaluation experiments, including those of ProTEuS, brings about a need for a simple, effective, reproducible and extensible way to describe an experiment and have the setup, execution and the subsequent tear-down be automated as much as possible.

NetSpec is a distributed network performance evaluation tool that uses a simple, block-oriented scripting language to describe an experiment. Users provide NetSpec an input script that describes an experiment in a manner that is meaningful to the appropriate NetSpec daemons. The script is parsed and blocks of the experiment are distributed to NetSpec daemons that execute on target hosts to perform the tasks outlined in the script. The daemons process their tasks in phases of execution and report the results back to the user upon completion of the experiment.

Due to the size of ProTEuS ATM simulations and the complexity of their associated setup and tear-down, command line and shell script based control is often cumbersome and impractical. Therefore, ProTEuS utilizes NetSpec to control proportional time simulations of ATM networks through the use of the NetSpec virtual network device daemon, the NetSpec virtual ATM software switch daemon and the NetSpec test daemon.

Chapter 4

Evaluation

This chapter evaluates the Proportional Time Emulation and Simulation (ProTEuS) platform, specifically as it pertains to the simulation of ATM networks. The performance and verity of ProTEuS ATM network simulation is compared with two popular alternative network simulation tools; the Block Oriented Network Simulator (BONeS) and Georgia Tech Time Warp (GTW). BONeS is a sequential discrete event simulator from Cadence Design Systems, Inc. that is geared specifically towards the simulation of communication networks[10]. GTW is a popular parallel discrete event simulator from the Gerogia Institute of Technology that is based on Jefferson's Time Warp principle and primarily intended for shared-memory multiprocessors[17, 29].

The comparison with BONeS is meant to be a testimonial to the ability of the ProTEuS platform to produce ATM network simulation results equivalent to those produced by a mainstream discrete event simulator. While a portion of the comparison with GTW serves the same purpose, it is also an effort to demonstrate some of the scaling properties of the two systems and investigate how the properties of the simulated system, and the simulator itself, affect scaling.

First, some of the properties of generic distributed synchronous computations are considered and presented in Section 4.2, followed by an examination of the faithfulness of the ProTEuS platform for simulating ATM networks in Section 4.3. Finally, a head-to-head comparison of the scaling properties of ProTEuS versus GTW is presented in Section 4.4.

All BONeS and GTW results presented herein are courtesy of Murthy and Chong, respectively[40, 14].

4.1 Experimental Setup

All of the BONeS results were gathered on a 300 MHz dual-processor Sun UltraSPARC-II with 512 MB of RAM. Keep in mind that although this was a dual-processor machine, BONeS gains very little, if anything, from multiprocessor architectures because the software is not designed to exploit such concurrency.

All of the GTW results were gathered on 168 MHz 8-processor Sun UltraEnterprise with 1 GB of RAM. GTW is, of course, specifically designed to utilize the parallelism of a multiprocessor system through optimistic concurrency control and the detection of and recovery from temporal violation.

All of the ProTEuS results were gathered on a rack of 200 MHz single-processor Intel x86s with 128 MB RAM each, connected via an ATM network. The rack also includes a few 233 MHz and 500 MHz machines, some of which were included in one set of experiments, but for the most part, ProTEuS utilized the 200 MHz machines. ProTEuS embraces a Network of Workstations (NOW) approach using conservative synchronization and real-time control to distribute the simulation across a rack of machines.

From the experimental setup alone, you can see the striking differences between the basic architectures of the three systems and their hardware demands. This is not to say that these systems absolutely *require* such hardware, but rather that they were designed for, and benefit greatly from, such hardware. BONeS generally runs best on high-end single-processor machines with a generous amount of memory. GTW is targeted at high-end shared-memory (lots of it) multiprocessors where it can exploit parallelism. ProTEuS, to the contrary, is designed for commercial off-the-shelf (COTS) Networks of Workstations (NOWs), which lends itself to inexpensive and simple system expansion.

The ProTEuS experiments were run using the NetSpec control system described in Section 3.8, which distributes pieces of the simulation to participating members ac-

cording to an input script[30, 31]. Some of the data gathered was done via the Data Stream Kernel Interface (DSKI), which is a standard and extensible interface for gathering performance data from the Linux kernel in a variety of forms, including counters, event streams and histograms[9].

4.2 Distributed Synchronous Computation

This section is a discussion of some of the properties of a generic distributed synchronous computation; that is, a computation where synchronization information is exchanged among distributed entities, but the computational component of each epoch is, in effect, just a busy-wait. Several system parameters are varied and their affects on the performance of the system are examined.

Figure 4.1 depicts a distributed computation comprised of 24 simulation elements distributed across three host machines - eight elements per machine. Assuming that each distributed element consumes approximately the same computation time, the distribution is well-balanced; that is, each host in the experiment has roughly the same amount of work to do in each epoch.

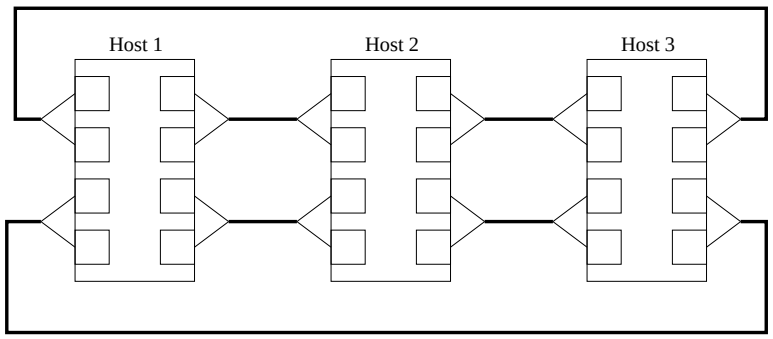

Figure 4.1: Balanced Synchronous Distributed Computation Topology

For the set of experiments in this section, we artificially impose this equality of computation components, but in general this may or may not be true, depending on the simulated system. In ProTEuS ATM network simulations, for instance, this is generally not the case, where virtual ATM software switch port entities consume more

computation time, for example, than a virtual ATM device (NIC) traffic source. Therefore, load balancing does not simply constitute equating the number of entities on each host, but more importantly, equating the computation time on each host.

Figure 4.2 shows an analogous distributed topology consisting of the same number of simulation elements as Figure 4.1. However, the busiest host here has twice as many simulation elements, and therefore twice as much work to do per epoch, as the other two hosts; this is an unbalanced distribution. As a consequence, this distribution also burdens the busiest host with twice as much communication overhead because it supports twice as many communication channels as the other two hosts. There is nothing fundamentally *wrong* with an unbalanced system, but the degree to which balance is skewed will affect the overall performance of the computation supported by ProTEuS.

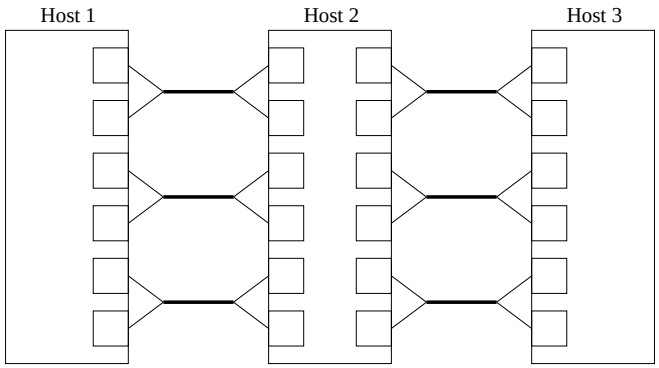

Figure 4.2: Unbalanced Synchronous Distributed Computation Topology

Two different scenarios are investigated which differ only by the individual computation component of each distributed element. As mentioned, for the sake of these experiments, we artificially impose a computation component for each element through a busy-wait. In the first trial, the computation component is 25µs per simulation element, while in the second, the computation component is increased to 200µs per element. The first scenario represents an application with very aggressive fine-grained needs, requiring epoch lengths on the order of 400 to 650 µs, depending on the system balance. The second scenario, with computation intervals nearly an order of magnitude larger, is more coarse, requiring epoch lengths on the order of 1800 to 3000 µs, and can

therefore support very high utilizations due to the large ratio of computation time to slack time.

Simulations utilizing the 25μs computation component per element were run for one million (10^6) successful epochs and those with a 200μs computation component were run for one hundred thousand (10^5) successful epochs.

It is worth noting here that the missed epoch counts reported in this section are not guaranteed to be 100% accurate. ProTEuS simulations are executed by a real-time thread running in kernel mode. A missed epoch occurs when the thread is awakened by KURT-Linux and it realizes that the information that it needs to proceed is not present. It increments the missed epoch counter and suspends, waiting for the next scheduled epoch to begin. However, consider the case where the thread does not complete its execution of the present epoch before the next epoch is scheduled to begin. When the timer expires, KURT-Linux suspends the thread and the real-time scheduler immediately resumes it. While this also constitutes a missed epoch, ProTEuS in and of itself has no way of detecting this, so it cannot keep track of this type of miss, which would require instrumentation by KURT-Linux itself. In general, this occurs most often when the length of the real-time period is prohibitively small. Note that in such instances, the computation still proceeds correctly, but somewhat more slowly as a result of the missed epochs.

4.2.1 Effects of Load Balancing

In this experiment, the effects of load balancing are investigated by using both the balanced and unbalanced topologies and varying the epoch length to find the minimum execution time for each distribution. Both *delta*, the epoch synchronization tolerance, and *wait time*, the second-chance waiting period for missing data, are kept at their default values of zero, meaning that the simulation enforces strict per-epoch synchronization, with no tolerance for late-arriving data. Table 4.1 and Figure 4.3 show the results.

First of all, in both scenarios, it is clear that the balanced topology performs better with regard to execution time. This is expected because the evenly balanced computa-

25µs Computation Component							
	Balanced Topology						
Epoch Length (µs)	375	400	425	*450*	475	500	525
Missed epochs	722884	584022	519202	*440644*	379641	337151	303834
% Missed	41.96%	36.87%	34.18%	*30.59%*	27.52%	25.21%	23.30%
Execution time (s)	746.719	663.657	658.125	*656.765*	663.065	675.156	691.563
	Unbalanced Topology						
Epoch Length (µs)	550	575	600	*625*	650	675	700
Missed epochs	529690	343012	234390	*176891*	136448	99345	70578
% Missed	34.63%	25.54%	18.99%	*15.03%*	12.01%	9.04%	6.59%
Execution time (s)	846.237	776.679	744.913	*739.633*	742.367	745.549	752.717
200µs Computation Component							
	Balanced Topology						
Epoch Length (µs)	1750	1800	1850	*1900*	1950	2000	2050
Missed epochs	73090	65051	59976	*54753*	53235	52845	51437
% Missed	42.23%	39.41%	37.49%	*35.38%*	34.74%	34.57%	33.97%
Execution time (s)	349.949	306.908	296.895	*294.640*	299.391	306.563	311.072
	Unbalanced Topology						
Epoch Length (µs)	2700	2750	2800	*2850*	2900	2950	3000
Missed epochs	35613	30419	25948	*22406*	21659	20767	19194
% Missed	26.26%	23.32%	20.60%	*18.30%*	17.80%	17.20%	16.10%
Execution time (s)	366.130	358.631	352.632	*348.836*	352.792	356.238	357.560

Table 4.1: The Effects of Load Balancing

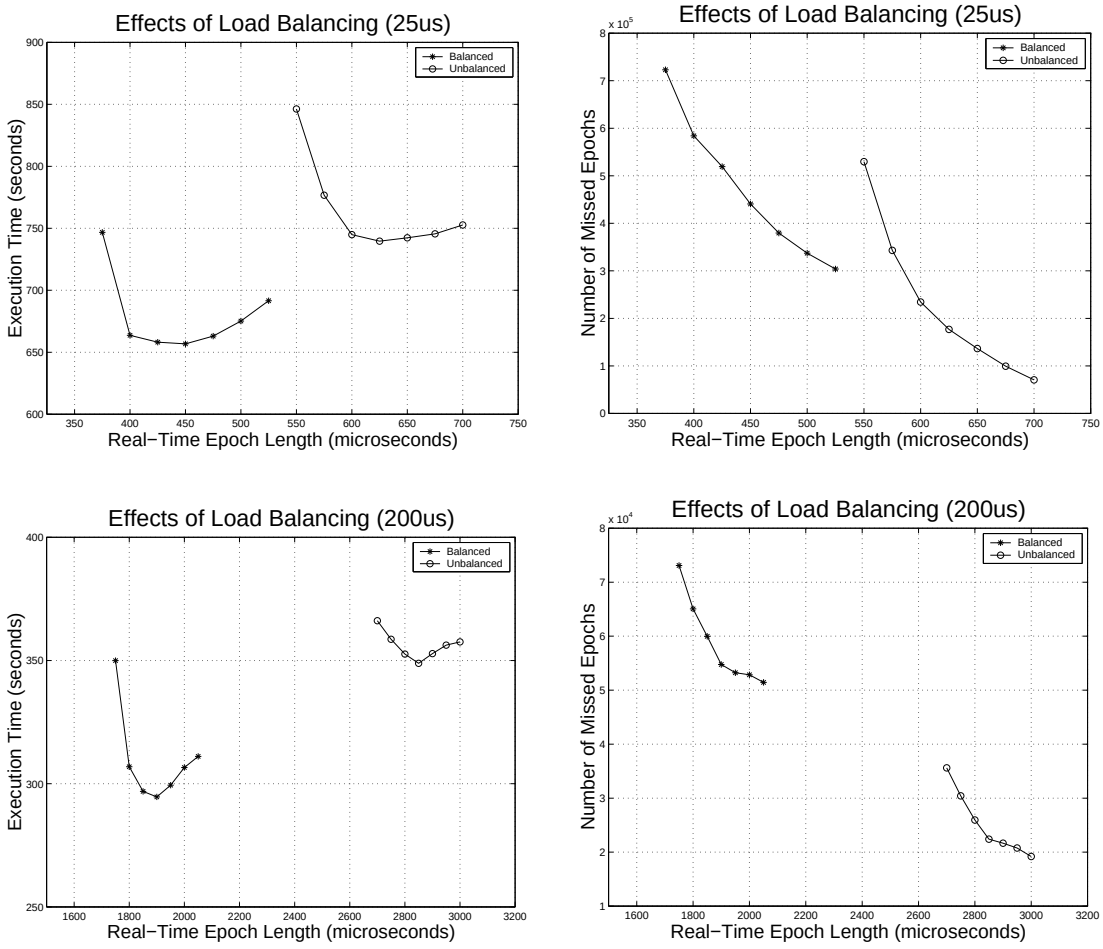

Figure 4.3: The Effects of Load Balancing

tion components allows ProTEuS to utilize a smaller epoch time than in the unbalanced case; the simulation moves only as fast as the slowest component. In the 200µs computation component scenario, this results in an epoch length nearly a millisecond smaller than in the balanced topology. Further, notice that the unbalanced topologies achieve their minimum execution times with a much smaller missed epoch count than in the balanced topologies. In both cases, its a factor of nearly two and a half times. In general, this is because missing an epoch in the unbalanced topology is a harsher penalty than in the balanced topology due to the longer epochs.

Another result from the graphs in Figure 4.3 worth noting is the shapes of the plots produced. The *Execution Time vs. Real-Time Epoch Length* curves are somewhat hyperbolic in nature; or, perhaps more generally, they are generic curves of degree two that exhibit a minimum value. If the epochs are too short, the system begins to miss a lot of epochs, as shown in the *Missed Epochs vs. Real-Time Epoch Length* graphs, resulting in an increase in execution time. The misses are often due to one of two things; (1) the phenomenon described earlier, where the real-time thread overruns its allotted execution time and spills over, undetected, into the next epoch (therefore, these misses are not accounted for in the missed epoch counts, but are nonetheless factored into the execution time, which accounts for both detected and undetected misses), or (2) because the computation time is generally fixed, the shorter epoch length causes a corresponding shrink in the slack time component of the epoch*. However, if the epochs are too long, the system usually misses fewer epochs, but it ends up wasting time because the slack time component of the epoch is overly large, which results in inflated execution times. Of course, the reason that the curves are steep on the left side and nearly linear on the right side is due to the steep slope of the *Missed Epoch vs. Real-Time Epoch Length* graphs, which are reminiscent of a negative exponential in shape. These plots show clearly that decreasing the length of an epoch causes more misses than increasing the length of an epoch saves. The result is that the cost, in terms of missed epochs, of having an epoch that is a bit too short is far greater than the cost of having an epoch that is a bit too long.

*Computation time is governed by the load distribution, not the epoch length.

4.2.2 Effects of Slack Time Choice

In this experiment, the effects of choosing a slack time, and therefore an epoch length, are investigated using the balanced topology and varying the utilization of the epoch. Both *delta*, the epoch synchronization tolerance, and *wait time*, the second-chance waiting period for missing data, are kept at their default values of zero, meaning that the simulation enforces strict per-epoch synchronization, with no tolerance for late-arriving data. Table 4.2 and Figure 4.4 show the results.

25µs Computation Component						
Utilization	20%	40%	60%	80%	90%	95%
Epoch Length (µs)	1580	790	525	395	350	-
Missed epochs	27	21084	301843	594706	739710	-
% Missed	0.00%	2.06%	23.19%	37.29%	42.52%	-
Execution time (s)	1579.800	810.459	690.528	673.003	721.311	-
200µs Computation Component						
Utilization	-	40%	60%	80%	90%	95%
Epoch Length (µs)	-	4290	2860	2145	1905	1805
Missed epochs	-	931	22718	46088	58929	64528
% Missed	-	0.92%	18.51%	31.55%	37.08%	39.22%
Execution Time (s)	-	432.922	350.947	314.142	303.304	304.676

Table 4.2: The Effects of Slack Time Choice

The first thing to note in the results presented here is that the 95% utilization experiment in the 25µs computation component scenario was not included because the magnitude of the slack time was prohibitively small and the epoch miss rate was *extremely* high. Similarly, the 20% utilization experiment in the 200µs computation component scenario was not included because it missed *no* epochs at all due to the magnitude of the slack time.

The first realization in these results is that using KURT-Linux as a real-time platform for a synchronous distributed computation, ProTEuS can achieve, for example, a ratio of missed epochs in the neighborhood of a mere 2% at a granularity of less than 800µs, which is 8% of the granularity of standard Linux. We believe this to be well within the operating range of *many* synchronous distributed applications.

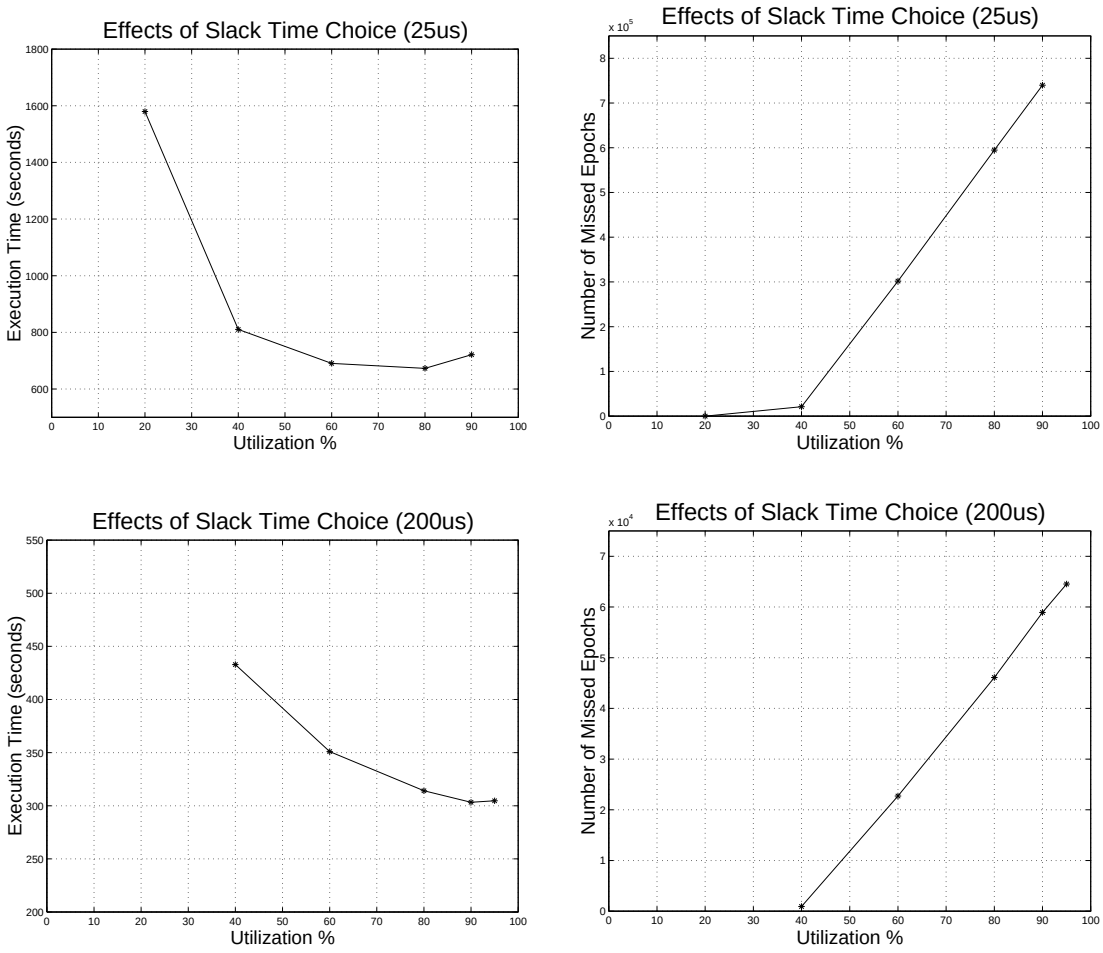

Figure 4.4: The Effects of Slack Time Choice

The results show that the missed epoch rate generally grows linearly with utilization, with exceptions to be made at the lower and upper bounds on utilization. Furthermore, the execution time also appears to be a function of utilization exhibiting a minimum, with each scenario achieving that minimum at a different utilization. In fact, what this exposes is the existence of a minimum necessary slack time. The slack time interval of the epoch length resulting in the minimum execution time, from the results from Section 4.2.1, is approximately 135µs in the 25µs computation component scenario, resulting in a utilization of about 70%, and 180µs in the 200µs computation time scenario, resulting in a utilization of about 90%*. Of course, these numbers are subject to the granularity of the chosen epoch intervals, but nevertheless, there appears to be a minimum slack time in the neighborhood of 150µs required to send and receive data in each epoch. The absolute value of this slack time, however, depends on several factors, including the network latency and delay variation, the processor speed, which dictates the length of an interrupt service routine, and the number of inter-processor communication channels supported on each node in the network. For instance, a node supporting communication with ten other nodes requires more slack than a node communicating with two other nodes. In the balanced topology, each node supports four lines of communication between other nodes, as shown in Figure 4.1.

Furthermore, the results show that the 200µs computation component scenario performs slightly better than the 25µs computation component scenario, insofar as missed epochs are concerned, due to its larger slack times. As alluded to in an earlier discussion in Section 3.7.2, the placement of the virtual time line in absolute time also affects the sufficiency of slack time, in that time lines that are out of sync on distributed nodes can even cause the *propagation* of misses around the system. The larger the epoch length is, the higher the probability of this phenomenon becomes and the combination of slack time and phase difference will dictate the effect on performance. It is important to note that in all experiments herein, including those presented in other sections, no effort was made to synchronize in absolute time because no mechanism current-

*The calculation of this value is not presented here, but the computation intervals on the busiest nodes in the two scenarios were measured over time and statistically determined to be approximately 316µs and 1717µs, respectively for the 25µs and 200µs computation component scenarios.

ly exists to do so with sufficiently fine granularity. However, doing so will be key to improving performance, and is part of future ProTEuS work.

4.2.3 Effects of Delta Values

In this experiment, the effects of *delta* values, or synchronization tolerance, is investigated using the balanced topology and varying the value of *delta* to provide a window of synchronization to lessen the effects of scheduling jitter, clock skew and communication overhead. The epoch lengths used are those producing the minimum execution times in Section 4.2.1; 450µs in the 25µs computation component scenario and 1900µs in the 200µs computation component scenario. The *wait time*, the second-chance waiting period for missing data, is kept at its default value of zero, meaning that the simulation will not wait around for late data outside the synchronization window. Table 4.3 and Figure 4.5 show the results.

25µs Computation Component							
Delta (epochs)	0	1	2	10	100	1000	10000
Missed epochs	437763	19360	4615	2843	1525	1216	0
% Missed	30.45%	1.90%	0.46%	0.28%	0.15%	0.12%	0.00%
Execution time (s)	655.418	466.943	460.674	460.116	459.645	459.413	458.861
200µs Computation Component							
Delta (epochs)	0	1	2	10	100	1000	10000
Missed epochs	57690	45	5	0	0	0	0
% Missed	36.58%	0.04%	0.00%	0.00%	0.00%	0.00%	0.00%
Execution time (s)	300.407	191.179	191.143	191.143	191.141	191.143	191.130

Table 4.3: The Effects of Delta Values

Note: In this chapter, delta values are referred to using a zero-indexed notation because of the manner in which they are specified in an actual ProTEuS experiment, while in Chapter 3, they were described in a manner consistent with being indexed by one. Therefore, while in Chapter 3 immediate consumption of data from epoch N *occurring in epoch* $N+1$ *was described as having a delta value of one, in this chapter, the same scenario is referred to as having a delta value of zero.*

The results show clearly that allowing distributed hosts to get out of sync by even

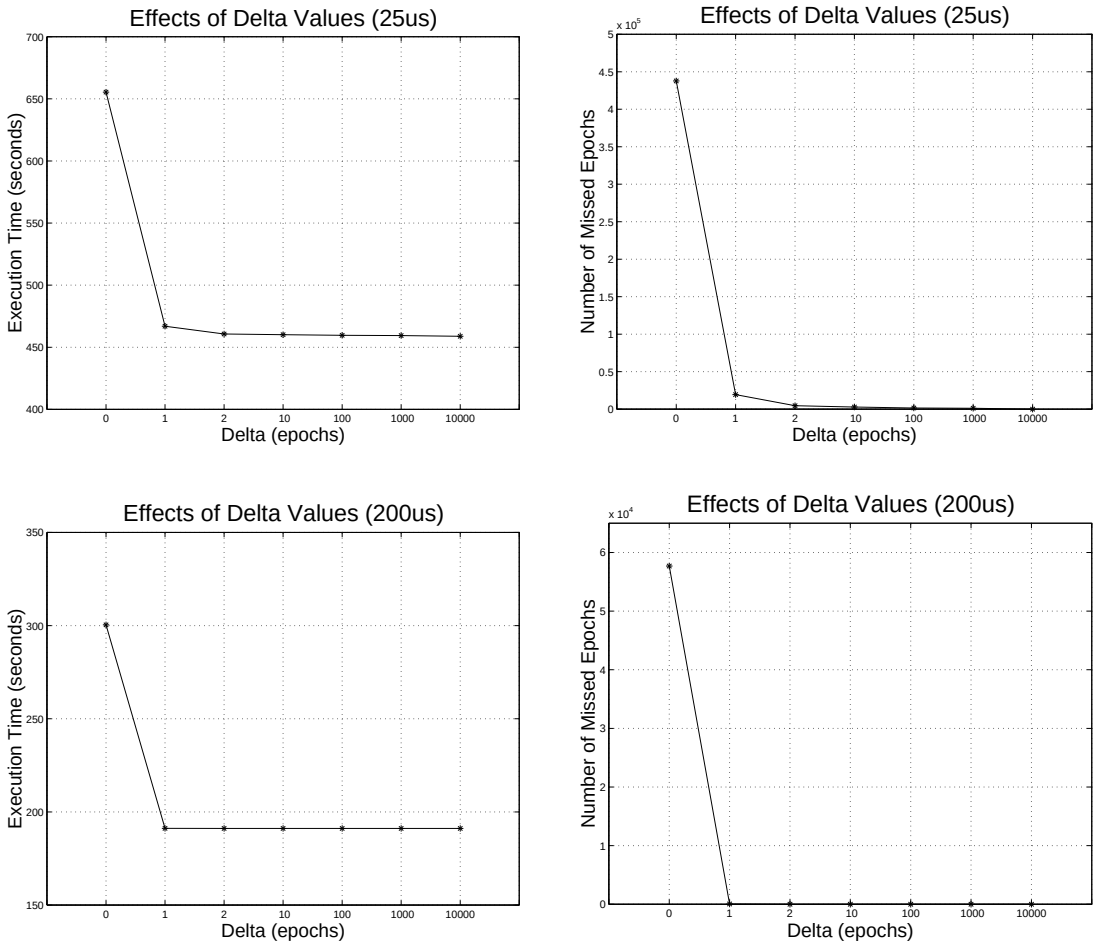

Figure 4.5: The Effects of Delta Values

a *single epoch* has startling effects on performance. In both scenarios, *delta*=1 decreases the number of missed epochs from over 30% down to less than 2%. This translates to an improvement in simulation time of nearly 200 seconds (30%) in the 25µs computation component scenario. *Delta* is very powerful because it not only offsets the effects of network delay variation, it also eliminates the problem of cascading epoch misses resulting from clock skew because nodes are often allowed to move on when data is missing.

One interesting result in the table that deserves explanation is the non-zero number of misses through *delta*=1000 in the 25µs computation component results that are not present in the 200µs computation component results. The reason that 1216 misses still occur with *delta*=1000 is that one or more nodes in the simulation is progressing faster than the rest. This is most notable in an unbalanced topology, but can still occur in a balanced topology due to distortions in execution time such as bottom half execution and interrupt service routines. The execution of bottom halves, which is done between the timer interrupt and the switch to the proportional time thread, will often delay the beginning of the epoch. If the delay is large enough, it can cause the execution to overlap into the next epoch. While this goes unnoticed on the local host, it is detected as a missed epoch by the other nodes. Further, depending on the relative positions of their respective virtual time lines on the absolute time line, some nodes may find themselves executing most of their interrupt service routines within their computation interval, while other nodes receive their interrupts during the slack time. The result is elongated computation intervals, which also may cause the epoch execution to overlap into the subsequent epoch. If this behavior is consistent, the result is that even a synchronization window of 1000 epochs in this case may not be enough to insure zero misses. However, larger slack times, taking control of bottom half execution and working to minimize the ATM interrupt service routines, which currently take approximately 30µs in Linux, will help to offset this problem. Lastly, recall that the 200µs computation component scenarios were only run for one-tenth the number of epochs as the 25µs computation component scenarios. If they had been run for a full one million epochs, it is possible, if not likely, that we would see a similar trend there.

As a reference for comparison, remember that in ProTEuS ATM network simulations, the minimum link delay simulated on a node determines its *delta* value. In an OC-3 ATM network simulation, a link delay of 5ms would translate to a *delta* of 1825 epochs.

4.2.4 Effects of Waiting for Missing Data

In this experiment, the effects of waiting for missing data is investigated using the balanced topology and varying the value of *wait time* to allow late data a second-chance to arrive, offsetting some variation in network delay and execution time. The epoch lengths used are those producing the minimum execution times in Section 4.2.1; 450µs in the 25µs computation component case and 1900µs in the 200µs computation component case. The value of *delta*, the epoch synchronization tolerance, is kept at its default value of zero, meaning that the simulation enforces strict per-epoch synchronization. Table 4.4 and Figure 4.6 show the results.

25µs Computation Component							
Wait Time (µs)	0	50	100	150	200	250	300
Missed epochs	445964	317383	223903	108370	55411	38133	27020
% Missed	30.84%	24.09%	18.29%	9.78%	5.25%	3.67%	2.63%
Execution time (s)	659.167	601.383	569.124	540.069	529.579	529.231	530.056
200µs Computation Component							
Wait Time (µs)	0	50	100	150	200	250	300
Missed epochs	58258	55736	52220	49117	44030	40099	36927
% Missed	36.81%	35.79%	34.31%	32.94%	30.57%	28.62%	26.97%
Execution time (s)	301.323	296.617	289.854	284.609	277.386	273.234	268.584

Table 4.4: The Effects of Waiting for Missing Data

The results of these experiments certainly support the notion that giving data a second-chance to arrive can be beneficial to simulation performance. In both scenarios, the number of missed epochs essentially decreases linearly with increased *wait time*. However, as the number of missed epochs nears zero, waiting longer begins to produce diminishing returns, as expected. In fact, as shown in the *Execution Time vs. Time to Wait* graph for the 25µs computation component scenario, waiting too long can

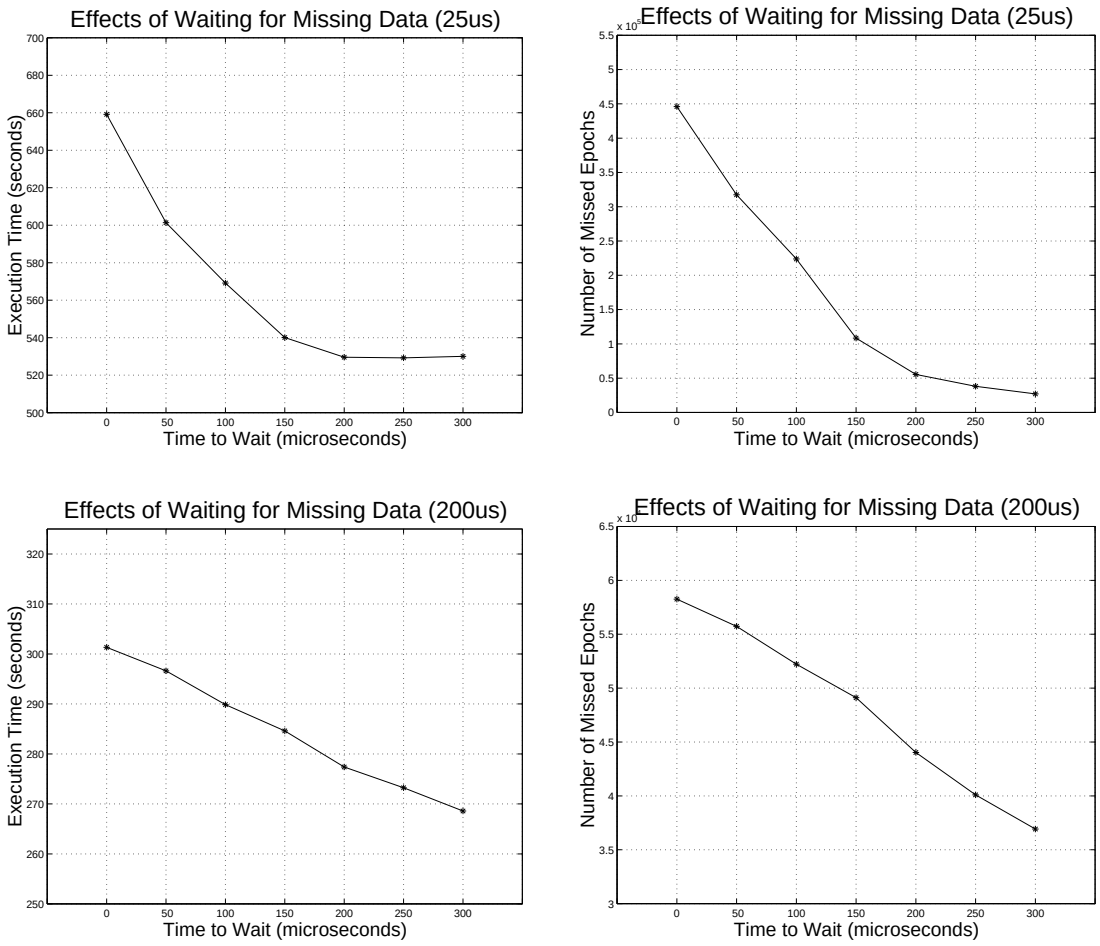

Figure 4.6: The Effects of Waiting for Missing Data

actually be detrimental to performance. The reason is that waiting longer postpones the execution of the epoch, which can cause its execution to overlap into the next epoch. The result is an apparent reduction in the number of missed epochs due to the inability of ProTEuS, in and of itself, to detect this type of missed epoch, but an increase in actual execution time, which accounts for both detected and undetected misses. In this graph, for instance, from a *wait time* of 250µs to 300µs, the number of missed epochs appears to be reduced from 38133 to 27020, but the execution time actually increases from 529.231 seconds to 530.056 seconds.

One of the functions of *wait time* is to offset the effects of virtual time lines on distributed nodes being interleaved in absolute time by giving late data extra time to arrive. However, for this to be ultimately successfully in doing so, the *wait time* needs to be near in size to the epoch length. This can be seen clearly in the results in Table 4.4, where in the 25µs computation component scenario, a *wait time* of 300µs successfully decreases the number of missed epochs from 445964 (30.84%) to 27020 (2.63%)*. In this case, the *wait time* is very near the epoch length of 450µs. In the 200µs computation component scenario, however, a *wait time* of 300µs only succeeds in decreasing the number of missed epochs from 58258 (36.81%) to 36927 (26.97%). In this case, the *wait time* of 300 µs is no where near the epoch length of 1900µs.

It is worth noting that while waiting for missing data appears to improve simulation performance, it was not used in any experiments presented in subsequent sections. Further, its value-added to the already enormous performance improvements achieved by *delta* values is as yet unknown and is not presented here.

4.2.5 Effects of Bottom Halves

In an effort to minimize the amount of work done in an interrupt service routine, many interrupt handlers use a bottom half, which is a way to defer work to be completed out of interrupt context. Only critical work is done during the interrupt, which marks the bottom half as a notification to the system that it needs to run the bottom half for this

*Keep in mind that for reasons already discussed in length, these missed epoch counts may not be completely accurate accounts.

interrupt, whose duty is to finish the work. Bottom halves are run at the discretion of the system and are executed without regard to time. Some bottom halves, such as the network and SCSI disk bottom halves introduce significant jitter because they are allowed to run for significant lengths of time, much of it with interrupts disabled. In a standard Linux system, which utilizes a 10ms periodic interrupt to keep time, occasional distortions of a few hundred microseconds are largely insignificant. When trying to maintain fine-grained periodicity, however, they become quite significant.

To demonstrate the effects of Linux bottom halves on the performance of a synchronous distributed computation, a histogram of the latency between the timer interrupt and the beginning of the epoch was gathered. This is important because essentially the *only* thing between the timer interrupt and the beginning of the epoch *is the bottom halves*. This experiment utilizes the balanced topology and the 25µs computation component. The epoch length is 500µs and both *delta* and *wait time* are kept at their default values of zero, meaning that the simulation enforces strict per-epoch synchronization, with no tolerance for late-arriving data. Histogram data was collected for 60 wall-clock seconds and Table 4.5 and Figure 4.7 show the results.

	Distortion (µs)			
Host	50-99	100-149	150-199	200+
Host 1	10802	392	102	14
Host 2	11413	425	145	55
Host 3	14890	2149	189	37

Table 4.5: Effects of Bottom Halves

It is important to note that not all results from the histogram data collected are presented here. In fact, the vast majority of the occurrences are less than 50µs, but only those that we feel were adversely affecting the execution of the ProTEuS real-time thread are shown here. Further, keep in mind that during this period, the two most notorious bottom half culprits were essentially inactive. There was no disk activity to speak of, so the SCSI bottom half was dormant, and while there was certainly some network activity, it was by no means significant, meaning that the network bottom half was also not very active.

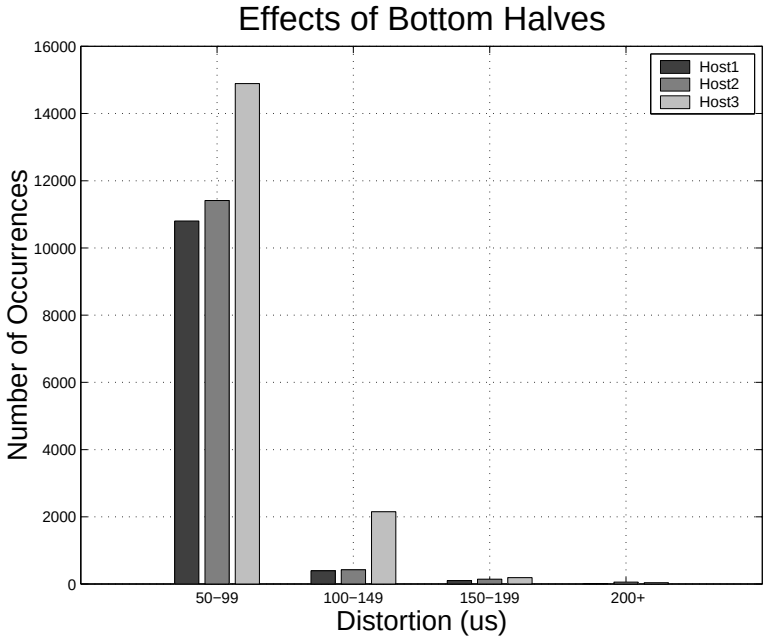

Figure 4.7: Effects of Bottom Halves

The histogram clearly shows that the latency between the timer interrupt and the actual start of the epoch is often intrusive. Several hundred times a second the latency is between 50 and 100µs, and few times per second it exceeds 150µs. These are significant distortions when attempting to maintain a periodicity of 500µs. Distortions such as these essentially offset the slack time of the epoch, and as alluded to earlier can actually cause subsequent missed epochs by forcing the execution of the current epoch to overlap the arrival of the next epoch. Getting control of the bottom halves is key to KURT-Linux's ability to provide reliable fine-grained schedulability.

4.3 The Faithfulness of ProTEuS for ATM Simulation

In order to make an argument advocating the viability of the Proportional Time platform for simulating ATM Networks, and by extension other systems, it is important to establish the ability of the platform to produce results faithful to the system being simulated; or in the very least, comparable to those produced by commonly used sim-

ulation techniques.

In this section, network and simulator metrics were gathered from a rather uninteresting but revealing ATM network simulation. Figure 4.8 shows the network topology, which contains three ATM switches, one of them a bottleneck, and four sources and sinks, two Available Bit Rate (ABR) and two Variable Bit Rate (VBR). Results are gathered and presented for three discrete event simulation platforms; BONeS, GTW and ProTEuS.

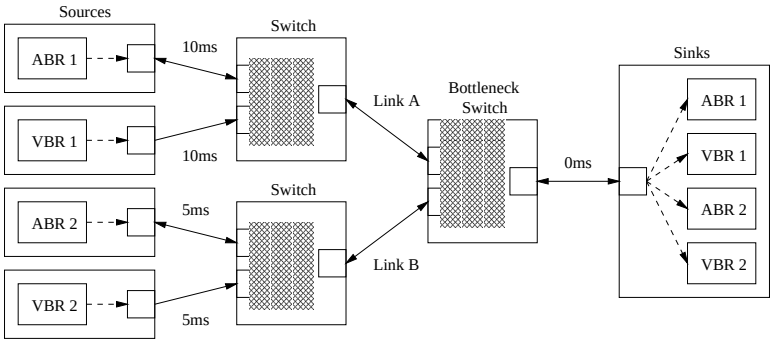

Figure 4.8: Faithfulness Testimonial Topology

Table 4.6 lists the relevant simulation parameters. The seemingly low line rate of 8000 cells per second was a legacy requirement of a predecessor of ProTEuS* and was used in some corresponding BONeS models in Murthy's experimentation[40]. This line rate is retained here solely for the sake of comparison with those BONeS results.

The ABR sources are greedy, initially sending at 1000 cells per second, and eventually will share the available bandwidth at the bottleneck link. The VBR sources are reconstructed cell streams from cell-level traces of an MPEG video clip from the movie *Star Wars*, whose behavior are bursty with a sustained cell rate of approximately 3000 cells per second. The software switches exhibit output port per-class queueing with weighted round-robin service between VBR and ABR queues and ABR RM cell feed-

*This line rate corresponds to a cell-time of 125µs, which is roughly the smallest reliably schedulable periodicity under KURT-Linux due to system distortions and interrupt service. In the work previous to ProTEuS, the simulations were run in real time, not virtual time, which imposed this limitation on simulated line rates. This limitation on line rate is lifted by the advent of proportional time, but the lower limit on real-time epoch lengths is not.

Simulated Line Rate	8000 cells per second (cps)
ABR Sources	Greedy: PCR=8000 cps, CCR=1000 cps, MCR=0 cps
VBR Sources	Bursty: SCR=~3000 cps
ABR Feedback Mechanism	EPRCA: Thresh_{low}=200 cells, Thresh_{high}=300
Switch Queue Lengths	Maximum=5000 cells
Switch Queueing Discipline	Per-class: VBR::ABR=200::1
Link Delays	Scenario (1) A=5ms, B=20ms
	Scenario (2) A=15ms, B=15ms
	Scenario (3) A=20ms, B=5ms
Simulated Time	50 seconds

Table 4.6: Faithfulness Simulation Parameters

back is provided by the EPRCA explicit rate scheme. The link delays on links A and B are varied over three scenarios and each was run for 50 simulated seconds.

The ProTEuS and GTW models were both mapped equivalently across three processors - GTW using three processors of a shared-memory multiprocessor, and ProTEuS using three Linux workstations connected by an ATM network.

4.3.1 Link Utilization

Due to of the burstiness of the VBR sources, the utilization of bandwidth on links A and B will constantly vary as the EPRCA feedback mechanism attempts to keep the bottleneck link full by giving the instantaneously available bandwidth to the ABR sources. Of course, the actual utilizations will depend on the feedback mechanism and the delay that backward RM cells that carry the explicit rate information incur. For each of the three scenarios, the average utilization of links A and B was measured at the output port of the upstream switch on each of the three simulation platforms. Table 4.7 shows the results.

First of all, it is obvious from the table that all three systems produce results sufficiently close that it is difficult, if not impossible, to declare any one set of results clearly right or wrong. However, consider what perturbations the three scenarios represent. First, each successive experiment (down the columns in Table 4.7) *increases* the round-trip time of the cells traversing link A and *decreases* the round-trip time of the cells

Experiment	Link A			Link B		
	BONeS	GTW	ProTEuS	BONeS	GTW	ProTEuS
A:5ms B:20ms	0.495	0.502	0.503	0.505	0.498	0.497
A:15ms B:15ms	0.494	0.498	0.499	0.506	0.502	0.501
A:20ms B:5ms	0.493	0.498	0.499	0.507	0.502	0.501

Table 4.7: Mean Normalized Link Utilization

through link B. Second, notice that the round-trip times through the two links are not equal in any of the three scenarios; in the first, round-trip times via link A are lower, and in the last two, round-trip times through link B are lower.

Consider that, in general, shorter delays between the bottleneck switch and the ABR sources means timelier feedback, and therefore should result in higher link utilizations. Stale feedback leads to periods of inefficient bandwidth utilization when bandwidth is available, but the sources are not yet aware, and potentially subsequent periods of long queueing delays and rapid back-off when the sources respond to the outdated feedback and send on a network that may now be congested. Therefore, as delay increases on a link, utilization should decrease, and vice versa.

Table 4.7 shows that in all three simulation environments, as the delay on link A increases (down the columns), its utilization decreases or remains constant. Likewise on link B, as delay decreases (down the columns), its utilization increases or remains constant. In this respect, all three platforms produce perfectly plausible results. However, the BONeS results are missing one key trend that the other two systems exhibit. In scenario one, cells across link A have a lower round-trip time than cells across link B, and therefore link A should attain higher average utilization. In the last two scenarios, the opposite is true, such that link B should achieve higher utilizations than link A. Table 4.7 shows that while the ProTEuS and GTW results do show this, the BONeS results do not; link B always achieves higher utilizations in the BONeS experiments. Now, this is not to say that BONeS is inherently flawed, but rather that the ABR model likely has discrepancies that may have been the result of errors, abstractions, omissions or some other factor. These are the potential pitfalls of implementing system code in a simulator that ProTEuS attempts to avoid altogether by using real, (essentially) unmodified

system code.

One last observation that also applies to all of the subsequent results in this section as well is the very close correlation between the GTW and ProTEuS results. This is due in part to the careful attention to implementation detail during the development of the two models. However, BONeS is not left out in all of this, in that both GTW and ProTEuS take their EPRCA and weighted round-robin cell service implementations directly from the BONeS models in Raguparan's work[48].

4.3.2 Mean Queueing Delay

ABR feedback mechanisms struggle to keep average utilizations high and average delays low through efficient ABR source rate pacing. Some feedback schemes designed specifically to do just that, such as ERICA, keep queueing delays *far* below those produced by EPRCA, while delivering only slightly lower utilizations in most cases[32, 50]. Therefore, the queueing delays incurred by ABR streams are largely dependent on the efficiency of the ABR feedback scheme, which in turn typically depends on the delay of the feedback. For each of the three scenarios, the mean queueing delays of each of the two ABR streams was measured at the output port of the bottleneck switch on each of the three simulation platforms. Table 4.8 shows the results.

	ABR 1 queuing delay (sec)			ABR 2 queuing delay (sec)		
Experiment	BONeS	GTW	ProTEuS	BONeS	GTW	ProTEuS
A:5ms B:20ms	0.143	0.159	0.156	0.147	0.164	0.163
A:15ms B:15ms	0.149	0.165	0.163	0.148	0.161	0.160
A:20ms B:5ms	0.154	0.167	0.165	0.147	0.159	0.157

Table 4.8: Mean ABR Cell Queuing Delay

Because the queueing delays are directly related to the link delays, as was the case with link utilizations, the trends, and much of the rationale is the same. For instance, as the delay incurred by backward RM cells increases, Table 4.7 shows that utilization suffers because the feedback from the congestion control scheme on the bottleneck switch is stale by the time it reaches the sources. As alluded to earlier, this often leads

to sources increasing their cell rates when they should not*, which in turn causes longer delays on the switch as queues build up, followed by a back-off from the sources, an oscillatory effect, as described by Raguparan and Murthy, resulting in higher mean queueing delays[48, 40].

In Table 4.8, as the delay on link A increases (down the column), so does the delay on RM cells associated with *ABR 1*, which increases the mean queueing delay experienced by data cells in that stream. Similarly, as the delay on link B decreases, the mean queueing delay for data cells in the *ABR 2* stream decreases. Further, just as before, in scenario one, link A, and therefore the *ABR 1* stream, has a lower round-trip time than link B, while for the last two scenarios, the reverse is true. Therefore, in the first scenario, the mean queueing delay of ABR cells on link A (*ABR 1*) should be lower, and for the last two scenarios, the mean queueing delay of cells on link B (*ABR 2*) should be lower. As Table 4.8 depicts, this is the case in all three models, but once again, some of the BONeS results look slightly suspect, specifically the *ABR 2* mean delays, which are essentially unchanged across the three scenarios. Again, the exact reason for the discrepancy is unknown, and further this is not concrete evidence that the BONeS model is actually flawed, but it does point to potential issues common to conventional simulation techniques, such as excessive abstraction. For instance, the original BONeS ABR models in Raguparan's studies modeled RM cell feedback to the ABR sources through timer-controlled mechanisms, based on round-trip times, as opposed to actual cell flow, which had adverse effects on simulation results[48].

4.3.3 ABR Queue Length

As discussed in Section 3.5.4.1, EPRCA is an explicit rate feedback mechanism whose primary input is the current cell rate and the length of the associated ABR output queue. EPRCA calculates an allowed cell rate using exponential averaging to provide explicit rate feedback and supplies binary feedback by categorizing the ABR queue length into one of three regions delineated by low and high thresholds. The length of

*Of course, this is also dependent on the properties of the other higher-priority network traffic; in this case, the VBR sources and their burstiness.

the queue controls whether the source is allowed to increase its rate, decrease its rate or neither and the algorithm, in effect, keeps the queue length between Thresh_{low} and Thresh_{high} on average. For this experiment, results were gathered only for Scenario (2) in Table 4.6, where link A and B both have delays of 15ms. In GTW and ProTEuS, the ABR queue length of the output port on the bottleneck switch was measured at the reception of each backward RM cell. This data was not collected in the BONeS simulation models, so no comparison with BONeS is made. Figures 4.9 through 4.11 show the results.

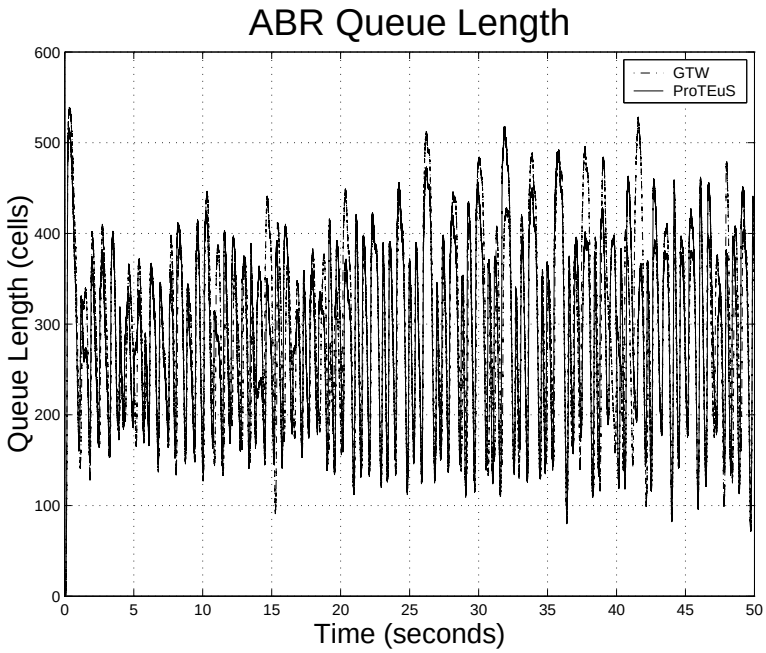

Figure 4.9: ABR Queue Length

The graph in Figure 4.9 shows the queue length over the entire 50 second simulation and while the granularity of the graph makes it difficult to claim anything with certainty, it is clear that the graphs are very similar. There are regions, such as between 10 and 20 seconds where there is some discernible differences, and regions where the plots are virtually indistinguishable, like between 20 and 28 seconds.

Figure 4.10 is a closeup of Figure 4.9 between 0 and 10 seconds. This is an interesting section of the time line because it includes the ABR ramp-up that occurs be-

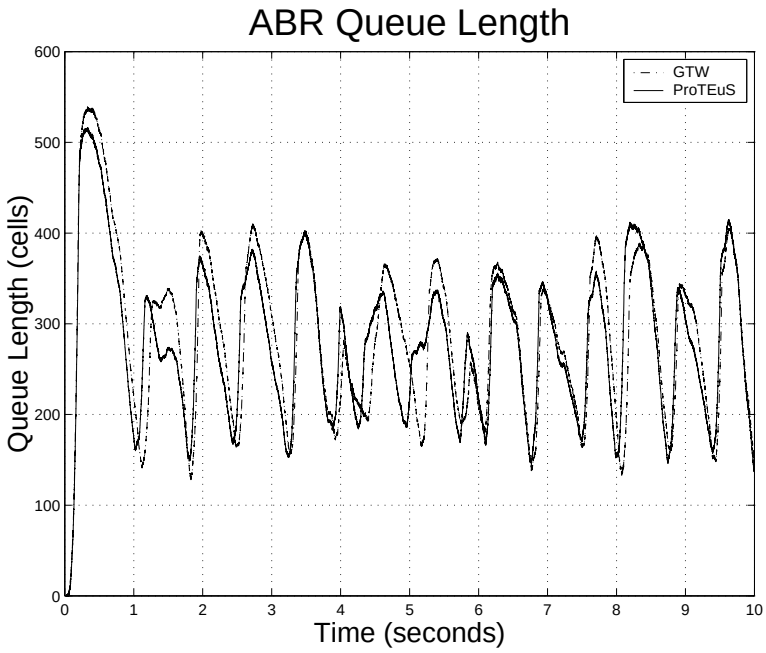

Figure 4.10: ABR Queue Length (zoom 0 - 10 seconds)

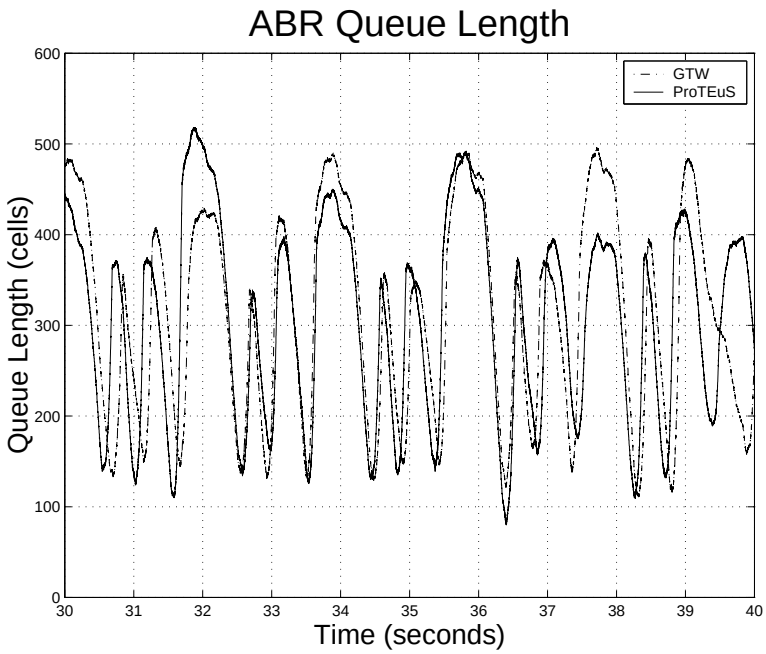

Figure 4.11: ABR Queue Length (zoom 30 - 40 seconds)

tween 0 and 1 seconds in simulated time. In this region, the queue length very quickly grows from 0 to beyond $\text{Thresh}_{\texttt{high}}$ (300 cells) as the feedback scheme continues to allow the ABR sources to increase their source rates. At some point near where the queue length reaches $\text{Thresh}_{\texttt{high}}$, EPRCA begins to throttle the ABR source rates and the queue length drops until it reaches approximately $\text{Thresh}_{\texttt{low}}$ (200 cells), at which point the sources are again allowed to increase their source rates. From then on, for all intents and purposes, the queue length oscillates between $\text{Thresh}_{\texttt{low}}$ and $\text{Thresh}_{\texttt{high}}$ without significant exceptions. Figure 4.11 shows another closeup of a different section of the simulated time line, from 30 to 40 seconds. In all three graphs, there are certainly a few regions where some disparities arise between the results from GTW and ProTEuS, but they are well within the realm of random variation and it is in no way clear where in the models the discrepancy may exist, nor even in *which* model it exists.

4.3.4 ABR Source Rate

The aim of ABR sources is to consume the instantaneously available bandwidth in a network and ABR feedback schemes vary their source rates attempting to maximize network utilization, while preserving the quality of service guarantees of all sources, including the ABR sources. Most ABR implementations utilize additive increase, multiplicative decrease, which permits them to linearly increase their source rate when bandwidth is available and exponentially decrease it in the face of congestion. For this experiment, like the last, results were gathered only for the the scenario in which link A and B both have delays of 15ms. In GTW and ProTEuS, the source rate of one of the two ABR sources was measured at the reception of each backward RM cell, which is where the source rate is updated by the feedback scheme. This data was not collected in the BONeS simulation models, so no comparison with BONeS is made. Figures 4.12 through 4.14 show the results.

Figure 4.12 depicts the source rate of *ABR 1* for the entirety of the 50 second simulation. Again, in general, the plots from GTW and ProTEuS are very similar with a few noticeable inconsistencies, such as the very high spikes in source rate that GTW encounters at 15 and 46 seconds that ProTEuS does not. Figure 4.13 shows a closer look

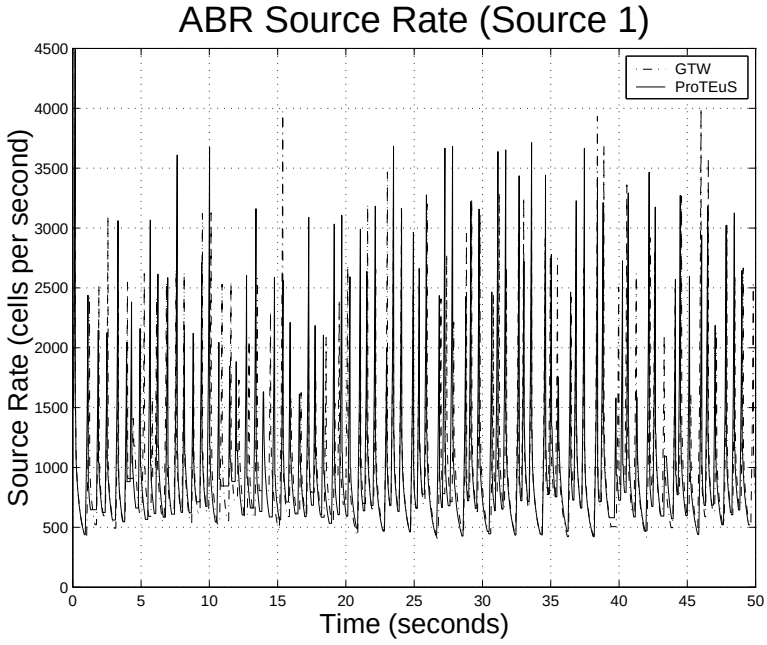

Figure 4.12: ABR Source Rate

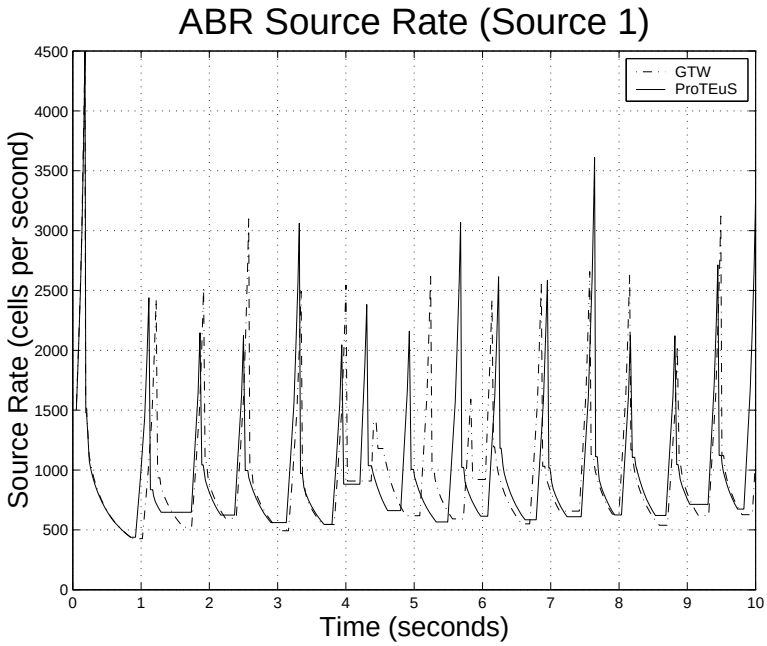

Figure 4.13: ABR Source Rate (zoom 0 - 10 seconds)

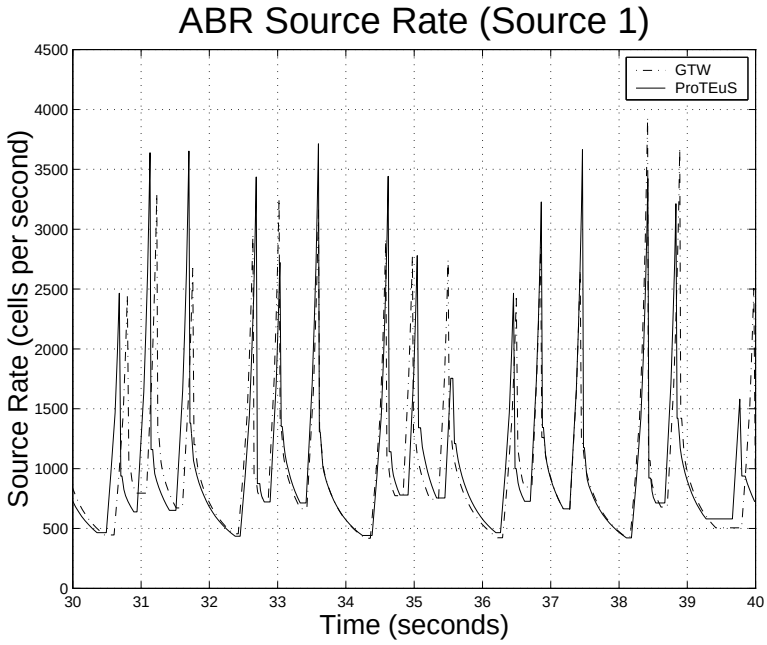

Figure 4.14: ABR Source Rate (zoom 30 - 40 seconds)

at the time period between 0 and 10 simulated seconds, including the ABR ramp-up between 0 and 1 seconds. During this period, the source rate ramps-up very quickly because of the initially empty queues on the switches and then declines rapidly as they fill. Notice that while the ABR sources have an initial cell rate (ICR) of 1000 cells per second (cps), the graph starts at 1500. This is because the measurements were taken *after* the RM cell feedback was processed, meaning that the first backward RM cell allowed the ABR source to increase its rate *50%* from 1000 cps to 1500 cps.

Figure 4.14 shows another zoom of simulated time; this one showing the period between 30 and 40 seconds. The two closeup graphs show that, other than a minor period of chaos between 4 and 6 seconds, the correlation of the results from two platforms is quite good. Of course, while the discrepancy in Figure 4.13 certainly deserves its due attention and investigation, when compared to the other 48 seconds of simulated time, this is not a significant enough disagreement to conclude that either simulation model or corresponding platform is inherently erred.

4.3.5 Execution Time

Table 4.9 shows the resulting execution times for each scenario on all three simulation platforms. To the surprise of nobody, BONeS, the sequential discrete event simulator, takes *orders of magnitude* longer to simulate this seemingly elementary network. While BONeS takes more than an hour and a half to simulate 50 seconds, both GTW and ProTEuS complete the task in less than a minute and a half. This is a somewhat expected result, because, if due to nothing else, GTW and ProTEuS distribute the simulation across three processors executing in parallel.

Experiment	Execution Time (HH:MM:SS)		
	BONeS	GTW	ProTEuS
A:5ms B:20ms	01:40:36	00:01:24	00:01:28
A:15ms B:15ms	01:40:36	00:01:23	00:01:28
A:20ms B:5ms	01:40:36	00:01:22	00:01:28

Table 4.9: Execution Time

One caveat to keep in mind is that the execution time of a ProTEuS simulation is directly related to the virtual to real time epoch ratio, which is strongly correlated to the simulated line rate in our ATM simulations, since it maps a real-time period to each simulated cell-time. Therefore, in these experiments, with comparatively large cell-times due to small simulated line rates, 125µs in this case, ProTEuS can *almost* run them in real-time; that is, a one-to-one mapping between real and virtual time. However, simulations where the line rate is increased to more realistic values, such as OC-3, such a feat would require a real-time period of less than 3µs - an impossible task on commercial off-the-shelf hardware. In general, however, larger real-time periods resulting in a larger simulation slowdown from real-time are not inherently bad because; (1) coarse time granularity lessens the affects of system scheduling distortions and network latencies, and (2) larger periods gives ProTEuS more time to do work, increasing CPU utilization and enabling the simulation of very large networks as presented in Section 4.4

4.4 ProTEuS vs. GTW

To further establish the capabilities of the Proportional Time platform, the need arises to explore the scaling properties of the system as compared to a system with well-established credibility. In this section, experiments of varying size and complexity were performed on both ProTEuS and GTW in an effort to examine the properties of each simulation platform and how they relate to the system being simulated. Two edge-core ATM network topologies are simulated; one with 6 ATM switches and 40 hosts, and the other with 16 ATM switches and 120 hosts. Some of the perturbations include varying the network size, traffic types and parameters, round-trip times, network load and the distributed mapping of entities to processors.

4.4.1 Scenario A

Scenario A is a 6 ATM switch, 40 host edge-core network topology consisting of a single bottleneck link depicted in Figure 4.15. The set of experiments presented here consists of four traffic scenarios:

- *Uni-directional ABR traffic*: Consists of ABR and VBR sources sending traffic logically from left to right in Figure 4.15. That is to say, each of the 20 hosts on the left side of the network is a traffic source, while each of the 20 hosts on the right side of the network is a traffic sink. Traffic sources are divided evenly, 10 ABR and 10 VBR, and are further balanced so that the same number of ABR and VBR sources flow across each edge switch (5 of each per edge switch), as shown in Figure 4.15.

- *Uni-directional TCP over ABR*: Identical to uni-directional ABR, except that instead of being raw ATM sources, the ABR sources are TCP sources transmitting over an ABR virtual circuit. TCP introduces a number of issues and behaviors that impact simulation performance, including slow-start, acknowledgement and MTU implications, just to name a few.

- *Bi-directional ABR*: Each host in Figure 4.15 now acts as both a source *and a sink*, such that the bottleneck pipe is filled in both directions. This essentially doubles

the load on the network.

- *Bi-directional TCP over ABR*: This scenario is the same as bi-directional ABR, except that the sources are TCP sending over an ABR pipe.

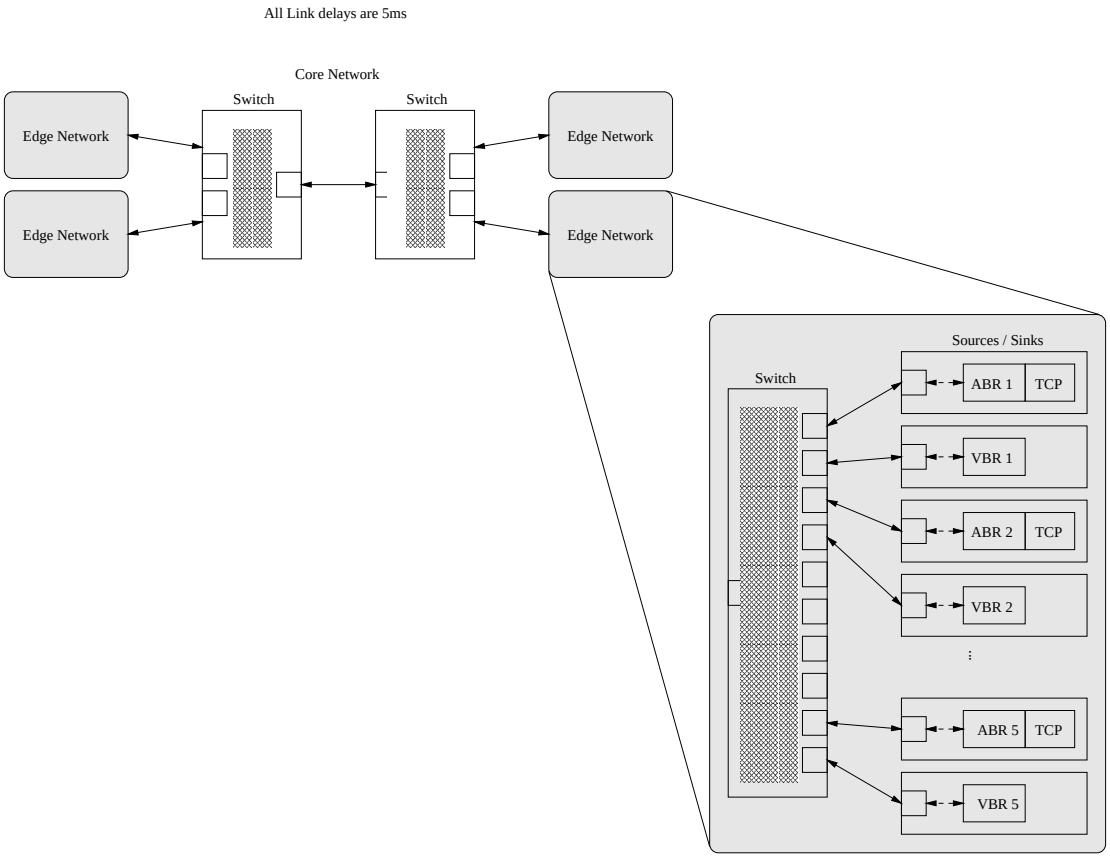

Figure 4.15: 6 Switch 40 Host Scalability Comparison Topology

Table 4.10 shows the parameters for the simulations. The simulated line rate is increased to a more realistic 365000 cells per second, which is approximately OC-3. The traffic sources for these experiments were specifically crafted to keep the normalized utilization of the bottleneck near unity without invoking cell-loss or ABR feedback mechanisms, because the intent of these experiments is not to examine a particularly interesting ABR network, but rather to stress the simulation platforms. The VBR sources are square waves sending at an average rate of 12500 cells per second (cps)

and a peak of 15000 cps. The ABR sources peak at 21000 cps, which is enough to keep the average utilization of the bottleneck at around 335000 cps, or about 0.92. Because the peak cell rate (360000 cps) is also below link capacity, ABR throttling should not occur at all. Because they are transmitted over an ABR virtual circuit, the TCP sources are bound by the same parameters as the ABR sources, and they have a TCP window of 512KB, which is more than enough to fill the ABR pipe. All simulations were run for 10 simulated seconds.

Simulated Line Rate	365000 cells per second (cps) ~ OC-3
ABR Sources	Greedy: PCR=21000 cps, ICR=25% PCR, MCR=0 cps
TCP Sources	Greedy: Window=512KB
VBR Sources	50% square: Min/Max=10000/15000, period=100ms
ABR Feedback Mechanism	EPRCA: Thresh_{low}=200 cells, Thresh_{high}=300
Switch Queue Lengths	Maximum=5000 cells
Switch Queueing Discipline	Per-class: VBR::ABR=200::1
Link Delays	5ms fixed delay
Simulated Time	10 seconds

Table 4.10: Scenario A Simulation Parameters

Furthermore, to demonstrate both speedup capabilities and flexibility, each of the four scenarios was run on three different physical mappings; two processors, four processors and six processors. The virtual to physical entity mappings for each of these are shown in Figures 4.16, 4.17 and 4.18, respectively, where the dotted ellipses demonstrate the virtual to physical mappings. It is worth noting here that the reason that no results are presented for mappings in excess of six processors is due to the limitations of the SMMP hardware at our disposal for the GTW simulations. The heftiest multiprocessor machine available was an eight processor Sun, of which only seven processors are available for GTW simulation purposes. Because mappings across seven processors were unnatural for the ATM simulations presented here, mapping simulations across six processors was the largest plausible mapping for the GTW simulations.

The two processor mapping is a very natural one, splitting the topology vertically in the center. One obvious advantage of such a mapping is that it is well-balanced. Actually, it is perfectly balanced in the bi-directional cases, and nearly perfect in the

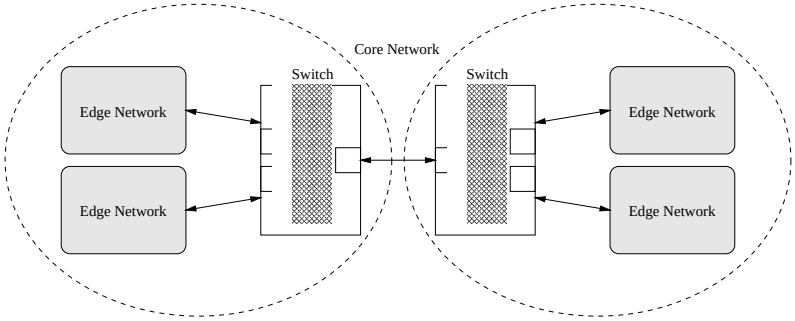

Figure 4.16: Scenario A - 2 Host Mapping

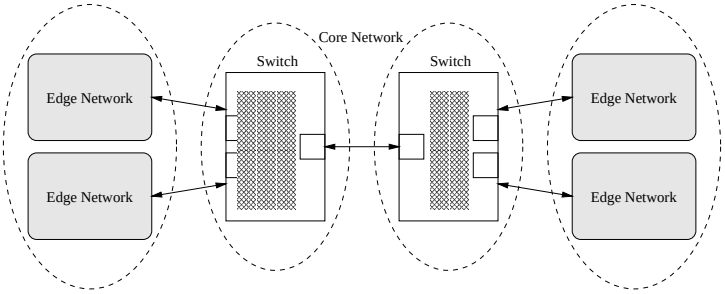

Figure 4.17: Scenario A - 4 Host Mapping

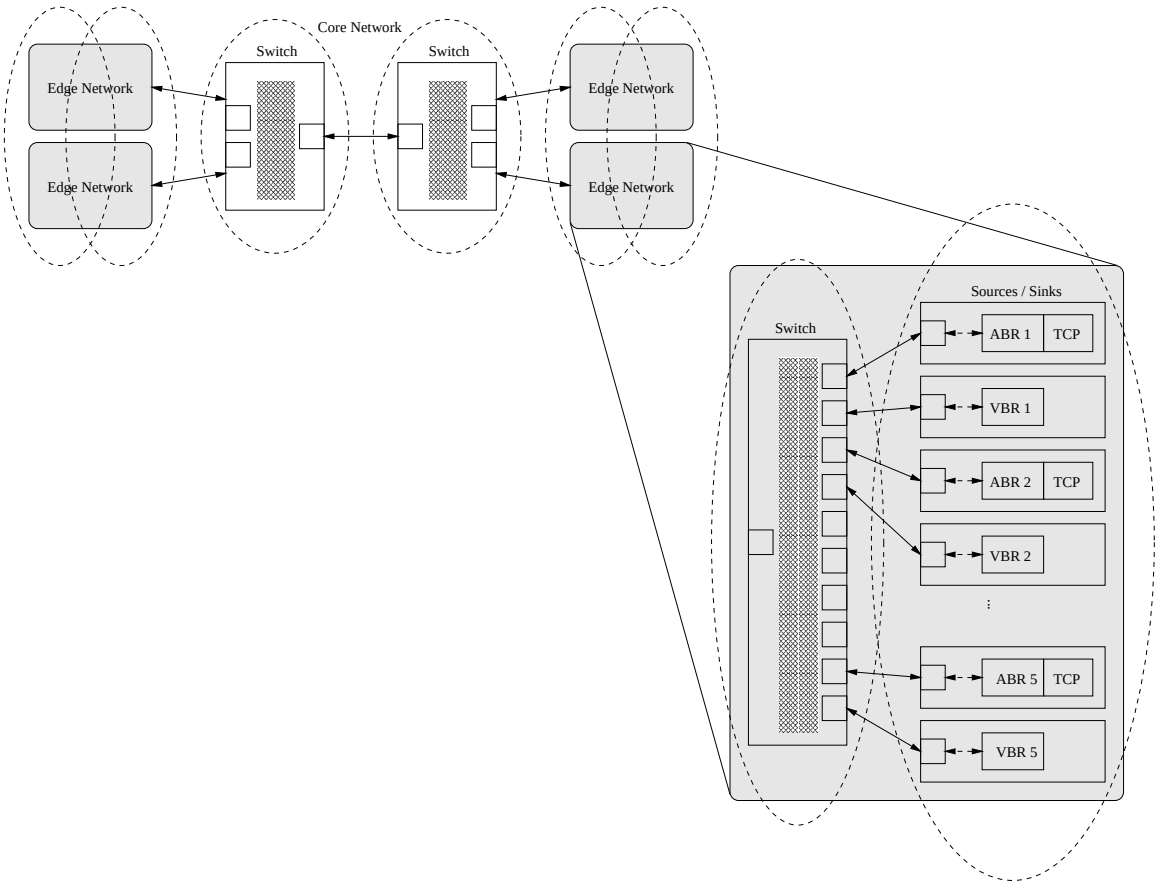

Figure 4.18: Scenario A - 6 Host Mapping

uni-directional cases, depending on the differences in execution times of sources versus sinks. The four processor mapping offloads the edge networks on each side of the network onto their own processor. As the results will show, this is not a very good mapping, because it is very poorly load balanced; the processors with the edge networks have *much* more work to do in each epoch than the core switches do. The six processor mapping additionally moves the sources and sinks onto a processor of their own. While this is slightly better balancing than the four processor mapping, it may or may not be the best six processor mapping. Load balancing a simulation involves tradeoffs between both execution time and communication costs.

Table 4.11 and Figure 4.19 show the execution time results on both simulation platforms. There are several interesting results to point out here.

Experiment	# Processors	Execution Time (seconds)	
		GTW	ProTEuS
Uni-directional	2	1551.75	1191.32
ABR	4	1228.38	1213.28
Traffic	6	610.33	1055.88
Bi-directional	2	2622.97	1548.12
ABR	4	2177.81	1540.79
Traffic	6	1134.29	1221.99
Uni-directional	2	1600.48	1234.22
TCP over ABR	4	1649.88	1243.12
Traffic	6	663.93	1070.77
Bi-directional	2	3016.11	1540.10
TCP over ABR	4	2730.70	1502.08
Traffic	6	1488.28	1200.08

Table 4.11: Scenario A: Execution Time (10 simulated seconds)

First of all, notice that in all four scenarios, ProTEuS achieves essentially no speedup when the number of processors is increased from two to four. As alluded to earlier, this is a direct result of the unbalanced nature of the four processor mapping. In this mapping, the core switches are offloaded to the two additional processors, but this does very little to lighten the load on the other two processors that host the edge networks. In fact, in some scenarios, this actually increases simulation time due to increased communication costs. Increasing the number of processors increases the number of

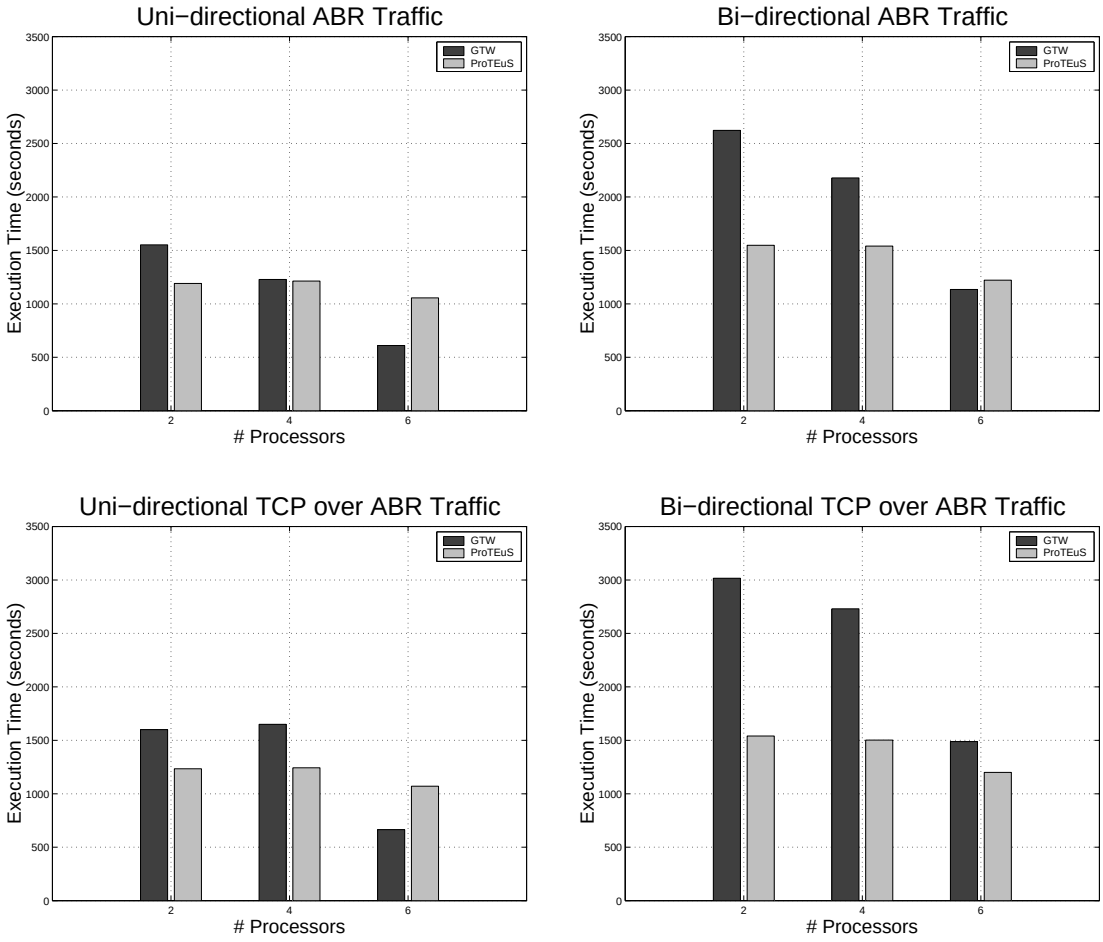

Figure 4.19: Scenario A: Execution Time (10 simulated seconds)

synchronization messages sent and received by each processor, which in turn increases the number of interrupt service routines executed on each host. Increasing the number of interrupts increases scheduling jitter and therefore disruption in the simulation time line, especially when attempting to maintain fine-grained real-time periods.

Further, notice that while ProTEuS does achieve some speedup in the six processor mapping, it is less significant than the speedup achieved by GTW; this is our first introduction to the ProTEuS *floor function*. The floor function exists because there are practical lower limits on the size of the real-time period that the system can reliably support, due to scheduling granularity, scheduling jitter and communication overhead. So, while in theory the six processor mapping could theoretically utilize a smaller real-time period due to its decreased load, the period cannot be reliably decreased past the "floor". This is one reason why ProTEuS actually performs *better* with higher loads. Compare, for instance, the uni-directional cases with the bi-directional cases, where even though the load on the simulated network doubles, which causes the execution time in GTW to essentially double, the ProTEuS execution time increases by a much smaller percentage; approximately 30% in the two processor case and as little as 15% in the six processor case, despite an increase in load by a factor of two. Of course, this marginal increase in simulation time, especially in the six processor mapping, is further evidence of the floor function in the lightly loaded uni-directional scenarios.

Another analogous trend in the results is the trend of execution times as the complexity of the simulation increases from uni-directional ABR, to uni-directional TCP over ABR, to bi-directional ABR, to bi-directional TCP over ABR. Notice that as the complexity and load of the simulated system increases through this progression, GTW is penalized much more heavily than ProTEuS. Pictorially, in Figure 4.19, the height of the bars representing the executing times of GTW "grow" at a much faster rate than those representing ProTEuS. This is actually a combination of the doubled load, as discussed earlier, and the added complexity that TCP brings to the simulation. Table 4.11 shows clearly that GTW takes a significant performance hit when the sources are TCP; as little as 5% in some cases, and as much as 30% in others, as compared to the analogous ABR experiments, depending on the load and mapping. This is due to the

increase in the state-saving overhead of the optimistic protocol that is necessitated by the presence of the TCP protocol. The effect of TCP on ProTEuS, however, is largely insignificant because ProTEuS is essentially source agnostic. ProTEuS suffers a 3% increase or decrease in the most significant cases, which is well within the range of random variation.

Furthermore, it is difficult to draw too much from a direct comparison between the absolute values of execution times in the two systems, due to the differences in the hardware that the two platforms employ. While it is true that the 168 MHz 8-processor shared-memory multiprocessor on which GTW executes has a slightly lower clock frequency than the 200 MHz workstations of the ProTEuS NOW, it is also true that it exhibits *significantly* smaller communication overhead than a NOW, giving GTW an immediate and obvious hardware advantage over ProTEuS. Regardless, the scaling trends are clearly in ProTEuS' favor.

4.4.2 Scenario B

Scenario B is a 16 ATM switch, 120 host edge-core network topology consisting of a four switch, fully-connected core depicted in Figure 4.20. The set of experiments presented here consist of the same four traffic scenarios presented in Section 4.4.1, uni- and bi-directional ABR and uni- and bi-directional TCP, but the routing of traffic is slightly different.

In this topology there are 12 edge networks, three per core switch, each of which is logically peered with an edge network located on another core switch. Each edge network on core switch A routes its traffic through a different destination core switch. The traffic from one edge network on core switch A would send to a destination edge network on core switch B, another to an edge network on core switch C, and the last through core switch D. This keeps the traffic in all parts of the network balanced and symmetrical. The assignment of peer relationships between edge networks was completely random, as was the assignment of source and sink duties in the uni-directional cases. In the bi-directional scenarios, the peer relationships between edge networks remain the same, but each edge network acts as both a source and a sink, just as in the

experiments of Scenario A.

Table 4.12 shows the parameters for the simulations, which are substantially the same as those from Scenario A, except that the source rates have been adjusted to the new topology. In this scenario, each link through the core network is shared by five ABR streams and five VBR streams whose combined rates should hover near the capacity of the link (365000 cells per second) in order to stress the simulation engine. The peak rate of each stream is set at 36000 cells per second, producing a peak link utilization of nearly 99%. The average utilization is approximately 295000 cells per second, or about 80%, due to the variability of the VBR sources, which are 50% duty cycle square waves. All simulations were run for 1 simulated second, as opposed to the 10 second simulations of Scenario A in Section 4.4.1, simply because, due to hardware limitations, GTW was *unable* to complete 10 second simulations of all of the scenarios in this section.

Simulated Line Rate	365000 cells per second (cps) ~ OC-3
ABR Sources	Greedy: PCR=36000 cps, ICR=25% PCR, MCR=0 cps
TCP Sources	Greedy: Window=128KB
VBR Sources	50% square: Min/Max=10000/36000, period=100ms
ABR Feedback Mechanism	EPRCA: Thresh_{low}=200 cells, Thresh_{high}=300
Switch Queue Lengths	Maximum=5000 cells
Switch Queueing Discipline	Per-class: VBR::ABR=200::1
Link Delays	5ms fixed delay
Simulated Time	1 second

Table 4.12: Scenario B Simulation Parameters

Furthermore, as in the previous scenario, each of the four scenarios in this section was run on both ProTEuS and GTW using three different physical mappings; two processors, four processors and six processors. The virtual to physical entity mappings for each of these are shown in Figures 4.21, 4.22 and 4.23, respectively, where the dotted ellipses demonstrate the virtual to physical mappings. Additionally, to showcase the flexibility and scalability of the ProTEuS architecture, the ProTEuS experiments were also mapped onto 16 processors, whose mapping is shown in Figure 4.24.

The two and four processor mappings are again quite natural and result in excellent

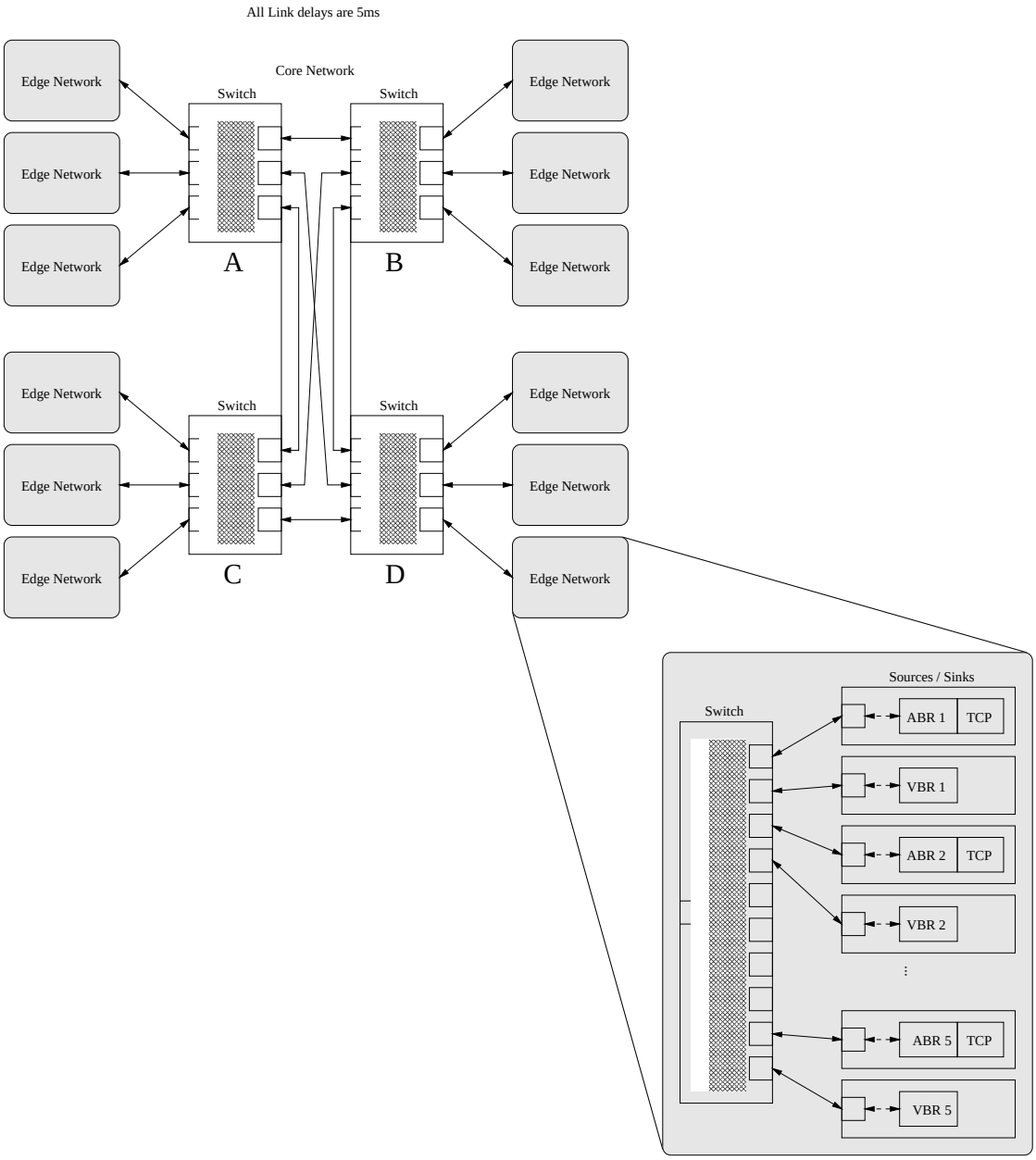

Figure 4.20: 16 Switch 120 Host Scalability Comparison Topology

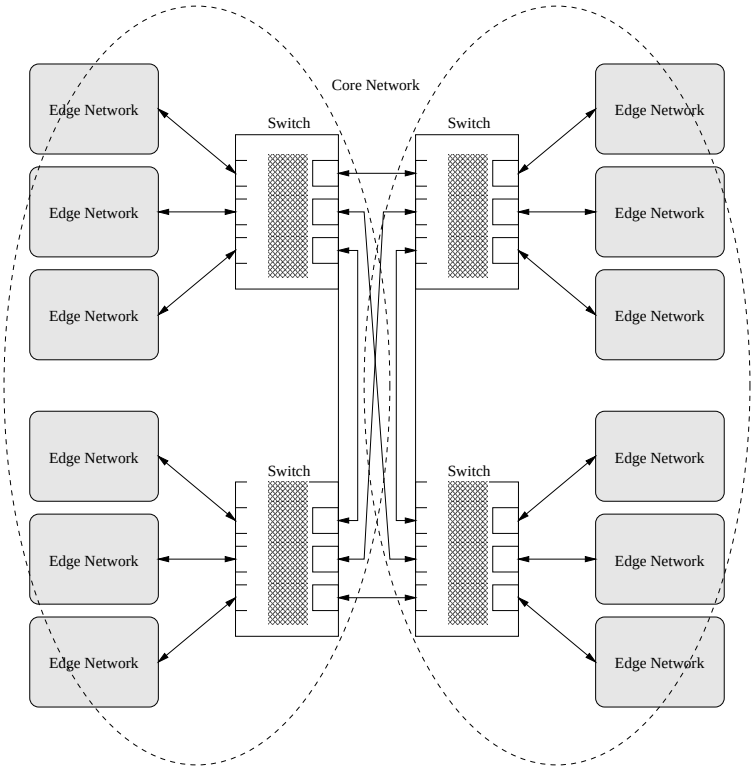

Figure 4.21: Scenario B - 2 Host Mapping

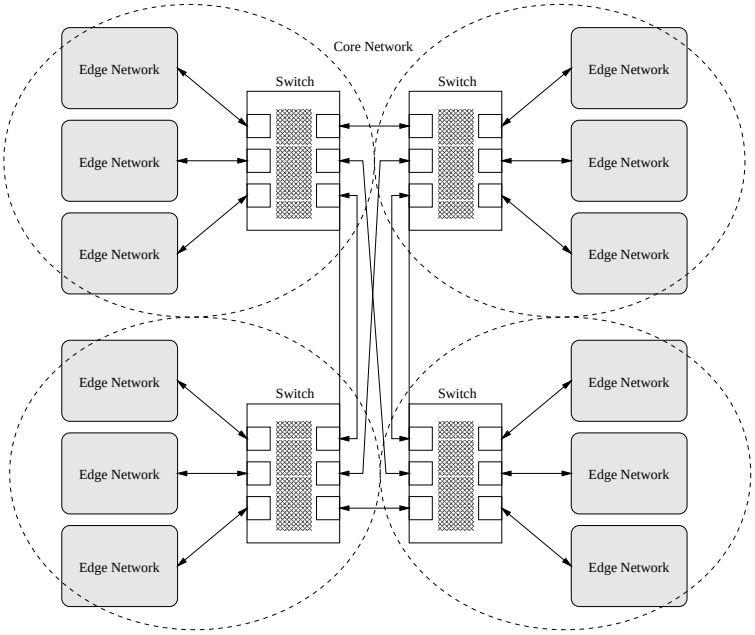

Figure 4.22: Scenario B - 4 Host Mapping

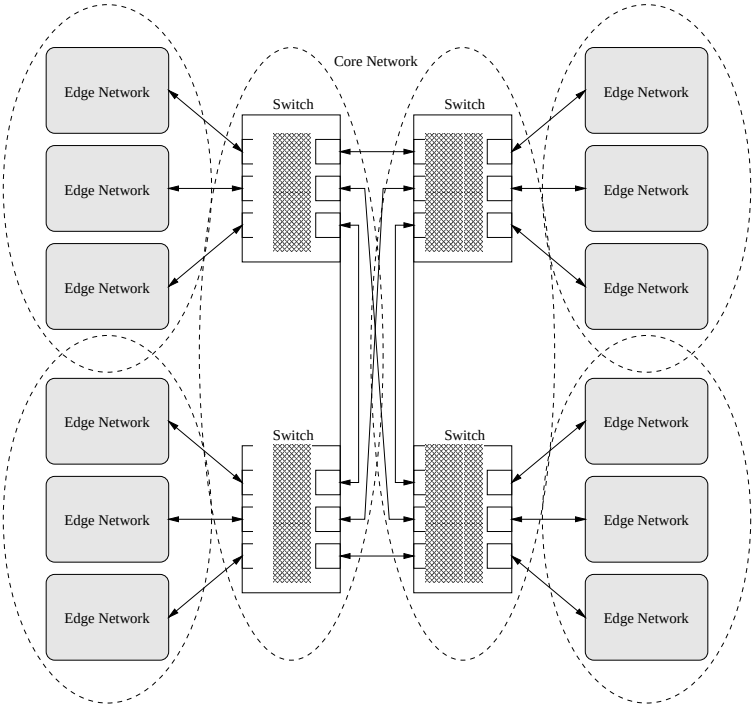

Figure 4.23: Scenario B - 6 Host Mapping

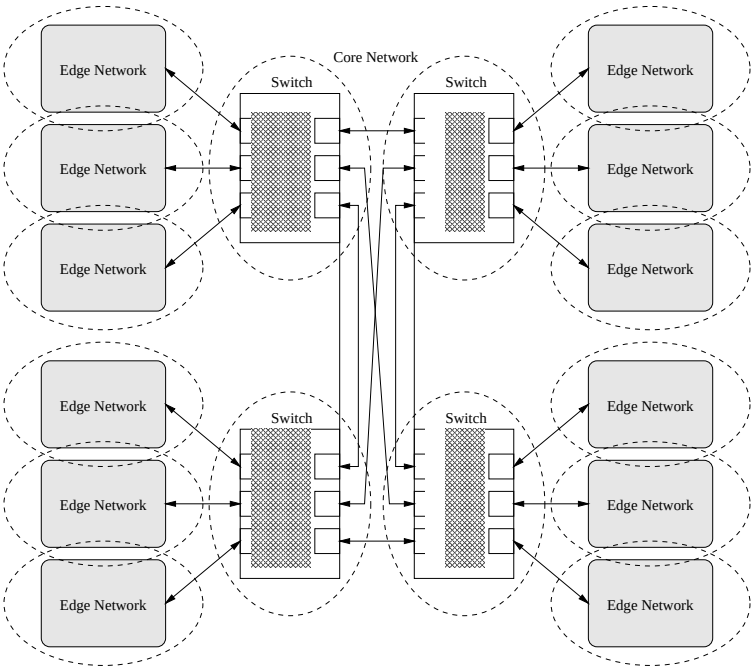

Figure 4.24: Scenario B - 16 Host Mapping

load balancing among processors. The six processor mapping also achieves good load balancing, although it may or may not be evident from the figure. Although each of the two processors hosting two core switches handles essentially double the event load of any processor hosting three edge networks, due to other factors, such as the number of simulation elements on each processor and communication overhead, it still achieves a good balance. This also highlights one of the many differences between GTW, which is most heavily dependent on the number of simulated events, and ProTEuS, which is less sensitive to load and more sensitive to communication costs due to its NOW architecture. While the 16 processor mapping achieves only mediocre load balancing, it is a natural and straight-forward distribution, and really the only sensible mapping across 16 processors for a simulated topology this small. It is worth noting that the machines supporting the four core switches in the 16 processor mapping, but not in the other three mappings, were 500 MHz Pentium III Linux workstations with 256 MB of memory. This was a strategic decision in that not only will they be the most heavily loaded machines (insofar as the number of events) in the simulation, they also have the

highest communication overheads, each communicating with six other nodes in the simulation. The faster processor speeds up all execution, including interrupt service routines, to offset the heavier load. This also illustrates an important advantage of a NOW architecture, which is its ability to be easily upgraded or extended in a piecewise manner. NOWs also make mapping a simulation extremely flexible and allow the balancing of simulation load to take into account factors that an SMMP simulation often cannot, such as processor speed.

Table 4.13 and Figure 4.25 show the execution time results on both simulation platforms. The results from these experiments, combined with those from Section 4.4.1 reveal a great deal about the two simulation platforms and their scaling properties.

Experiment	# Processors	Execution Time (seconds)	
		GTW	ProTEuS
Uni-directional *ABR* Traffic	2	762.88	327.14
	4	385.36	239.43
	6	298.47	178.87
	16	*N/A*	126.05
Bi-directional *ABR* Traffic	2	1569.48	527.29
	4	851.44	335.64
	6	662.42	257.88
	16	*N/A*	159.79
Uni-directional *TCP over ABR* Traffic	2	784.74	349.75
	4	425.72	241.39
	6	331.21	178.66
	16	*N/A*	126.03
Bi-directional *TCP over ABR* Traffic	2	1535.20	549.22
	4	871.57	327.74
	6	668.90	251.07
	16	*N/A*	148.47

Table 4.13: Scenario B: Execution Time (1 simulated second)

The first observation evident in the results from Scenario B is the clear improvement in the speedup achieved by ProTEuS. Through better load balancing and heavier load per processor to eliminate the effects of the floor function, ProTEuS attains speedups nearly those typically achieved by GTW. Further, once again, because of its strong dependency on the number of events processed, GTW is exhibiting a well-known and

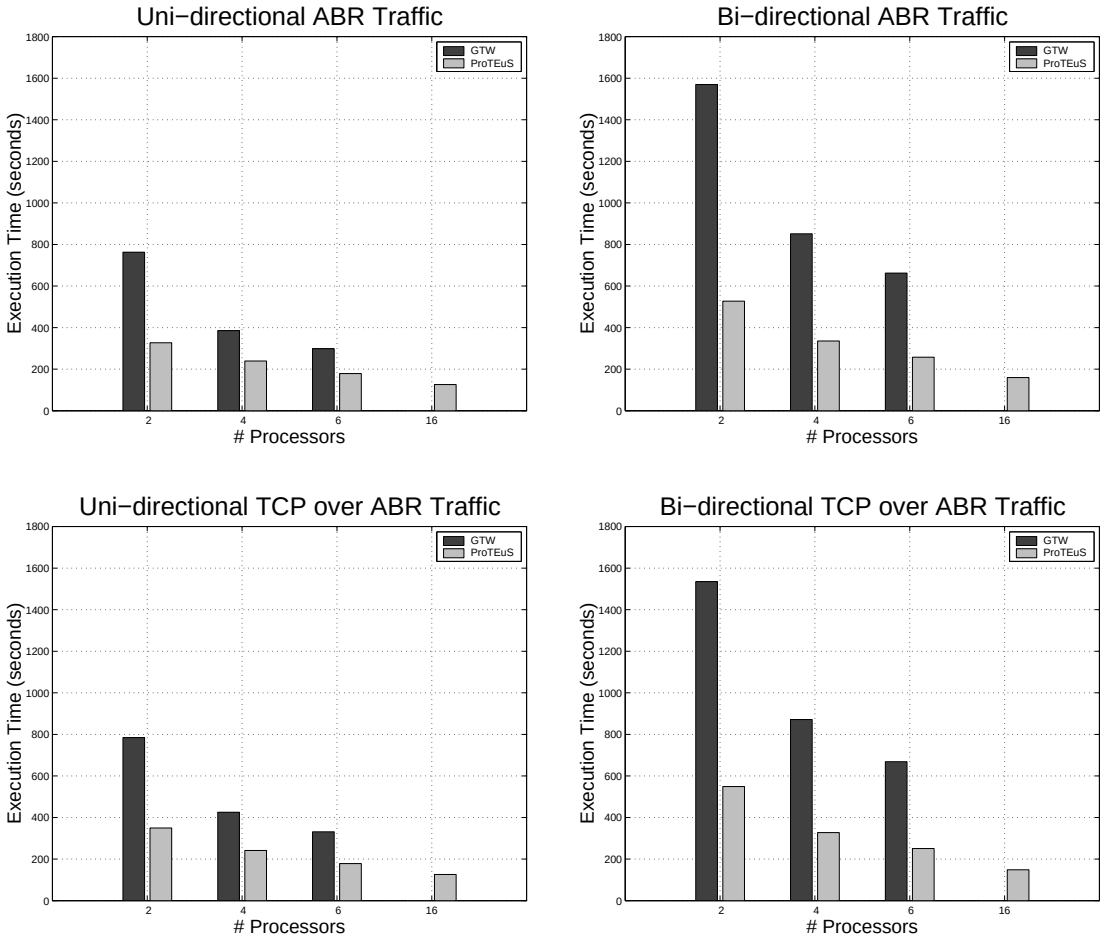

Figure 4.25: Scenario B: Execution Time (1 simulated second)

expected performance penalty as the traffic load doubles. Namely, when the event load doubles, so does the GTW execution time. ProTEuS, however, takes a much smaller hit, from 60% to as little as 35%, depending on the scenario and its distribution.

Furthermore, it is clear, especially from the graphs in Figure 4.25 that, hardware differences aside, ProTEuS is clearly dominating and out-performing GTW. For instance, in the bi-directional traffic cases, ProTEuS' execution times are less than half of GTW *even in the 6 processor cases*.

Lastly, to illustrate the flexibility of the Proportional Time platform, Scenario B was mapped across 16 commercial off-the-shelf (COTS) Linux workstations connected by an ATM network. As the results clearly show, despite the additional communication overheads in such a distribution, ProTEuS continues to achieve speedup. Consider the cost difference between 16 COTS Linux workstations, which are general purpose and are not limited to high performance simulation, and a 16 processor shared-memory multiprocessor, and the appeal of such a simulation platform is evident.

4.4.3 Execution Time vs. Network Size

Two of the most fundamental issues motivating the creation of ProTEuS was the fact that conventional simulation techniques do not scale well with size of the simulated network, nor with simulated time. In this section, results from Scenario A and Scenario B, which differ in size by roughly a factor of three, are compared to evaluate the ability of both simulators to scale with network size.

For these experiments, only the mapping across six processors is presented and each of the four traffic scenarios was run for 10 simulated seconds. The results are shown in Table 4.14 and Figure 4.26.

The most obvious result here is the distinct differences in scalability with network size. In the uni-directional cases, GTW exhibits nearly a factor of six performance penalty between Scenario A and Scenario B, while in the bi-directional scenarios, its almost seven. ProTEuS, on the other hand, takes a hit of less than two times in the uni-directional cases, and just over 2 in the bi-directional cases. However, we speculate that the ProTEuS scaling factors *may* be slightly off due to somewhat inflated execution

Experiment	Network Size (Scenario)	Execution Time (seconds)	
		GTW	ProTEuS
Uni-directional *ABR* Traffic	6 Switch / 40 Host (A)	610.33	1055.88
	16 Switch / 120 Host (B)	3520.28	1754.40
Bi-directional *ABR* Traffic	6 Switch / 40 Host (A)	1134.29	1221.99
	16 Switch / 120 Host (B)	7845.38	2528.08
Uni-directional *TCP over ABR* Traffic	6 Switch / 40 Host (A)	663.93	1070.77
	16 Switch / 120 Host (B)	3834.70	1873.59
Bi-directional *TCP over ABR* Traffic	6 Switch / 40 Host (A)	1488.28	1200.08
	16 Switch / 120 Host (B)	*N/A*	2579.70

Table 4.14: Execution Time vs Network Size (10 simulated seconds)

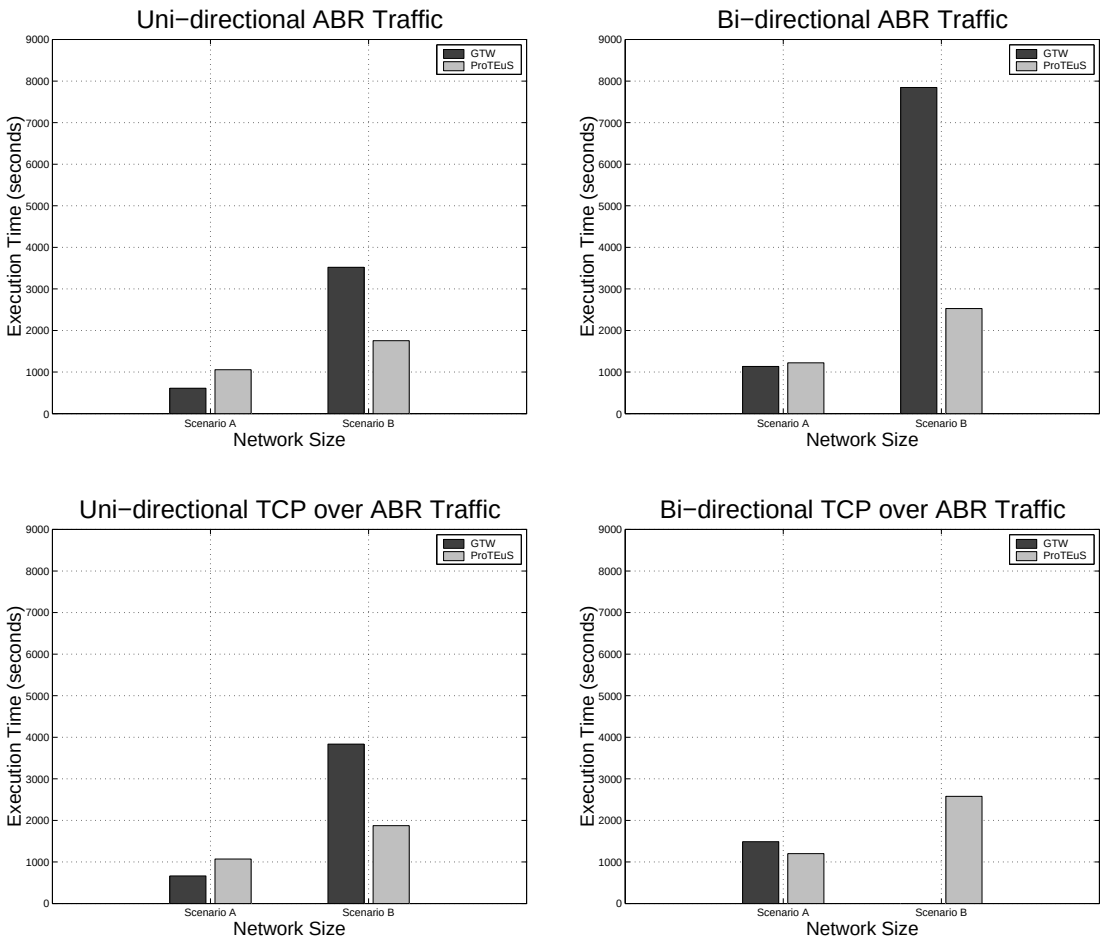

Figure 4.26: Execution Time vs. Network Size (10 simulated seconds)

times in Scenario A attributed to the effects of the floor function. However, execution time profiling of Scenario A has shown that, at worst, it may be as high as a factor 2.5 or thereabouts. Further, notice that GTW was *unable* to complete a 10 second simulation in the Scenario B, bi-directional TCP over ABR case due to memory limitations. When the system reached the bounds of physical memory and started invoking disk swapping, the simulation was effectively stopped. Keep in mind that this SMMP machine was an 8-processor 168 MHz Sun with 1 GB of main memory - not, by any standard, meager computing power.

It is quite clear, especially looking at Figure 4.26, that ProTEuS is exhibiting superior scaling with network size *for these scenarios*. While we don't presume that ProTEuS scales this much better than GTW for all applications, or even for all network models, we do believe that it will produce superior scaling in many of them.

The other very interesting result comes from a comparison between Table 4.13, which contains one second simulations, and Table 4.14, which contains 10 second simulations, both of Scenario B. It is interesting to note that GTW does not scale linearly with simulated time either. For instance, while a one second simulation of uni-directional ABR traffic took GTW 298.47 seconds to complete, a 10 second simulation of the same model took 3520.28 seconds; not 10 times longer, but nearly 12 times longer. For the same scenario, ProTEuS takes 178.87 seconds for a one second simulation and 1754.40 for a 10 second simulation; slightly less than 10 times longer. ProTEuS appears to be slightly better than linear for the ABR cases and slightly more than linear for the TCP over ABR cases; overall, a linear scaling with time. In fact, ProTEuS should scale *precisely* linearly with time if the number of missed epochs remained constant with time as well; variations in the number of missed epochs will result in variations in execution time.

4.4.4 Execution Time vs. Round Trip Time

In this section, perturbations of a fundamental parameter of a network simulation model are performed to demonstrate their effect on different simulation methodologies. For these experiments, the topology and parameters of Scenario A (Section 4.4.1,

page 117) were used, while varying the round-trip time between hosts. The round-trip time is divided evenly between the 10 one-way delays in a round-trip. For instance, a round-trip time of 50ms is modeled as 5ms one-way delays on each link in the network. All simulations were mapped across six processors and rather than running them for a fixed length of time, which results in an unfair comparison between ProTEuS and GTW, the simulations were run for a fixed number of data cells. In the ABR scenarios, the simulation proceeds until the last ABR source sends its 100000th (10^5) data cell. In the TCP over ABR simulations, the simulation stops when the last TCP source sends 5 MB of data. In both cases, the VBR sources continue to send their background traffic for the duration of the experiment. The results are shown in Table 4.15 and Figure 4.27.

Experiment	Round-Trip Time (ms)	Execution Time (seconds)	
		GTW	ProTEuS
Uni-directional *ABR* Traffic	10	330.44	496.00
	50	333.67	513.50
	100	341.49	513.46
	200	366.40	518.33
	400	398.47	533.03
Bi-directional *ABR* Traffic	10	609.16	585.95
	50	649.27	603.76
	100	651.15	602.05
	200	708.41	617.11
	400	857.33	643.14
Uni-directional *TCP over ABR* Traffic	10	307.28	565.12
	50	330.40	595.94
	100	349.24	633.44
	200	389.14	702.97
	400	470.61	858.66
Bi-directional *TCP over ABR* Traffic	10	574.69	656.51
	50	648.79	684.89
	100	703.38	754.35
	200	835.71	802.07
	400	1017.36	1098.67

Table 4.15: Execution Time vs. Round Trip Time

First and foremost, it is certainly expected that GTW will suffer as a result of increase round-trip time due to several factors, including rollback and state-saving. Pro-

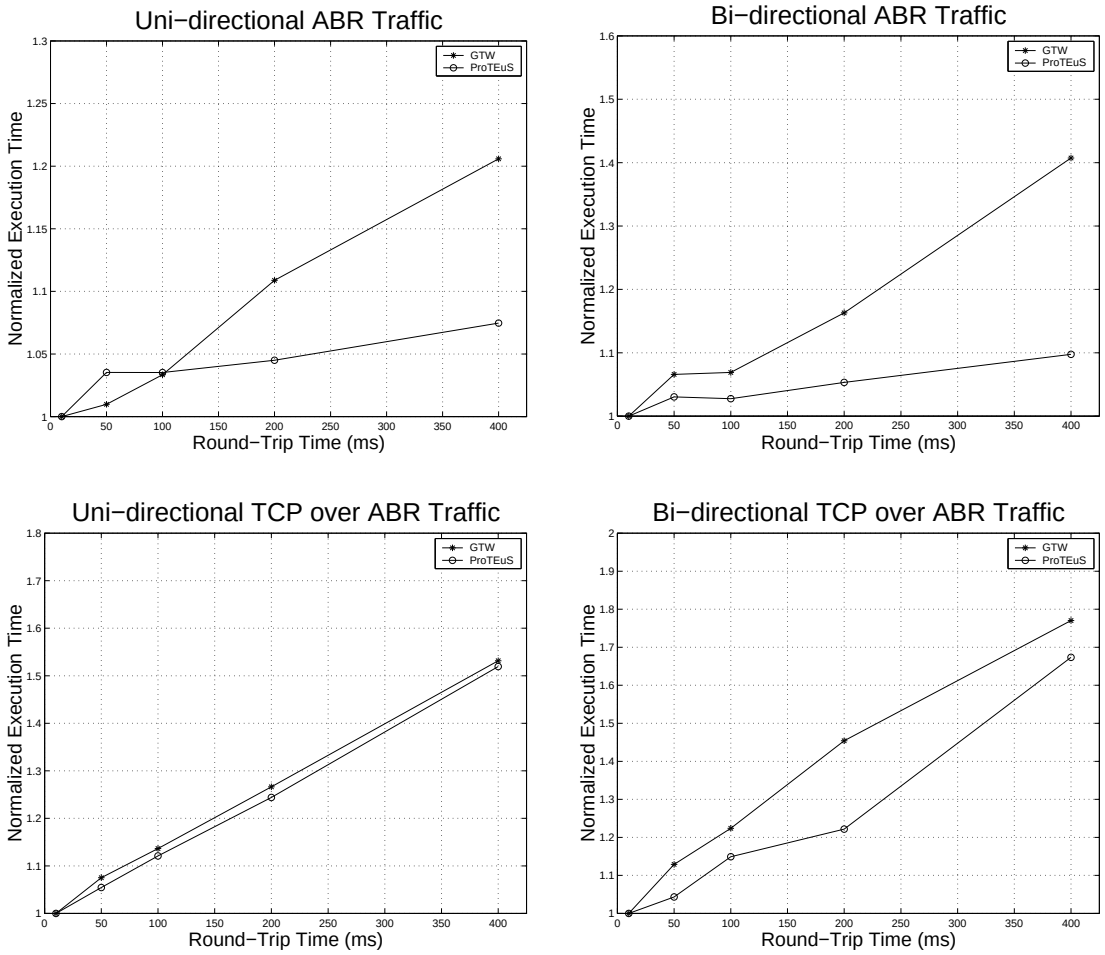

Figure 4.27: Normalized Execution Time vs. Round Trip Time: ABR sources

TEuS, however, has no real dependency on round-trip time, and therefore should not show any real adverse effects as round-trip time increases. The results paint a very intriguing picture. *Note: Realize that the graphs in Figure 4.27 are Normalized Execution Time vs. Round-Trip Time, where the execution time for each round-trip time was divided by the execution time of the 10ms round-trip time case.*

In the ABR scenarios, the results are essentially what is expected; GTW performance takes a significant hit as round-trip time increases, while ProTEuS experiences an almost negligible performance penalty. However, when TCP traffic is introduced over ABR, the results are completely different. GTW continues to worsen with increased round-trip time, reaching increases of nearly 80% in the bi-directional case, but now ProTEuS also suffers a performance impact. The interesting part is, *why*?

In an ABR simulation, ProTEuS is basically immune to changes in round-trip time; it doesn't significantly affect the length of simulated time necessary to send N data cells, which was the stopping condition for these experiments. Of course, as shown in Section 4.3.1, delayed ABR feedback certainly can make *some* difference on link utilization, but not in a scenario in which the feedback mechanism is never invoked. It is likely that the very slight increase in simulation time for ProTEuS is due to the delayed ABR feedback during ramp-up, which lengthens the time necessary for ABR sources to ramp-up to their peak cell rates. GTW, on the other hand, experiences a performance penalty for well-known reasons, including increased probability of rollback and increased state saving overhead mandated by the optimistic control.

In a TCP over ABR simulation, GTW is essentially in the same situation (made worse by the added protocol layer), but ProTEuS is penalized by being placed at the whim of protocol idiosynchracies, such as windowing mechanisms and Van Jacobsen's slow-start. Because it is a conservatively synchronized simulator, ProTEuS doesn't race ahead across idle periods optimistically like GTW does. Therefore, ProTEuS actually waits for TCP ACKs to return before sending out more data, which obviously takes longer as the round-trip time increases. So, for these specific experiments, because of TCP slow-start, ProTEuS takes longer to send N data cells, which increases the simulated period. In the bi-directional TCP scenario, the ABR source rate update and ACK

delay is compounded by the fact that TCP ACKs are now forced to wait in queues behind data cells in the same direction generated by the peer source to which the ACK belongs.

The bottom line on round-trip time and TCP is a *very* important observation, which is that the two platforms exhibit very similar scaling behavior, but *for completely different reasons*. GTW suffers a performance penalty because of *the properties of the simulation engine itself*. ProTEuS, on the other hand, suffers as a result of *the properties of the system which it is simulating*. This is manifest in the difference between the ABR and TCP over ABR results; it is TCP, and to a lesser extent ABR, that has the dependency on round-trip time, *not ProTEuS*.

Chapter 5

Conclusions and Future Work

It is our desire, as well as that of many others, to evaluate the performance of ATM networks of significant size and complexity under varying conditions and architectures. There have been many attempts to solve this problem, including sequential discrete event simulation techniques and many flavors of parallel discrete event simulation. Sequential discrete event simulations, however, often do not scale well in two important respects; the size of the network and the length of simulated time. Networks of significant size become increasingly difficult to *even model* in sequential discrete event systems and their size significantly affects the run-time of a simulation. Parallel discrete event simulations, Time Warp in particular, are quite promising with respect to scalability, however, Time Warp is subject to poor performance when the computation is fine-grained, as it is in many network simulation scenarios, including ATM. Further, systems with significant feedback cause interactions among distributed components that can result in significant overhead in Time Warp systems due to frequent state-saving and rollback.

The real problem is how to efficiently simulate large networks for long periods of simulated time. It is our contention, as well as that of many others, that faster single processors are not adequate to offset the scaling of the simulation. Further, current methods for parallelizing such simulations are not entirely satisfactory for some simulated systems, including our driving applications. Our approach to improving performance is an innovative application of real-time and embedded techniques producing

a system which supports parallel simulations executing in proportional time, named ProTEuS: Proportional Time Emulation and Simulation.

ProTEuS uses real-time, distributed system, and embedded system techniques to create synchronized distributed proportional time simulations. Because the simulated system is distributed across any number of physical machines, it is orders of magnitude faster than conventional sequential discrete event simulation experiments and shows scalability tendencies superior to those of Time Warp implementations in many significant situations. Furthermore, ProTEuS uses the *real code* that networks and systems could, and often do, use. ProTEuS ATM simulations use a real operating system protocol stack and the ATM signaling support is the same as that used in an off-board signaling architecture. Because it uses real system networking code, it does not require the implementation of system code abstractions in a software simulator, avoiding the pitfall of excessive abstraction suffered by many other discrete event simulators. The only changes necessary to utilize existing system code in ProTEuS are those required to support the notion of virtual time, such as those described in this work pertaining to the TCP/IP stack in Linux. ProTEuS uses commercial off-the-shelf PC hardware running Linux at a modest cost and all of the necessary software is free and open source, including the real-time platform.

This work has demonstrated the ability of the ProTEuS platform to produce simulation results extremely similar, arguable identical, to those produced by conventional methods. It has also explored some of the properties of synchronous distributed computation using KU Real-Time modifications to Linux (KURT-Linux) as a platform and discovered several methods by which to improve performance, uncovered interesting properties of such a distributed real-time system, and identified areas that need attention to further improve performance. Furthermore, it has established the capacity of ProTEuS to scale linearly with simulated time, a feat that Georgia Tech Time Warp (GTW) is unable to match in the scenarios modeled herein. It has also showcased scalability in network size superior to that of GTW, even for the relatively small networks simulated thus far. Lastly, it has exposed a fundamental difference between the properties of GTW and ProTEuS that was a well-known fact, but manifest in round-trip

time perturbations.

In short, ProTEuS is a simulation platform that shows noteworthy promise regarding the two fundamental problems of conventional simulation techniques; scaling with network size and simulated time. It does it without the loss of generality, without the loss of verity due to over-abstraction and without the addition of prohibitive hardware needs. It is further worth mentioning that the version of ProTEuS evaluated herein should be considered beta quality, i.e. a proof of concept implementation, meaning that *no* optimization has been attempted whatsoever, as opposed to the obvious maturity of a product like GTW. It can only get better.

5.1 Future Work

The Proportional Time platform is truly in its infancy and a multitude of enhancements and optimizations are possible. This is a brief list:

- Separate the ATM-specific ProTEuS implementation. Because ProTEuS was initially designed for the simulation of ATM networks, there exists a tight coupling between the two in the current implementation. Separating the two layers, Proportional Time and ATM Quality of Service, and providing a generic interface will open a door to a wide range of applications for which a general distributed real-time communication framework is useful.

- Create a Master-Slave protocol between simulation nodes. By assigning one machine the duty of simulation master, the simulation can be controlled from a single point, for instance, to pause or stop the simulation under certain conditions, or to control the start time of the simulation from a master to alleviate some of the clock synchronization and race condition issues that have already been encountered.

- Add support for virtual ethernet devices. The current virtual network device implementation has support only for virtual ATM devices. Virtual ethernet devices will allow the application of ProTEuS techniques to the simulation of IP networks.

- Optimize critical paths of execution. This includes more efficient methods of demultiplexing virtual device traffic and the preallocation of packets (cells) to avoid dynamic allocation wherever possible.

- Dynamically control the epoch lengths. This is a key enhancement that is necessary for several reasons; (1) it will allow the simulation to track its own performance and make adjustments when necessary, for example increasing the epoch size when too many epochs are being missed, and (2) it will eliminate the need to manually probe for the right epoch length to use for a particular simulation.

- Provide better clock synchronization. This includes both synchronization in absolute value, but also controlling the rate at which time progresses. Further, combined with the dynamic control of real-time periods, this could include the adjustment of epoch phase to mitigate the effects of virtual time lines interleaved in absolute time.

- Exert better control over Linux bottom halves to decrease scheduling jitter. This may include selectively turning them off, or a more sophisticated solution such as delegating and scheduling a thread whose sole purpose is to run bottom halves. Because thread scheduling is controlled by KURT-Linux, this can prevent bottom halves from interfering in the execution of fine-grained real-time applications.

- Extend the notion of proportional time to other simulation domains. This includes the obvious extension to the simulation of native IP networks, but also includes other non-networking domains such as data collection, hardware simulation and distributed virtual environments. Once generalized, ProTEuS is applicable to any application with distributed real-time communication needs.

- Add support for mismatched line rates. In the current ProTEuS implementation, every link in the simulation must exhibit the same simulated line rate. Of course, most real-world networks have several line rates depending on the proximity to the core. This will require some bookkeeping as well as some notion of proportionality between links.

- Create tools for visualization, topology and script generation, etc. Sooner or later, as the scenarios get larger and larger, ProTEuS will need front-end applications to perform tasks like automatic topology generation, parameter assignment and testbed/network configuration, as well as NetSpec script generation and result visualization.

Bibliography

[1] ATM Forum Traffic Management Specification Version 4.0. ATM Forum Techincal Committee, Traffic Management Sub-Working Group, April 1996.

[2] A. Atlas and A. Bestavros. Design and Implementation of Statistical Rate Monotonic Scheduling in KURT Linux. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1999.

[3] R. Bagrodia and V. Liao. Maisie: A Language for the Design of Efficient Dicrete Event Simulation. *IEEE Transactions on Software Engineering*, April 1994.

[4] R. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, B. Park, and H. Song. Parsec: A Parallel Simulation Environment for Complex Systems. *IEEE Computer Magazine*, 31(10):77–85, October 1998.

[5] R. Bagrodia and X. Zeng. Glomosim: A Library for the Parallel Simulation of Large Wireless Networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, 1998.

[6] Bell Communications Research, Inc. *Q.Port: Portable ATM Signaling Software*, 1996.

[7] S. Bhatt, R. Fujimoto, A. Ogielski, and K. Perumalla. Parallel Simulation Techniques for Large-Scale Networks. *IEEE Communications Magazine*, pages 42–47, August 1998.

[8] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. *IEEE Computer Magazine*, 33(5):59–67, May 2000.

[9] B. Buchanan, D. Niehaus, G. Dhandapani, R. Menon, S. Sheth, Y. Wijata, and S. House. The Datastream Kernel Interface (Revision A). Technical Report ITTC-FY98-TR-11510-04, Information and Telecommunication Technology Center, University of Kansas, 1994 June.

[10] Cadence Design Systems, Inc., http://www.cadence.com. *BONeS Designer Modeling Reference Guide*, 1993.

[11] K.M. Chandy and J. Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, 5(5), September 1979.

[12] K.M. Chandy and J. Misra. Asynchronous Distributed Simulation via a Sequence of Parallel Computations. *CACM*, 24(11), April 1981.

[13] M. Chetlur, N. Abu-Ghazaleh, R. Radhakrishnan, and P. Wilsey. Optimizing Communication in Time Warp Simulators. In *12th Workshop on Parallel and Distributed Simulation*, pages 64–71, May 1998.

[14] M. Chong. Optimistic Parallel Simulation of TCP/IP over ATM Networks. Master's thesis, University of Kansas, November 2000.

[15] J. Cowie, H. Liu, J. Liu, D. Nichol, and A.T. Ogielski. Towards Realistic Million-Node Internet Simulations. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, June/July 1999.

[16] J. Cowie, D. Nichol, and A.T. Ogielski. Modeling the Global Internet. *Computing in Science & Engineering*, 1(1):42–50, January/February 1999.

[17] S. R. Das, R. Fujimoto, K. Panesar, D. Allision, and M. Hybinette. GTW: A Time Warp System for Shared Memory Multiprocessors. In *Proceedings of the Winter Simulation Conference*, pages 1332–1339, December 1994.

[18] L. Torvalds et al. The Linux Kernel Archives. http://www.kernel.org.

[19] W. Almesberger et al. ATM on Linux. http://icawww1.epfl.ch/linux-atm.

[20] K. Fall. Network Emulation in the Vint/NS Simulator. Technical report, University of California at Berkeley, 1999.

[21] R. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, 1990.

[22] R. Fujimoto. Parallel and Distributed Simulation. In *Proceedings of the Winter Simulation Conference*, pages 118–125, December 1995.

[23] Richard M. Fujimoto. Time Warp on Shard Memory Multiprocessors. *Transactions of the Society for Computer Simulation*, 6(3):211–239.

[24] R. Hill. Improving Linux Real-Time Support: Scheduling, I/O Subsystem, and Network Quality of Service Integration. Master's thesis, University of Kansas, June 1998.

[25] R. Hill, B. Srinivasan, S. Pather, and D. Niehaus. Temporal Resolution and Real-Time Extensions to Linux. Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, University of Kansas, June 1998.

[26] S. House, S. Murthy, and D. Niehaus. Proportional Time Simulation of ATM Networks. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1999.

[27] Imagine That!, Inc., http://www.imaginethatinc.com. *Extend Software Manual*, 1998.

[28] R. Jain, S. Kalyanaraman, S. Fahmy, R. Goyal, and S. Kim. Source Behavior for ATM ABR Traffic Management: An Explanation. *IEEE Communications Magazine*, 34(11):50–57, November 1996.

[29] D. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[30] R. Jonkman. NetSpec: Philosophy, Design and Implementation. Master's thesis, University of Kansas, February 1998.

[31] R. Jonkman, D. Niehaus, J. Evans, and V. Frost. NetSpec: A Network Performance Evaluation Tool. Technical Report ITTC-FY98-TR-10980-28, Information and Telecommunication Technology Center, University of Kansas, December 1998.

[32] S. Kalyanaraman, R. Jain, S. Fahmy, R. Goyal, and B. Vandalore. The ERICA Switch Algorithm for ABR Traffic Management in ATM Networks. *ACM Transactions on Networking*, 8(1):87–98, February 2000.

[33] D. Martin, T. McBrayer, and P. Wilsey. WARPED: A Time Warp Simulation Kernel for Analysis and Application Development. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, volume 1, pages 383–386, January 1996.

[34] E. Mascarenhas, F. Knop, and V. Rego. ParaSol: A Multithreaded System for Parallel Simulation Based on Mobile Threads. In *Proceedings of the Winter Simulation Conference*, December 1995.

[35] R. Menon. Large Scale Distributed Real-Time Computation: A Time Driven Framework. Master's thesis, University of Kansas, January 1998.

[36] Mil3, Inc., http://www.mil3.com. *OPNET Modeler*, 1998.

[37] D. Mills. Network Time Protocol (Version 3) Specification, Implementation and Analysis. RFC 1305, Network Working Group, March 1992.

[38] D. Mills. A Kernel Model for Precision Timekeeping. RFC 1589, Network Working Group, March 1994.

[39] Jayadev Misra. Distributed Discrete-Event Simulation. *Computing Surveys*, 18(1), March 1986.

[40] S. Murthy. A Software Emulation and Evaluation of Available Bit Rate Service. Master's thesis, University of Kansas, October 1998.

[41] D. Nichol. Scalability, Locality, Partitioning, and Synchronization in PDES. In *Proceedings of the Parallel and Distributed Simulation Conference*, 1998.

[42] D. Nichol and R. Fujimoto. Parallel Simulation Today. *Annals of Operations Research*, 53:249–286, December 1994.

[43] R. Pasquini and V. Rego. Efficient Process Interaction With Threads in Parallel Discrete Event Simulation. In *Proceedings of the Winter Simulation Conference*, December 1998.

[44] R. Pasquini and V. Rego. Optimistic Parallel Simulation Over a Network of Workstations. In *Proceedings of the Winter Simulation Conference*, December 1999.

[45] P. Prasithsangaree, G. Dhandapani, K. Kalarickal, V. Barathan, and D. Niehaus. KU PNNI User's Manual. Technical Report ITTC-FY2000-TR-18833-09, University of Kansas, 2000.

[46] R. Radhakrishnan, D. Martin, M. Chetlur, D. Rao, and P. Wilsey. An Object-Oriented Time Warp Simulation Kernel. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments*, pages 12–23, December 1998.

[47] R. Radhakrishnan, T. McBrayer, K. Subramani, M. Chetlur, W. Balakrishnan, and P. Wilsey. A Comparative Analysis of Various Time Warp Algorithms Implemented in the WARPED Simulation Kernel. In *Proceedings of the Simulation Symposium*, pages 10y–116, March 1996.

[48] M. Raguparan. Performance Analysis of Simple ABR Congestion Controllers Augmented with a Minimum Variance Predictor. Master's thesis, University of Kansas, May 1998.

[49] D. Rao and P. Wilsey. Simulation of Ultra-large Communication Networks. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 112–119, October 1999.

[50] L. Roberts. Enhanced Proportional Rate-Control Algorithm. In *ATM Forum Techincal Committee, Traffic Management Sub-Working Group*, number 94-0735R1, 1994.

[51] R. Sanchez. Virtual ATM Switch Driver for ATM on Linux. http://www.ittc.ukans.edu/~rsanchez/software/vswitch.html.

[52] B. Srinivasan. A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software. Master's thesis, University of Kansas, February 1998.

[53] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A Firm Real-Time System Implementation using Commercial Off-The-Shelf Hardware and Free Software. In *Real-Time Technology and Applications Symposium*, June 1998.