ISSUES IN INTEGRATED NETWORK EXPERIMENTATION USING SIMULATION AND EMULATION

by

Shashikiran B. Guruprasad

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

August 2005

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Shashikiran B. Guruprasad

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Jay Lepreau

Gary E. Lindstrom

John S. Heidemann

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Shashikiran B. Guruprasad in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date	Jay Lepreau Chair: Supervisory Committee

Approved for the Major Department

Christopher R. Johnson
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Discrete-event network simulation is widely used to rapidly design, evaluate, and validate new networking ideas as well as study the behavior of existing ones. It is characterized by complete control, absolute repeatability and ease of use while often sacrificing detail and realism to increase execution efficiency and the scale of models. Network emulation allows the study of applications running on real hosts and “somewhat real” networks. A key difference between the two approaches is that in the former, the notion of time is virtual and is independent of real time, whereas the latter must execute in real time. Typically, emulated resources are also distributed in nature. Thus, emulation gains realism while naturally foregoing complete repeatability; historically, emulation was also tedious to control and manage.

Integrated Experiments, where we spatially combine real elements with simulated elements to model different portions of a network topology in the same experimental run, enable new validation techniques and larger experiments than obtainable by using real elements alone.

In this thesis, we present a system in which we employ multiple loosely coordinated simulator instances running on distributed PCs in real-time to model the simulated portion of a network topology. Our key design techniques are to perform optimistic automated resource allocation, and to use feedback to adaptively allocate simulated resources in order for the simulators to run in real-time. Multiple simulator configurations specific to a resource assignment are automatically generated from an experimenter configuration which is agnostic to the details of the physical realization. The entire system is highly automated and is available for production use in Emulab.

To my parents

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

While completing my masters thesis, I have acquired valuable experience and skills which I believe will take me a long way in my career as well as life. However, some people say that in graduate school, one learns about how much more there is to learn than what is actually learnt. My graduate school life has helped me understand the value of intellectual humility, while balancing it with a healthy amount of confidence.

I'm thankful to my advisor, Jay Lepreau, for giving me the opportunity to work in the Emulab project and advising me towards the completion of this thesis. I'm grateful that he had more faith in my thesis work and abilities than I sometimes had in myself. I would like to thank my committee members, John Heidemann and Gary Lindstrom, who asked me the right questions during my thesis proposal and defense that helped me to strengthen the evaluation of my work.

I would like to thank Robert Ricci for developing and improving "assign", entertaining my feature requests of which many were difficult problems and then earnestly coming up with novel solutions very quickly. I'm also thankful to have worked with Mike Hibler and Leigh Stoller in the Emulab project. Having them as my role models helped me immensely to improve my programming skills and work attitude, and I still continue to be inspired by them. I have to thank Brian White for driving many discussions during the initial wireless testbed research and then writing most of our position paper. Thanks to my flux group cohorts: Abhijeet and Parveen, as well as fellow graduate student Hemanth for helping me to see things clearly when I had questions or problems. Mohit, Amit and Piyush deserve special thanks for letting me gatecrash their place for chai and dinner on a regular basis. Thanks to friends (numerous to name here) for all the fun times we had as a group on hiking trips and idle chat sessions that made my graduate school life so memorable. Finally, many thanks to my best friend Niti for providing me constant support and encouragement, especially during difficult times.

CHAPTER 1

INTRODUCTION

There are three experimental techniques used in the design and validation of new and existing networking ideas: simulation, emulation and live network testing. All three techniques have unique benefits and tradeoffs compared to each other. However, they need not be viewed as competing techniques. Using all the three techniques in a process continuum helps validate ideas better than using any one technique alone.

Network simulation provides an absolutely repeatable and controlled environment for network experimentation. It is easy to configure and allows a protocol designer to build at a suitable level of abstraction making simulation a rapid prototype-and-evaluate environment. Such a rapid process allows discarding of many bad alternatives before attempting a full implementation. Ease of use also allows for large parameter-space exploration. Discrete-event simulation, where the simulation state changes only at discrete points in time, is the most commonly used network experimentation technique. *ns* [7, 11] is a widely used discrete-event packet-level network simulator known for the richness of transport, network and multicast protocols, buffer management, QoS and packet scheduling algorithms as well as for models to perform wireless and satellite network experimentation. The accuracy of simulations is dependent on the level of abstraction of the models. Models that incorporate a higher level of detail reduce both execution efficiency and scalability of the simulation. An experimenter is forced to make a tradeoff between accuracy and efficiency without any systematic means of validating the choice of abstraction [23]

Network emulation [5, 48, 41, 42, 55, 18] is a hybrid approach that combines real elements of a deployed networked application, such as end hosts and protocol implementations, with synthetic, simulated, or abstracted elements, such as the network links, intermediate nodes and background traffic. Which elements are real and which are partially or fully simulated will often differ, depending on the experimenter's needs and the available resources. For example, as more components run real code across live networks, one gains increasing realism at the potential cost of scale, control and unpredictability. A fundamental difference between simulation and emulation is that while the former runs in virtual simulated time, the latter must run in real time. Another important difference is that it is impossible to have an absolutely repeatable order of events in an emulation due to its real-time nature and typically, a distributed set of resources. Note that the mere presence of real elements does not necessitate emulation. For example, NCTUns [58] and umlsim [6] are simulators that use only real elements. However, these real elements execute in virtual time bringing the advantages of real implementations and simulation together. What makes emulation a useful experimental technique is that realism is gained by foregoing complete repeatability. Emulation provides an environment that is closer to real environments than simulation. However, emulation is more tractable as an evaluation environment than the real world, such as the Internet.

Live networks such as the Internet provide realistic network conditions. However, they lack repeatability and the ability to monitor or control intermediate routers and links in the network. Despite these drawbacks, researchers need to run experiments over live networks to make their ideas completely credible.

We define integrated network experimentation as *spatially* combining real elements with simulated elements in one or more instances of an existing simulation engine to model different portions of a network topology in the same experimental run. An integrated experiment leverages the advantages of using real and simulated elements to enable a) validation of experimental simulation models against real traffic b) exposing experimental real traffic to congestion-reactive cross traffic derived from a rich variety of existing, validated simulation models, c) scaling to larger topologies by multiplexing simulated elements on physical resources than would be possible with just real elements. A related form of network experimentation is to integrate experimental techniques *temporally*, as researchers experiment iteratively on the same input, providing comparison and validation. This is one of the key design goals of Emulab. The advantages of the latter are discussed elsewhere [60]. This thesis is confined to discussing issues in the former.

In order to simulate workloads that cannot be simulated in real-time in a single simulator instance, we use multiple simulator instances on a distributed set of PCs. Note that an integrated network experiment where all simulation resources of an experiment are modeled in a single simulator instance is different in the level of repeatability from an experiment where multiple simulator instances are used to model different portions of the overall topology. Although the former guarantees that no event is dispatched out of order, the latter has no such guarantees due to the fundamental nature of real-time distributed systems. Thus, integrated experiments as we have implemented it may have global "causality errors," naturally foregoing absolute repeatability offered by "pure" simulations.

This thesis discusses many of the issues in integrated network experimentation and presents work that solves some of them in the process of seamlessly integrating simulated resources via *nse*[1] in Emulab.[2] In this thesis, we characterize the limits on performance and accuracy for *nse*. We design, implement, and evaluate an adaptive algorithm to partition simulation models across a cluster to enable them to track real-time while minimizing resource use. We present the routing challenges raised in integrated network experimentation, discuss their relationship to other forms of simulation (e.g., virtual machines), design and implement a solution. We discuss the issues associated with event management in integrated experimentation. As a result of the work in this thesis, a user of Emulab is able to include simulated resources in integrated experiments in a consistent, highly automated manner without being concerned about their physical realization.

The following is a list of contributions of this thesis.

- Elucidate several tricky semantic and technical issues, in order to support multiplexed virtual resources such as simulated resources and Emulab's virtual-node support.

[1]*ns* emulation facility [18] permits simulated packets to leave the simulator for the "real" network, and vice versa. Refer to section 1.1.2 for an overview.

[2]Refer to section 1.1.1 for an overview.

- Add support for multiple routing tables in the OS, which solves one of the above issues. Develop and implement solutions to solve some of these issues to support simulated resources.
- Develop methods to automatically adapt the packing of simulated resources onto simulators running on physical PCs so that they are able to keep up with real-time. We refer to this as "auto-adaptation."
- Add primitives to *nse* to support multiple loosely coordinated instances of *nse* that each model different portions of the simulation topology in the same experiment. By loose coordination, we mean the delivery of user-specified simulation events that typically change traffic flows, such as starting or stopping flows, over a common time-base.
- Integrate simulated resources seamlessly into an experimental environment that allows experimenters to spatially combine real and simulated resources in the same experiment.
- Validate and analyze the performance integrated experiments.

We define here some of the terms we use in the rest of the thesis. A *pnode* is a physical PC node in Emulab. A virtual topology is one that an experimenter specifies and is independent of its physical realization in the experimental environment. Unfortunately, two definitions exist for the term vnode. A *vnode* is a node in the virtual topology which could be of different types such a PC vnode, simulated vnode or "virtual machine" vnode. The term vnode is also sometimes used by itself to mean a "virtual machine" vnode. In this thesis, however, we restrict ourselves to the first definition unless otherwise stated. Similarly, *vlinks* and *plinks* are virtual links in the virtual topology and physical links respectively.

1.1 Background

1.1.1 Emulab

Emulab is a general system for network experimentation, designed to provide consistent access to all three experimental techniques listed above. As such, its architecture separates the front-end presented to experimenters from the internal workings of Emulab, which is also separated from the back-end mechanisms that instantiate experiments. In this section, we give a brief overview of that architecture, shown in Figure 1.1. Emulab functions like an operating system for experiments in distributed systems and networking—it provides interfaces and tools to turn bare hardware into something much more effective to use. Emulab strives to preserve, however, "raw" access to the hardware for users who require it.

Similar to an operating system process, an "experiment" is Emulab's central operational entity—it encapsulates an experiment run by a user. It is directly generated from a front-end representation and then represented by a database. Emulab instantiates the experiment onto available hardware, performing the difficult tasks of choosing the appropriate resources and configuring nodes and links. A Emulab experiment may last from a dozen minutes to many weeks, giving researchers time to make multiple runs, change their software and parameters, or do long-term data gathering.

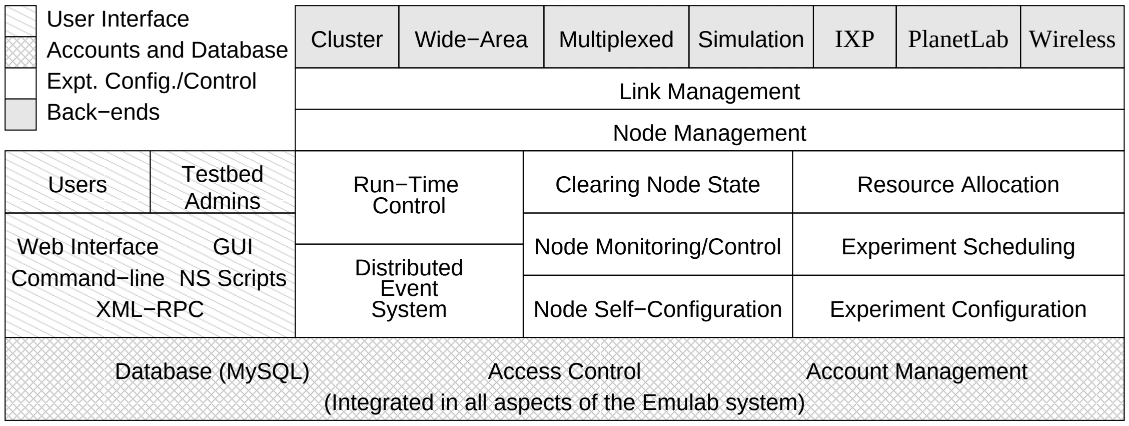


Figure 1.1. Emulab system architecture

To specify an experiment, users upload an *ns* script. Its use provides a powerful and familiar specification language; in addition, Emulab also features a “point-and-click” GUI that allows experimenters to draw their desired experiment topology, generating an appropriate *ns* script for them. The main primitives in an experiment specification are nodes and links. When specifying nodes, experimenters can request specific hardware, operating systems, and other parameters. Network links may include characteristics such as delay, bandwidth, and packet loss. Events can be also be scheduled to change link characteristics, bring down links or nodes, and start and stop traffic generators. The distributed event system is based on the Elvin [17] publish-subscribe system.

After being read by Emulab’s custom parser, the experiment specification is uploaded to a database. The experiment state stored in the database is further used in all parts of Emulab including experiment instantiation (also called “swap in”). Experiment instantiation begins with resource assignment. Finding hardware that matches an experimenter’s desired topology is an NP-hard problem. To find matches that make efficient use of resources, without overloading bottlenecks such as cross-switch links, Emulab implements a custom solver discussed in detail in [44]. Different resources are reserved and configured using different back-ends. Allocated physical resources such as PCs boot up and self-configure by contacting Emulab’s *masterhost* to download the necessary customizations. At the end of this process, an experiment is “swapped in” and ready for experimentation.

Emulab’s emulation back-end uses dedicated PCs and physical links to realize experiments. Although this conservative allocation introduces the least experimental artifacts, the size of any experiment is limited to the number of physical PCs and the node degree to the number of network interfaces on these PCs. This opens up an opportunity to use “soft” resources such as simulated or “virtual machine” type elements and multiplex them on physical PCs, thus scaling experiment size. For example, several simulated nodes and moderate speed simulated links can fit on a physical PC. Similarly several moderate speed virtual links can be multiplexed on a high-speed physical link.

1.1.2 *Nse*

The simulation back-end in Emulab that is described in the rest of the thesis takes advantage of the *ns* emulation facility (called *nse*) [18] permitting simulated packets to leave the simulator for the “real” network, and vice versa. *nse* supports two modes: *opaque emulation mode* and *protocol emulation mode*. Mode selection is merely based on *nse*’s runtime configuration. These modes are not mutually exclusive and can be combined in a single run. In *opaque emulation mode*, live packets from the network are treated as opaque packets that may be dropped, delayed, reordered or duplicated in the simulated network and then perhaps reinjected into the network. Such packets typically interact with packets natively generated in the simulator in router queues. *Protocol emulation mode* is where a simulator protocol implementation is communicating with its real-world counterpart. To assist this mode, software “transducers” convert packet formats from simulator to real-world and vice versa. In this thesis, we use only the opaque emulation mode. The reason for this is that *ns* is predominantly composed of abstracted protocol elements that are incapable of communicating with real protocol elements without large-scale modifications. However, *ns* in itself does not pose any restrictions in building a detailed protocol model that is fully compliant with the specification as much as a real world implementation. The basic operation of *nse* is illustrated in Figure 1.2.

Opaque emulation mode by itself is widely useful in the kind of real-simulated traffic interaction in simulated router queues that it enables. Note that even in opaque emulation mode, some protocol fields, such as packet length and TTL, are taken into account when it is introduced into the simulation.

Other than the ability for simulator packets to cross into the real network and vice versa, *nse* is indistinguishable from *ns*. Even though we only use *nse* to implement integrated experiments, the term *ns* is sometimes interchangeably used in this thesis.

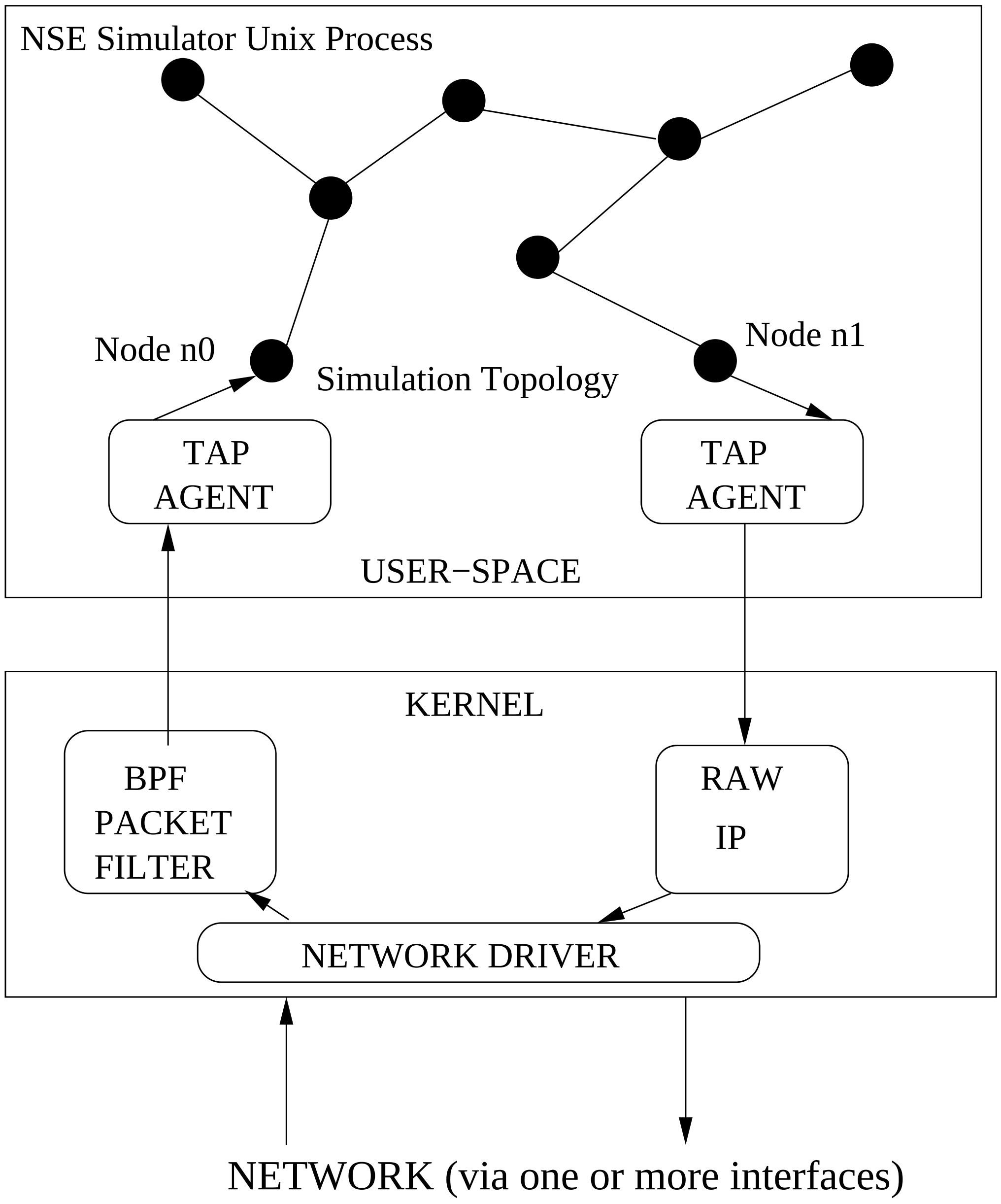


Figure 1.2. The basic operation of the *ns* emulation facility (*nse*)

CHAPTER 2

ISSUES

In this chapter, we elucidate several semantic and technical issues that need to be addressed in order to support multiplexed virtual resources such as simulated resources and "virtual machine" type resources in an integrated experiment. We only discuss simulation performance, model validity and fidelity issues whereas we implement solutions for the other issues.

2.1 Naming and Addressing Issues

Multiplexing virtual nodes onto physical nodes, and combining these with simulated and emulated nodes, raises some interesting challenges with respect to naming, addressing, and routing between nodes.

2.1.1 Node Naming

One of the strengths of Emulab, which we build on, is virtualized naming for all nodes. Within an experiment, all nodes share a common namespace. The hostnames by which they refer to themselves and other nodes are the virtual names assigned by the experimenter, rather than the physical name of the hardware they happen to be instantiated on. This is critical for transparency in two ways. First, it frees experimenters from having to change their experiment depending on which hardware it was instantiated on. Second, it allows portions of the experiment to be moved between different simulation and emulation mechanisms, without having to change the rest of the experiment.

2.1.2 Node Addressing

Subtle differences in addressing semantics between simulation and emulation presents several problems when our goals are to achieve both transparent integration and equivalence. The facility the testbed uses for allowing simulated traffic to interact with real traffic, *nse* [18], was designed as a simulator that is run on a single machine. When two or more instances of *nse* are distributed across physical machines to simulate resources in the same experiment, the node addresses used by the stock *nse* are not unique across these instances. Routing packets between these instances would thus not be possible without global addresses. IP addresses within Emulab are unique in an experiment Therefore, we use IP addresses for global routing of simulator packets. To support this, we have extended packet forwarding in *nse* to use IP addresses.

nse assigns a single address to each *node*. In real operating systems, however, IP addresses are assigned to each *network interface*. To support seamless integration of simulated nodes in integrated experiments, we have extended *nse* to add IP addresses to every link (i.e., interface) on a node. The source address of packets originating from that node is marked with one of its addresses.

Operating systems typically do not require that packets arrive on an interface with the destination IP address equal to the IP address of the interface. Thus, all the IP addresses of such a multihomed host refer to the same node. This represents the “weak end system model” [53, 9], in which network layer addresses refer to nodes rather than interfaces. Emulab’s support for static routing did not originally route to all IP addresses of a node. In Emulab we have extended the routing to use the same shortest path for all IP addresses of a multihomed host. This sometimes causes nonintuitive routes as well as unused links in the case of redundant paths. However, route paths are consistent regardless of which IP address of a node is used by an application, and thus helps preserve repeatability across experimental invocations. This has its downsides, for example if studying the effect of controllable multihoming on end to end performance. In such a case an experimenter could turn off static routing, instead using manual routing or Emulab’s dynamic routing.

2.2 Routing

Multiplexing virtual or simulated nodes and links on physical nodes and links poses an acute problem in the correct routing of packets. Packets should be routed according to the virtual topology and its routing dynamics over the physical topology. This appears straightforward, but there are several challenges. In the remainder of this section, we discuss these problems, referring to Figure 2.1 to illustrate the examples.

First, relying on a single routing table provided by the host physical node operating system, and shared by all virtual nodes, is insufficient. Multiple routing tables, one per virtual node, is an elegant solution to address the issue.

Second, incoming packets on a physical node require the context of the virtual link on which they arrived so that route lookups can be directed to the correct routing table, the table that corresponds to the virtual node on the endpoint of the virtual link. For example, a packet from source A0 to C0 should follow the path A0→B0→A1→B1→C0. This translates to physical hops A→B→A→B→C. On physical node B, the physical next hop will be either A or C depending on whether the packet arrived on virtual link vlink0-B0 or vlink2-B1. Therefore, if a packet arrived on vlink0-B0, the route lookup should be directed to B0's copy of the routing table and if it arrived on vlink2-B1, to B1's routing table. Touch et al. identify this issue and term it "revisitation" [53], although in their context, a packet leaves a physical node before returning on a different virtual link. In our system, multiple routing tables as well as the context of a virtual link are needed even when all the nodes and links of a virtual topology are hosted on one physical host.

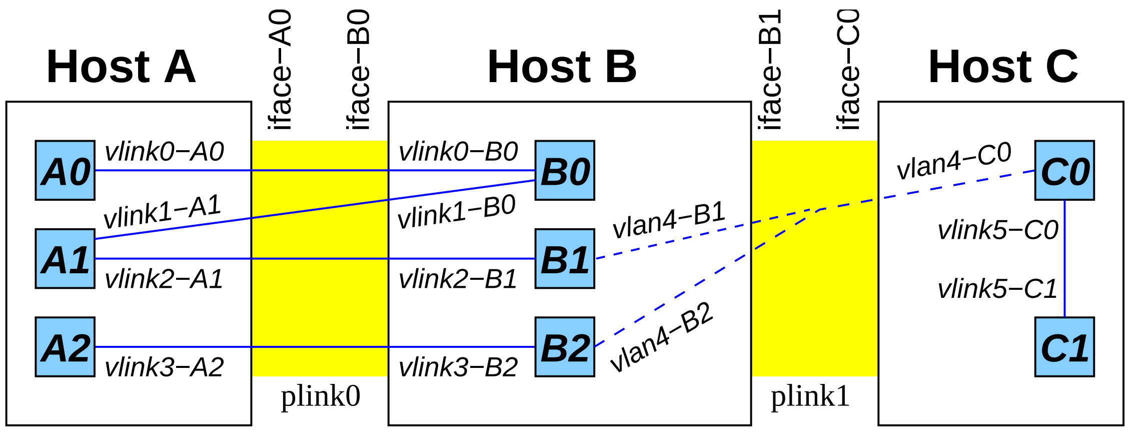


Figure 2.1. A network topology illustrating routing issues due to the multiplexing of simulated nodes and links. Large boxes represent physical nodes and links, while small boxes and lines (with *italicized labels*) represent simulated nodes and links. Virtual network interfaces (*vlinks*), virtual LANs (*vlans*), physical interfaces (*iface*), and physical links (plinks) have names as shown.

Third, depending on the mapping, packets will have source and destination virtual nodes on the same physical node. Under such circumstances, OS modifications are required to ensure that packets are not short circuited within the protocol stack instead of routed according to the virtual topology. For example, consider a packet from source A0 to A2. Since these two virtual nodes are on the same physical node, many OSes short circuit the packet.

Fourth, virtual nodes on a LAN that are mapped to different physical nodes and span multiple physical links/interfaces on one or more physical nodes are an issue. A LAN requires that the IP addresses of its members be in the same IP subnetwork. Attaching two IP addresses in the same subnetwork on two different network interfaces is disallowed by most OSes. Overcoming this problem requires considerable OS modifications. For example, virtual nodes B1, B2 and C0 are on a virtual LAN. B1 and B2, however, are mapped to the same physical node B. If vlan4-B1 and vlan4-B2 are the two virtual interfaces that are part of the virtual LAN, most OSes do not allow the attachment of IP addresses in the same subnet, which is necessary in this case, on these interfaces. Emulab solves all these issues.

Finally, as the topology is scaled using multiplexing, the global and per-vnode routing table size increases. A simple all pairs shortest path algorithm for N nodes results in $O(N^2)$ routes with N routes per virtual node. We discuss our solution to this scaling problem in this paper currently under submission [25].

2.3 Model Validity and Fidelity

Integrated experiments are typically constructed with models that represent the corresponding real world phenomenon for reasons of scale, efficiency and availability of the real-world phenomenon for experimentation. Models are constructed at different levels of abstraction or detail with respect to the real-world phenomenon they represent. For example, it may be easy to include a full implementation of a protocol in an experiment rather than using an abstract model. On the other hand, it is easier[1] to model network links or nodes than include those real components. In order to leverage the benefits

[1]Especially when automated experimental environments such as Emulab are unavailable

of integrated experimentation, such as scaling by replacing portions of a real topology with a simulated one, it is necessary to ensure the fidelity of the models with respect to the real-world components they replace. For example, replacing a real low-error-rate wired-network-link with a simulated one having a simple model of delay, bandwidth and statistical errors will continue to provide expected results. The same is difficult to ensure in wireless networks [23]. Even if a model is faithful to a real-world implementation, both the model and the implementation may not be valid with respect to the real-world phenomenon they represent. For example, a TCP dynamics experiment [60] on Emulab that compared different flavors of TCP between FreeBSD implementations and *ns* models uncovered a subtle bug in the FreeBSD implementation. This experiment was based on a *ns*-simulation comparison of different TCP flavors performed by Fall and Floyd [19].

2.3.1 Shared Implementation

Several simulators [10, 38, 58, 6] provide interfaces for shared implementation between pure simulation, emulation and live experimentation. Such mechanisms have many advantages: fidelity of the model across experimental techniques, increased realism and rapid progress from concept to production due to reduction in development time. However, it is important to have abstraction as a technique in validating networking ideas. This is because shared implementations require substantially more detail that makes it difficult to quickly prototype and validate new ideas. Fully detailed implementations also make it difficult to attribute observed experimental results to specific parts of the implementation and also increases the chances of introducing bugs.

ns does not export a POSIX or POSIX-like API for shared implementations. Therefore, it requires a large-scale software reengineering effort to port real implementations to *ns*.

Because of issues in abstraction and fidelity of models, it aids an experimenter performing integrated experimentation to know the differences between real resources and simulated resources. We discuss such differences below between key resources such as nodes, links, LANs and protocols in Emulab and *nse*.

2.3.2 Nodes

Emulab supports node types such as simulated nodes, PC nodes, widearea PCs, virtual PCs using a modified version of FreeBSD's jail [30], a primitive form of virtual machine. Simulated and jail nodes are lightweight virtual node abstractions that allow multiplexing on PCs and scale experiments beyond the limit of PCs. A single PC in Emulab could be used as a router, a delay node, an end-node or as a host for multiplexing simulated or jail nodes.

ns does not model node delays common in real PCs: packet copying overhead across PCI buses and memory and buffering in the hardware interface while the OS is busy performing other useful work. It is also not common to model unpredictable hardware interrupts in network simulation.

2.3.3 Links

Links in Emulab between PC nodes with different bandwidth and delay characteristics are realized by interposing a traffic-shaping node between the PC nodes. A real link that has the above characteristics behaves differently by causing packet drops at the network interface of the node on the link-endpoint rather than in the link. Such interface packet drops can be detected on the nodes and could cause applications to behave differently between emulated and real links. In practice, this difference has no effect on congestion reactive TCP or TCP-friendly traffic. Simulated links in *ns* are somewhat similar to the traffic-shaped links in Emulab in that interface drops are also not present here. A difference between emulated and simulated links is in observed value of delay versus the specified one. Emulated links are realized using hardware switched VLANs and involves copying overhead that is not present in simulated links. Switch effects increase inter node delay by a factor of 1.5 to 2 per-hop [61]. Because these are constant overheads, compensation could be easily added into simulation if required.

2.3.4 LANs

LANs in Emulab are essentially switched VLANs. Effects such as collision are very limited in this environment and can thus scale the number of nodes on a LAN well beyond traditional broadcast LANs. *ns* models a broadcast LAN. Because they are not equivalent, replacing one with the other will affect the results of experiments.

2.3.5 Protocols

ns has a large number of TCP model variants, many of which now have real implementations that behave and perform similar to their counterparts in simulation. However, the simulation TCP models, known as one-way TCP, are abstracted models. They do not perform initial connection setups and teardowns, do not handle sequence number wraparounds, use nonstandard data types for header fields, do not implement dynamic receiver advertised window, etc. [3]. The name "one-way TCP" is used because data transfer is allowed only in one direction and only ACKs are allowed in the reverse path for every connection. *ns* also has a TCP variant known as FullTcp which is intended to model a TCP in complete detail. This model allows two-way simultaneous data transfer and performs initial connection setup. Although this model is derived from a Reno BSD implementation, the port to *ns* was not entirely faithful. It uses a 31-bit signed integer to represent TCP sequence numbers similar to other *ns* one-way TCP variants. It does not handle sequence number wraparounds and does not implement dynamic receiver advertised window. Implementation issues such as this make it impossible to have a mixture of real and simulated protocol endpoints without fixing the simulated implementation.

ns uses more accurate timers for TCP than does a real OS implementation. In *ns* every TCP endpoint schedules separate timer events each with a fine granularity. A real implementation such as TCP in BSD uses very few timers per host and thus timeouts have a coarser granularity. For example, BSD TCP uses a slow timer of 0.5 seconds that sweeps through all active TCP connections to declare time-outs for ones that have expired in the last 0.5 seconds [39]. Differences such as the ones listed above can alter initial or steady state throughput by a factor of 2-10 [24]

2.4 Scalable Resource Allocation and Mapping

Network experimentation on real hardware requires a mapping from the virtual resources an experimenter requests to available physical resources. This problem arises in a wide range of experimental environments, from network emulation to distributed simulation. This mapping, however, is difficult, as it must take a number of varying virtual resource constraints into account to "fit" into physical resources that have bandwidth bottlenecks and finite physical node memory and compute power. Poor mapping can reduce efficiency and worse, introduce inaccuracies—such as when simulation events cannot keep up with real-time—into an experiment. We call this problem the "network testbed mapping problem" [45]. In general graph theory terms, this is equivalent to the graph embedding or mapping problem with additional constraints specific to this domain. This problem is NP-hard [45].

The mapping could be many-to-one, such as multiple vnodes and vlinks on a physical node, one-to-one, such as a router node on a physical PC, or one-to-many, such as a vlink of 400Mbps that uses four physical 100Mbps links[2] [45].

When simulated traffic interacts with real traffic, it must keep up with real time. For large simulations, this makes it necessary to distribute the simulation across many nodes. In order to do this effectively, the mapping must avoid "overloading" any pnode in the system, and must minimize the links in the simulated topology that cross real plinks. By "overload," we mean that there are are more simulated resources mapped to a pnode than the instance of a simulator can simulate in real-time.

"Pure" distributed simulation also requires similar mapping. In this case, rather than keeping up with real time, the primary goal is to speed up long-running simulations by distributing the computation across multiple machines [8]. However, communication between the machines can become a bottleneck, so a "good" mapping of simulated nodes onto pnodes is important to overall performance. Although this is primarily achieved by minimizing the number of vlinks that cross pnodes, another factor that affects perfor-

[2] Although the latter is not supported currently in Emulab by mapper known as `assign`. `assign` was designed and implemented by other Emulab contributors. Our primary role is that of a user of this mapper to implement integrated experimentation

mance is the lookahead that can be achieved. Lookahead refers to the ability to determine the amount of simulated time that could be safely processed in one simulator process without causality errors due to events from a different simulation process. Lookahead is affected by the *distance* between simulation processes [21]. Distance provides a lower bound in the amount of simulated time that must elapse for an unprocessed event on one process to propagate (and possibly affect) another process. Therefore, it is not just important that a good mapping has fewer links crossing simulation processes, but also for them to be lower bandwidth links because they increase distance and thus lookahead, leading to improvement of efficiency of a distributed simulation.

A good mapping has the following properties:

- Sparse cuts: The number of vlinks whose incident vnodes are mapped to different pnodes should be minimized. At the same time, the number of vnodes and vlinks mapped to the same pnode should not exceed its emulation capacity.

- Low congestion: The number of vlinks that share plinks should be minimized without over-subscribing the plinks. Although some plinks such as node-to-switch plinks are dedicated to an experiment, others such as interswitch plinks are shared between experiments. By minimizing vlinks mapped to shared plinks, space-sharing efficiency is increased.

- Low dilation: The physical length (i.e., hops) that correspond to mapped vlinks, also known as dilation, should be kept to a minimum. For example, a vlink that is mapped to a plink that traverses multiple cascaded switches, is less desirable than one that traverses only one switch.

- Efficient use of resources across experiments: The unused capacity of physical resources that are not shared across experiments must be kept to a minium. In other words, minimize usage of shared resources such as interswitch links and maximize usage of experiment private resources such as pnodes and switch links from/to these nodes.

- Fast Runtimes: A suboptimal solution arrived at quickly is much more valuable than a near optimal solution that has very long runtimes (e.g., minutes vs. hours). This aspect becomes important when we map iteratively using runtime information to perform auto-adaptation of simulated resources. Due to the NP-hard nature of the problem, the runtimes are easily exacerbated by larger topologies made possible by "soft" resources such as simulated or "virtual machine" resources.

`assign` supports a node type system. Each node in the virtual topology is given a type by the experimenter, and each node in the physical topology has a set of types that it is able to satisfy. Each type on a pnode is associated with a "packing factor" (also known as "co-locate factor"), indicating how many nodes of that type it can accommodate. This enables multiple vnodes to share a pnode, as required by integrated experimentation as well as "pure" distributed simulation. For example, if `sim` is the type associated with simulated nodes, a pnode will support a co-locate factor for nodes of type `sim`. However, if all virtual or simulated nodes are considered to be equal, this can lead to suboptimal mapping since typically the pnode resources consumed by vnodes are all different. To achieve better mapping, arbitrary resource descriptions for vnodes and pnodes need to be supported. However, this adds a bin-packing problem to an already complicated solution space. In order to flexibly support soft resources such as simulated or "virtual machine" resources, several new features were added recently to `assign` [25]. We describe these features below[3]:

- Limited intranode bandwidth.
- Resource descriptions.
- Dynamic physical equivalence classes.
- Choice of pnode while mapping.
- Coarsening the virtual graph.

[3]We discuss how they are used to iteratively map simulated resources in section 3.4

2.4.1 Limited Intranode Bandwidth

When multiple vnodes are mapped to a pnode, vlinks are also mapped to the same pnode. Originally, there was no cost for such vlinks which makes it possible to potentially map an arbitrarily large number of vlinks. In reality however, there is a limit on the number and total capacity of vlinks that can be supported. An idle vlink has no cost other than memory used up in the simulator. However, there is a computational cost of processing packets when traffic passes through vlinks. `assign` now supports an upper limit on the intranode bandwidth and uses it when mapping vlinks whose capacities are allowed to add up to the bandwidth. When mapping simulated resources, we set this capacity to 100Mbps on Emulab hardware, based on measurements reported in section 4.1.

2.4.2 Resource Descriptions

Pnodes support arbitrary resource capacity descriptions such as CPU speed, memory, measured network emulation capacity, and real-time simulation event rate. Essentially this could be any value that represents an arbitrary resource capacity. Capacities for multiple resources are possible per pnode. Thus, vnodes with resource usage values for multiple resources are counted against the above capacities. It is possible to use resource descriptions even if only relative resource usage between vnodes is known. For example, if vnode A consumes thrice as many resources as vnode B, vnode A when mapped to an empty pnode would become 75% full.

2.4.3 Dynamic Physical Equivalence Classes

`assign` reduces its search space by finding groups of homogenous pnodes and combining them into physical equivalence classes. When multiplexing vnodes on pnodes, a pnode that is partially filled is not equal to an empty node. This is not just in pnode capacity but also in its physical connectivity to other pnodes since the plinks between them are also partially filled. `assign` now computes the physical equivalence classes dynamically while mapping. Although this helps a little, this feature is close to not having physical equivalence classes at all. This factor is the dominant contributor to longer runtimes when mapping multiple vnodes on pnodes, compared to an equal-sized topology with one-to-one mapping. For large topologies, the runtimes can be very long, into the tens of minutes and even hours.

2.4.4 Choice of Pnode While Mapping

As we noted before, a good mapping is likely to map two vnodes that are adjacent in the virtual topology, to the same pnode. Instead of selecting a random pnode to map a vnode, `assign`, now, with some probability, selects a pnode to which one of the vnode's neighbors has already been assigned. This dramatically improves the quality of solutions, although not the runtimes on large topologies.

2.4.5 Coarsening the Virtual Graph

Using a multilevel graph partitioner, METIS[31], which runs much faster than `assign` primarily because it has no knowledge of the intricacies of the problem, the virtual topology is "coarsened." By "coarsening," we mean that sets of vnodes are combined to form a "conglomerate" to form a new virtual graph which is then fed to `assign`. This feature dramatically improves runtimes, again due to the reduction in search space, making it practical to perform auto-adaptation.

2.5 Automation of Integrated Experiments

Virtualization and mapping are not nearly as useful without automation of the complete experiment life cycle. For example, comparisons between manual experimental setup against an automated one of a six node "dumbbell" topology in Emulab show an improvement of a factor of 70 in the automated case [60]. One of the aspects of automating the mapping of the simulated portion of an integrated experiment is that the specification of the simulated topology must be used to generate multiple specifications for each subportion of the topology that gets mapped to different physical nodes. The structure of the specification language and the relationship between virtualized resources can have an impact on the ease of doing this. The use of OTcl [59], a general purpose programming language with loops and conditionals, for specification in *ns* and Emulab makes the above task difficult compared to a domain specific language that enforces relationships between resources at the syntactic level. For example, the domain specific language (DML) used by the scalable simulation framework (SSF [15]) simulator has a hierarchical attribute tree notation [14] that would make it easier to generate specifications of subportions of the full topology. DML is a static, tree-structured notation similar to XML. Because of the tree-structured nature of DML, a node and its attributes as well as any resources belonging to the node are syntactically nested. Thus, a simple parser can partition such a node and everything associated with it easily. On the other hand, *ns* OTcl can have simulated resources with logically nested relationships scattered anywhere in the code without explicit syntactic hints, making such partitioning more complex.

Emulab uses an OTcl interpreter to parse a user's OTcl [59] specification into an intermediate representation and stores this in a database [60]. The parser statically evaluates the OTcl script and therefore takes loops and conditionals into account. Using a similar approach, we have developed a custom parser to parse the simulated portion of the integrated experiment specification and generate new OTcl specifications for each subportion of the topology mapped to a physical node. The design of such a custom parser is discussed in the section 3.2.3. The implementation described in this section can be retargeted to generating OTcl subspecifications to map pure distributed simulation using *pdns*, albeit with modest changes to our implementation. It is a separate project that will be done outside of this thesis.

2.6 Performance

In the case of integrated experimentation, the performance of a simulation is even more critical than "pure" simulations as it could result in inaccurate results if performed with a model size that exceeds the capacity of a simulator instance[4] as opposed to just taking longer to complete in pure simulations. Mapping is related to performance because the quality of mapping is dependent on the ability to characterize the performance of simulation as accurately as possible. An overly optimistic mapping of simulated

[4]In our system, a violation is detected under an overload and the experiment is retried with a different mapping or aborted after several retries

resources could cause one or more simulated partitions to not track real-time, thus producing inaccurate results. On the other hand, an overly pessimistic mapping does not fully exploit physical resources. An additional overhead over pure simulation is the need to capture and inject live network packets. Link emulators such as dummynet [48] are therefore implemented in the OS kernel to avoid the packet-copying and context-switch overheads across user-kernel boundaries.

We first present an overview of the performance aspects of pure simulation. The execution time of a discrete event simulation is proportional to the number of events that need to be processed in a unit of simulation, known as the event rate. Even though the computation performed by different events is different, averaging over large runtimes of a complex simulation provides an adequate measure of the overall computational effort of a simulator [35]. The event rate is proportional to the size of the model and the rate of traffic flow. In real-time simulations, it is also useful to measure the number of events processed in a unit of runtime. In this thesis, we present results with events per runtime as we mostly experiment with simulations that are short. Several techniques are used to speed up simulations: increasing the computational power available for simulations via faster CPUs and parallel simulations on multiple processors [21, 46, 40], improved event list algorithms and increasing the level of abstraction [4, 26, 35, 22]. The latter technique reduces the event rate of a comparable model by changing the granularity of simulations. For example, fluid models abstract streams of packets as fluid flows in which events correspond to fluid rate changes at traffic sources and network queues. Most of these techniques trade off accuracy for increased speed. In some cases such as large scale networks, packet-level simulation is likely to outperform fluid simulation due to “ripple effects” [36].

Improved event list algorithms such as calendar queues [12] have a theoretical asymptotic running time of O(1) for all operations. The splay tree [51] is an O(log n) algorithm [29] that is preferred when the number of events in the queue is not large. In practice, it is easy for the running time of the calendar queue to degrade to linear time [40]. In our experience, both the calendar queue and splay tree implementations in *nse* perform equally well with the link emulation workloads we have tested.

A study that compares the performance of a few packet-level simulators, namely *ns*, *JavaSim*, *SSFNet in Java* and *SSFNet in C++*, under a specific type of workload with a "dumbbell" topology, found *ns* to be the fastest simulator [39]. Thus, *ns* seems to be a better choice for integrated experimentation than other packet-level simulators, in both performance as well as the variety of existing, validated protocol modules.

2.7 Real-time Simulation Issues

We present two real-time simulation issues that can arise out of the choice of implementation. The basic pseudo code for the real-time event scheduler in *nse* is presented in Figure 2.2. The simulator clock is frequently syncrhonized to real-time.

2.7.1 Physical Clock Rate Stability and Skew

The simulator uses some physical clock to track real-time. The physical clock rate stability is more important for our purposes than clock offset from the true time. Synchronization to real-time was originally performed in *nse* by making a Unix `gettimeofday()` system call and subtracting the time from the initial value of the same at the beginning of

```
bool halted = false;
double sim_clock;
while( !halted ) {

    // get event with earliest timestamp;
    next_event = get_next_event();

    // synchronize simulator-clock to real-time;
    // Get real time relative to start of simulation
    sim_clock = get_real_time_clock();

    while( next_event->timestamp <= sim_clock ) {
        dispatch(next_event);
    }

    // Get real time relative to start of simulation
    sim_clock = get_real_time_clock();

    // Check and introduce live packets (e.g. from network)
    // or events (e.g. from event system) from external
    // sources if any.
    if ( poll_external_event() == true ) {
        insert_external_events();
    }
}
```

Figure 2.2. Pseudo code for the *nse* real-time event scheduler causing accumalated errors the simulation. Since the above synchronization is performed frequently, the relatively high cost of a system call– as compared to simple function calls– increases the overhead of the scheduler. Most of the newer *x86* processors support an accurate CPU cycle counter or in Pentium processor terminology, known as the time-stamp-counter (TSC) directly readable in "user-mode." Most operating systems perform a calibration of the rate of this counter in order to determine CPU speed during boot time and export this value to user applications. Reading the TSC hardware counter is cheap and using the rate above, we get an accurate estimate of the real-time elapsed. We have modified *nse* to use this counter when it is available. Although the oscillator stability of the CPU clock is very high, the method of determining the CPU clock rate at boot time by interpolating the standard clock from the 8254 chip over a small measurement interval of around 50ms introduces the skew of the 8254 clock. The 8254 clock has a skew of around one second every 5.55 hours [43]. However, our method above is no less accurate than using the `gettimeofday()` system call since that uses the standard clock from the 8254 chip. One downside of this method is that it cannot be used on multiprocessors. Each processor supports a different TSC, usually with different rates. When a user process is scheduled on two different processors in different scheduling intervals, it reads the value of two different TSCs that are not correlated with one another.

2.7.2 Separation of Simulator Virtual Clock and Real-time Physical Clock

In the algorithm described in Figure 2.2, all events with timestamps earlier and up to the current simulator clock are dispatched. Note that the simulator clock is frequently updated with real-time relative to start of the experiment. Depending on how long it takes to check for live-packets or dispatch an earlier event, nearly all future events are dispatched late with respect to real-time, by some δ. The event being dispatched could introduce future events in the scheduler queue, all relative to the current simulator clock.

The above algorithm seems simple enough. However, we show that it reduces the accuracy of the simulation due to accumulated errors with the following example illustrated in Figure 2.3 (a). Consider an event, e, dispatched in the scheduler that results in a packet being scheduled for transmission over a link which has both transmission delay and propagation delay components being modeled. This causes an event e_1 to be scheduled after the link transmission time from the current simulator clock. Similarly, another event e_2 is scheduled after the sum of the link transmission time and propagation delay.

Figure 2.3 (a) shows the scenario just described where e_1 and e_2 are scheduled relative to the simulator clock synchronized to real-time. In pure simulation running in virtual time, these events would be scheduled relative to e as shown in Figure 2.3 (b). Thus, the algorithm described above causes an error of δ for a single packet. Now consider a packet that traverses many such links. Each time an event that corresponds to this packet is dispatched late by some δ, the simulation error for this packet increases by that δ. Eventually, after crossing many links, simulation errors for a packet will accumulate and become noticeable. When this happens for every packet along this path, aggregate statistics for the flow will be noticeably incorrect. Thus, it is important to avoid this accumulation error in order to keep the simulation accurate. The error should be within only a small constant.

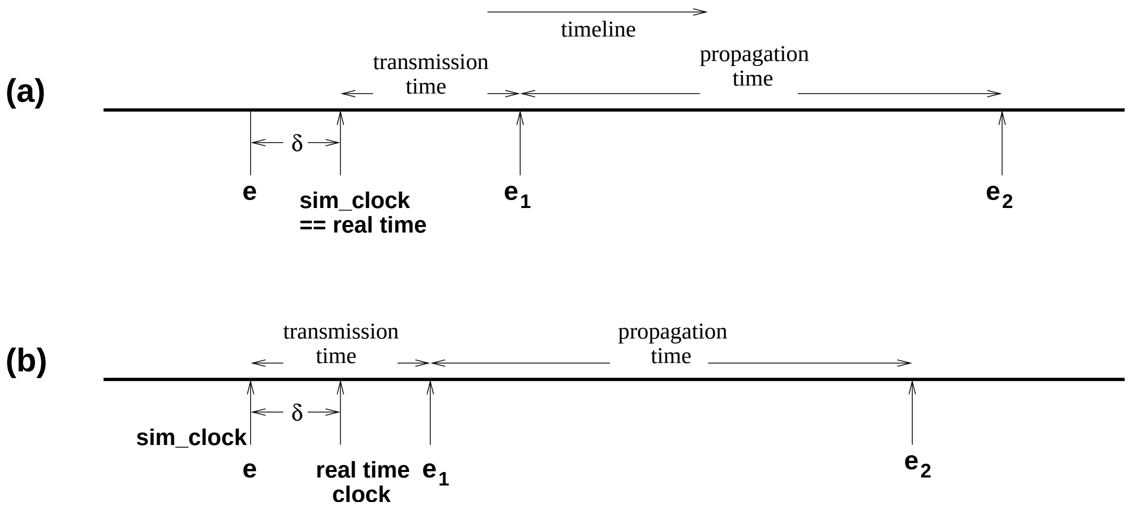


Figure 2.3. Time-line for real-time event dispatch that causes accumulated errors

This issue is addressed by keeping the simulator virtual clock and real-time physical clock separate. The simulator virtual clock is initialized to the timestamp of the event being dispatched. Only when live packets are introduced into the simulation is the simulator virtual clock initialized to the real time physical clock. This is required in order to ensure that causal future events due to live packets are inserted into the scheduler relative to the real-time physical clock. An event dispatch that avoids accumulated errors is illustrated in Figure 2.3 (b). We have addressed this issue in *nse* with straightforward changes whose pseudo code is given in Figure 2.4.

```
bool halted = false;
double sim_clock;
double real_time_clock;
while( !halted ) {

    // get event with earliest timestamp;
    next_event = get_next_event();
    // synchronize simulator-clock to real-time;

    // Get real time relative to start of simulation
    real_time_clock = get_real_time_clock();

    while( next_event->timestamp <= real_time_clock ) {
        sim_clock = next_event->timestamp;
        dispatch(next_event);
    }

    // Get real time relative to start of simulation
    real_time_clock = get_real_time_clock();

    // sim_clock is now synchronized to real time so
    // that external events introduced into the scheduler
    // queue have future time relative to real time as
    // closely as possible
    sim_clock = real_time_clock;

    // Check and introduce live packets (e.g. from network)
    // or events (e.g. from event system) from external
    // sources if any.
    if ( poll_external_event() == true ) {
        insert_external_events();
    }
}
```

Figure 2.4. Pseudo code for the *nse* real-time event scheduler without accumulated errors

CHAPTER 3

DESIGN AND IMPLEMENTATION

In this chapter, we first discuss the design changes we have made to the stock *nse* in order to support integrated experimentation. In particular, the changes are primarily related to supporting multiple *nse* instances running on a distributed set of PCs in a coordinated manner simulating different portions of the experimental topology. We then discuss how we integrated the simulation back-end with the rest of the experimental environment.

3.1 Distributed *Nse*

In order to scale to larger simulation topologies that cannot be supported in real-time by a single instance of *nse* on Emulab hardware, we map portions of the simulation topology to different instances of *nse* on multiple PCs in a single experiment. Each instance of *nse* processes simulator events independently of one another and in real-time. However, user-specified events must be distributed to the simulator instance where the associated simulator object is mapped. This is discussed in section 3.1.1. When portions of the topology are mapped to different simulator instances, several links are cut. In section 3.1.2, we discuss how we replace such links with special links that we have designed that cross simulator instances. In sections 3.1.3 and 3.1.4 we discuss how we transport simulator packets and globally route them across multiple simulator instances.

3.1.1 Distribution of User-specified Events

When a user explicitly specifies future events, such as bringing links up or down, it is easily implemented in pure simulation by adding such an event to the global event list in the correct position corresponding to the time when it needs to be dispatched. We face the following challenges when implementing the above with multiple simulator instances responsible for different simulated resources:

3.1.1.1 Common Notion of Time

Since multiple simulator instances are modeling different portions of the same experimental topology, they must share a common notion of time. The rate of the flow of time is roughly taken care of by real-time-simulation modulo clock-skews that may be different for different PCs. This aspect is discussed in section 2.7. The offset of these clocks from the global common time at any given time should be nearly zero. This is needed in order to dispatch two user-specified same-time-events at the same global time if those events were present in the event lists of two simulator instances. There are at least two ways of implementing this, although achieving perfect synchronization for same-time events across distributed simulator instances executing in real-time is impossible:

- Use distributed *barrier synchronization* to synchronize the start time of each simulator instance and insert user-specified events in the individual event lists. Clock skews across the simulator instances could result in out of order event dispatch for same-time or nearly same-time events across these simulator instances.

- Use a centralized event scheduler that maintains the notion of a per-experiment time and dispatches user-specified events to the distributed instances at their dispatch time. There is a latency associated with every event delivered as well as a skew between different same-time events. We use this method in integrated experimentation, the details of which are discussed later in this section.

3.1.1.2 Map Event to Simulator Instance

An event must be mapped to the correct simulator instance in which it is to be dispatched. Unlike pure simulation with *ns*, it is difficult to support execution of arbitrary OTcl code in user-specified events. Without a binding between the event and a particular simulated resource, such as an object that has been mapped to a particular simulator instance, there is not enough information on where to direct an event. In Emulab, an event is associated with objects such as nodes, links and traffic agents. After the mapping (aka resource allocation) phase, we can establish the mapping between an event and the physical host to which the event needs to be delivered. The list of events as well as mapping of simulation objects to physical hosts are stored in Emulab's database. For events that affect simulation objects, actual OTcl strings that affect the object are stored. As part of the experiment swapin process, a per-experiment event scheduler process is run on Emulab's *masterhost*. It reads the events from the database, queues them up and delivers them to the physical hosts over the control network. Besides processing static events that the experimenter specified in the *ns* file, the event scheduler also supports dynamic events. An experimenter can affect simulation objects dynamically by running an Emulab provided event generation tool with the correct parameters, which includes the name of the simulation object, time relative to when the above program runs, the type of event and event arguments.

We have integrated *nse* with Emulab's *event system*. Event delivery in Emulab is supported via the Elvin [17] publish-subscribe system that supports a flexible, content-addressable messaging model. We have modified the *nse* real-time scheduler to check for events sent from the Emulab central event scheduler similar to checks for live packets. The events themselves contain OTcl strings that are evaluated in *nse*, which typically act on simulation objects, such as starting and stopping traffic agents.

The latency of the delivery of events over Emulab control network was measured and reported in a technical report [61]. The latency for a single event is expected to be on the order of a packet round-trip between the *masterhost* and the PC running *nse* over the control network. When same-time events are dispatched to multiple PCs, an additional skew exists between events dispatched to the first and last nodes. From [61], out of 10 PCs receiving same-time events, the smallest latency was around 200μs and the largest value was around 2ms.

We choose Emulab's event system over barrier synchronization for delivering user specified events to be uniform with event delivery mechanisms across simulation and emulation objects. The centralized event scheduler at least has the guarantee of dispatching events in the order it was specified even though they could come into effect in different instances of *nse* out of order. If the order of event dispatch needs to be strictly preserved in integrated experimentation, it is best achieved by keeping events at least 10ms or more apart, although there are no guarantees. Notice that with this method, there is no need to keep the start times of different *nse* instances synchronized. In other words, the absolute value of the simulator-clocks in different *nse* instances are unimportant. In fact, they could be seconds to tens-of-seconds apart depending on how different PCs running the simulator instances boot up. The centralized event scheduler, however, starts processing user specified events only after all PCs in an experiment are ready.

Although simulation offers the ability to be perfectly synchronized across nodes, thus having absolute repeatability, it is not possible or necessarily desirable to achieve perfect synchrony in integrated experimentation. In fact, modeling asynchrony in simulation that is common in distributed systems is important in getting realistic results [24]. Thus, using integrated experimentation, it is possible to explore the effects of clock drifts on a simulated protocol under study. By mapping simulated nodes one-to-one on PCs, complete asynchrony as seen in real-systems is achieved. At the other end of the spectrum, if all simulated nodes were mapped to a single PC, the effects of synchrony could be observed. Hence, our system enables qualitatively new validation techniques.

3.1.2 Simulated Links Between Nodes in Different *Nse* Instances

The Emulab mapper considers nodes as the basic unit of resource assignment. When two simulated nodes with a duplex link between them are mapped to different instances of *nse* running on two PCs, we replace such a duplex-link with a new kind of link object we have developed, known as an `rlink` (short for remote link). An `rlink` is a simplex link with one endpoint being a node and the other endpoint being an object that encapsulates a simulator packet in a live IP packet and sends it over the physical network. The details of the encapsulation and decapsulation are discussed in section 3.1.3. The idea of an `rlink` is similar to the one developed for parallel and distributed *ns* (*pdns*) [46] in that both represent a link whose one endpoint is in a different simulator instance. However, they are different both conceptually as well as in implementation. In *pdns*, events are passed around between simulator instances whereas integrated experiments encapsulate packets and transport them over the physical network. A duplex link between two simulated nodes mapped to two *nse* instances is implemented using two `rlinks`. Each `rlink` is instantiated in the *nse* instance where the source endpoint node is present instead of mapping both in the same *nse* instance. This helps to load balance the link simulation.

Figure 3.1 has a code snippet that demonstrates how our system uses an `rlink` to create a simplex link from node `n0` to a node in another *nse* instance with the IP address `<dst_ip>`. Notice that we also set an IP address `<src_ip>` on the `rlink` using the `set-ip` method on the node object. Every link object now supports a `set-ip` method to set IP addresses on every link (i.e., interface), which was discussed in section 2.1.2. The result of a `set-ip` method is to add the IP address to the list of IP addresses of a node. A simulated node now has as many IP addresses as there are links originating from it. A simulator packet with the destination address equal to any of these IP addresses will be forwarded and delivered to traffic agents attached to this node (i.e., if the destination port matches with one of the agents). An `rlink` can also be used without bandwidth, delay or queue type to specify a connection to a real PC node. If link shaping is required on the link between a simulated node and a PC node, it is performed by an interposed delay PC running Dummynet in Emulab.

For every `rlink` that is created, a TAP agent is internally instantiated. A packet that traverses an `rlink` undergoes link simulation and is eventually handed over to its TAP agent. If a simulator packet is received, encapsulation is performed and injected

```
set ns [new Simulator]
set n0 [$ns node]

# All IP addresses are in dotted quad

# rlink from n0 to another node in a different nse
set rl0 [$ns rlink $n0 <dst_ip> <bandwidth> <delay> <queue_type>]
$rl0 set-ip <src_ip>

# rlink from n0 to a real PC node
set rl1 [$ns rlink $n0 <dst_ip_for_pc>]
```

Figure 3.1. *nse* code in one instance to create a link with one endpoint in a different *nse* instance illustrated in Figure 3.2

into the live network. A live packet previously captured from the network is injected "as is." An encapsulated packet is decapsulated just before being delivered to a traffic agent. Figure 3.2 shows the dataflow between different simulator objects. Although a TAP agent is internally allocated for every `rlink`, complete configuration of the TAP agent is deferred untill after the setup of the PC host that runs *nse*. In particular, information on the network interface(s) from which live packets are to be captured is not available untill PC host boots up.

3.1.3 Encapsulation and Decapsulation of Simulator Packets

In order to support traffic agents in different instances of *nse* to communicate with one another, simulator packets must be encapsulated in an IP payload and transferred over the network. *nse* protocol state for different protocols is present in the form of several header fields organized in contiguous memory also known as a "bag of bits." Therefore, a simple copy is sufficient to perform the encapsulation. Every packet has space for header fields of nearly all the protocols supported by *ns* even if only a small portion of the header space is used at a time. The unused portions are zeroed. We encode such headers into a compact summary so that they occupy less space in the IP payload and thus reduce the likelihood of IP fragmentation. The summary is made up of repeated patterns of `<offset>`, `<length>` and nonzero words. The `<offset>` values are the locations relative to the start of an unencapsulated buffer from where nonzero data exists.

An encapsulated packet may have to traverse multiple simulator instances before it has to be delivered to a traffic agent. Although most of the protocol state in the simulated packet is used by the traffic agents on the ends, nearly justifying performing the decapsulation just before it is delivered to a traffic agent, some state such as the simulated packet size are needed to perform correct link simulation. Similarly, other state such as TTL or *Explicit Congestion Notification (ECN)* bits are modified in the intermediate router nodes. Therefore, we perform decapsulation as soon as an encapsulated packet is captured from the live network even if the final destination node is in a different simulator instance. If such a packet again leaves the simulator instance, it is encapsulated again before being injected over the live network.

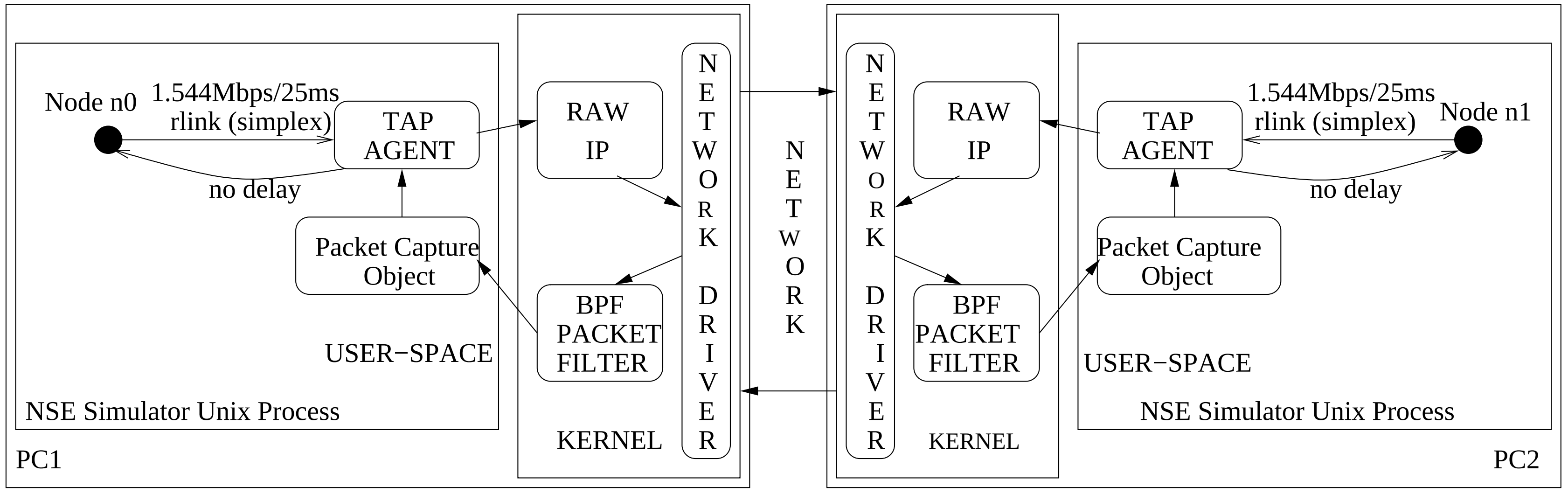


Figure 3.2. Implementation of a simulated duplex-link between nodes mapped to two *nse* instances

3.1.4 Global Routing of Packets

As discussed in section 2.1.2, we have extended *nse* packet forwarding to use IP addresses for global routing. Node addresses in *nse* prior to our changes used a 32–signed address. Changing this to a 32–bit unsigned address to fit in IPv4 addresses was trivial. We also modified the bitmasks used for multicast addresses to reflect IPv4 multicast addresses.

We have extended *nse* to be able to add IP address based routes with the following syntax described in Figure 3.3.

3.2 Integration with Experimental Environment

Note that the experimenter in Emulab need not worry about routes or IP addresses as shown in section 3.2.1. It is merely an implementation detail of our system. In setting up integrated experiments, we already compute the all pairs shortest paths for the experimental topology. We manually add only IP address based routes in all *nse* instances. Currently, we support only two bitmasks 255.255.255.255 and 255.255.255.0. The address classifier performs a route-lookup using all 32 bits of the destination address of the packet. If that fails to find a next hop, it tries another lookup with the lower eight bits masked. This is a limitation of our current implementation. However, extending *nse* to support a modern longest-prefix-match based classifier is straightforward. We believe that this feature is not necessary to demonstrate the ideas we present in this thesis.

Integrating simulation into Emulab involves changes to many parts of the software, the details of which are all not discussed in this thesis. One key point worth noting is that the integration of the simulation back-end fits nicely with the abstractions already present to support cluster PC nodes and multiplexed virtual node backends. Although the number of lines of code added to support the simulation back-end are not very large,

```
set n0 [$ns node]
# IP addresses and netmasks in dotted quad
$n0 add-route-to-ip <dst_ip> <nexthop_ip> <netmask>
```

Figure 3.3. *nse* code to add IP address based routes thus supporting our claim above, the changes themselves were not straightforward due to the immense task of understanding the software of a large distributed system with many complex interactions. We first discuss how an experimenter specifies an integrated experiment and use the reminder of the section to discuss independent portions of the simulation back-end.

3.2.1 Use

To create an experiment in Emulab with simulated resources in it, a user simply has to enclose a block of NS OTcl code in `$ns make-simulated` { }. Connections between simulated and physical nodes are specified as usual using a `duplex-link`. Multiple `make-simulated` blocks are allowed in a single experiment which results in the concatenation of all such blocks. Figure 3.4 illustrates an example of an experiment with a "dumbbell" topology comprised of both real PC nodes and simulated nodes. The OTcl source-code is presented in Figure 3.5.

The "dumbbell" topology of six nodes is mapped to four PCs in Emulab. Note that this is a very low multiplexing factor explicitly specified in the code to keep the example simple. Two simulation host PCs are automatically allocated by the system. The code in the `make-simulated` block will be automatically reparsed into two OTcl subspecifications, of which each is fed into an instance of *nse* running on the simulation host. Depending on how the mapping happens, there can either be one or two simulated links that cross PCs. In Figure 3.4, we have one such link that cross PCs.

3.2.2 Experiment Configuration

We list the (details abstracted) steps performed from specification to actual experiment instantiation below:

1. Perform initial parsing of user-specified OTcl code. Store experiment information in the database.

2. Compute all pairs shortest path static routes for the topology and store the routes in the database. This includes routing to every interface of a node, each having an IP address.

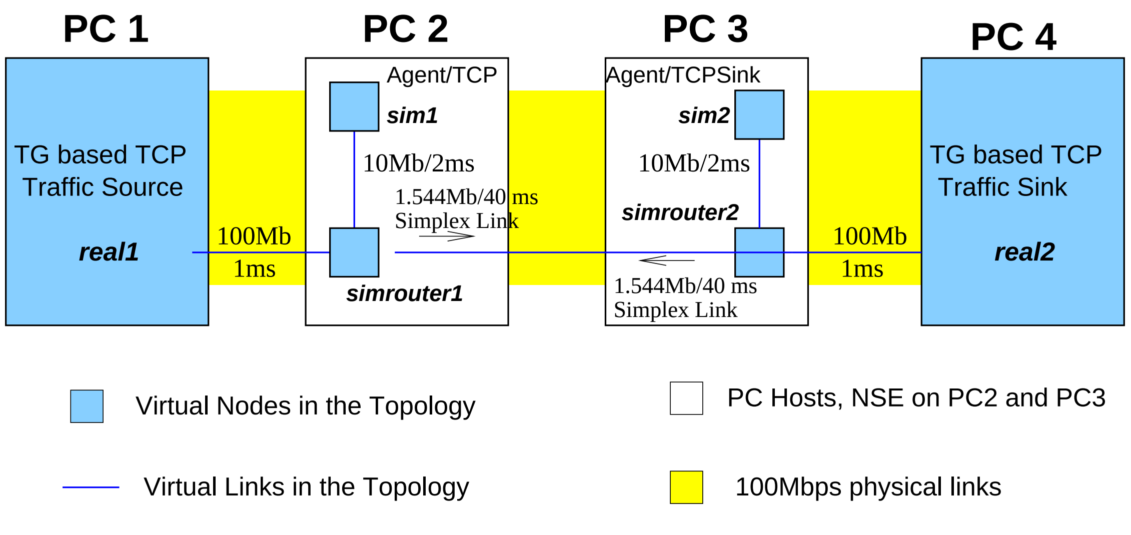

PC 1
PC 2
PC 3
PC 4
TG based TCP Traffic Source
real1
100Mb 1ms
Agent/TCP
sim1
10Mb/2ms
1.544Mb/40 ms Simplex Link
simrouter1
Agent/TCPSink
sim2
10Mb/2ms
simrouter2
1.544Mb/40 ms Simplex Link
100Mb 1ms
TG based TCP Traffic Sink
real2
Virtual Nodes in the Topology
PC Hosts, NSE on PC2 and PC3
Virtual Links in the Topology
100Mbps physical links

Figure 3.4. An integrated experiment in a "dumbbell" topology

3. Retrieve the experiment virtual topology and available physical resources from the database and perform resource assignment. Repeat on partial failures that may occur due to reasons such as another experiment acquiring some physical resources.

4. Reserve physical resources.

5. Update database with information specific to the mapping. For example, we list a few below:

   - Switch VLANs.
   - IP address bindings for network interfaces.
   - Creation of virtual network interfaces and their binding to physical interfaces for `rlinks`.
   - Compute routing table identifiers only for simulated nodes with `rlinks`.
   - OS that needs to run on the simulation host PCs.
   - Second parsing of the `make-simulated` block to generate OTcl subspecifications.

```
set ns [new Simulator]
# Enable automatic static routing
$ns rtproto Static

# Get two real PCs
set real1 [$ns node]; set real2 [$ns node]

# Use the standard FreeBSD image in Netbed
tb-set-node-os $real1 FBSD-STD; tb-set-node-os $real2 FBSD-STD

$ns make-simulated {
    # All the code here run in the simulation. Get 2 sim. end-nodes and 2 router-nodes
    set sim1 [$ns node]; set sim2 [$ns node]
    set simrouter1 [$ns node]; set simrouter2 [$ns node]

    # Bottleneck link inside simulation. Simulated and real traffic share this link
    $ns duplex-link $simrouter1 $simrouter2 1.544Mb 40ms DropTail

    # More duplex links inside the simulation
    $ns duplex-link $sim1 $simrouter1 10Mb 2ms DropTail
    $ns duplex-link $sim2 $simrouter2 10Mb 2ms DropTail

    # TCP agent object on node sim1 and TCPSink object on node sim2,
    #     both in simulation
    set tcp1 [new Agent/TCP]
    $ns attach-agent $sim1 $tcp1
    # FTP application object in simulation on node sim1
    set ftp1 [new Application/FTP]
    $ftp1 attach-agent $tcp1
    set tcpsink1 [new Agent/TCPSink]
    $ns attach-agent $sim2 $tcpsink1
    # Tell the system that $tcp1 and $tcpsink1 agents will talk to each other
    $ns connect $tcp1 $tcpsink1

    # Starting at time 1.0 send 75MB of data
    $ns at 1.0 "$ftp0 send 75000000"

    # Connecting real and sim. nodes. Allowed inside/outside make-simulated block
    $ns duplex-link $real1 $simrouter1 100Mb 1ms DropTail
}

# connecting real and simulated nodes.
$ns duplex-link $real2 $simrouter2 100Mb 1ms DropTail

# A real TCP traffic agent on PC real1
set tcpreal1 [new Agent/TCP]
$ns attach-agent $real1 $tcpreal1
set cbr0 [Application/Traffic/CBR]
$cbr0 attach-agent $tcpreal1

# A real TCP sink traffic agent on PC real2
set tcprealsink1 [new Agent/TCPSink]
$ns attach-agent $real2 $tcprealsink1
# Tell the system that $tcpreal1 will talk to # tcprealsink1
$ns connect $tcpreal1 $tcprealsink1

# Start traffic generator at time 10.0
$ns at 10.0 "$cbr0 start"

# Drastically reduce colocation factor for sim. nodes to show distributed NSE.
# With this, the 4 simulated nodes will be mapped to 2 PCs.
tb-set-colocate-factor 2

$ns run
```

Figure 3.5. *ns* code to create the integrated experiment illustrated in Figure 3.4

6. Load OS if required and boot the physical PCs.
7. Set up VLANs on switches.
8. Start up per-experiment event scheduler to deliver user-specified events.
9. Simulation host PCs perform self-configuration.

3.2.3 Parsing

User-specified experiment configuration such as the one given in section 3.2.1 has to be parsed before the experiment can be realized on various physical resources. We have extended the Emulab parser to store the simulated part of the experiment specification (enclosed in one or more `make-simulated` blocks) into the database "as is" for further parsing later. This is necessary since OTcl subspecifications can be generated only after the mapping phase. We will call this second parse an *nse parse*. This parse is similar to Emulab's initial parsing. The output of the *nse* parse is a set of OTcl subspecifications that are targeted to different simulator instances. Once generated, these subspecifications are stored in Emulab's database to be used during the experimental run. Essentially, a source to source translation is performed where both the source and target language are the same, i.e., OTcl. A single source results in one or more target OTcl scripts based on the mapping where each target script executes in an instance of *nse*. We describe our approach for this parse below.

Written in OTcl, the parser operates by overriding and interposing on standard *ns* procedures. Some key *ns* methods are overloaded. These methods use mapping information from Emulab's database to partition user-specified OTcl code into OTcl subspecifications for each simulator instance. For example, the `duplex-link` method of the `Simulator` class is normally used in creating links. The overloaded version of `duplex-link` generates either two `rlinks` if the endpoint nodes of the link are mapped to different simulator instances or regenerates the same `duplex-link`. Due to the structure of classes in *ns*, we are able to support a large portion of *ns* syntax in the `make-simulated` block. *ns* classes that are important for this parsing phase are `Simulator`, `Node`, `Agent` and `Application`. Links in *ns* are typically instantiated using the `duplex-link` method of the `Simulator` class. Traffic in *ns* has a simple two-layer model: transport and application. Subclasses of the `Agent` class normally implement the transport layer functionality of any protocol. These agents are all attached to a `Node` object. Similarly, subclasses of the `Application` class implement the application layer functionality of a protocol. The application objects are all attached to some agent. `Agent` and `Application` are thus directly or indirectly associated with a `Node` object, allowing the OTcl code for them all to be generated for a particular simulator instance. All simulation objects support the specification of per-object attributes via instance variables.

Classes in *ns* have structured names with the hierarchy delineated by the slash (`/`) character. For example, all subclasses of the `Agent` class have a `Agent/` prefix. Tcl/OTcl also supports `info` procedures that can help extract the state of the OTcl interpreter [59]. Similarly, an `unknown` method permits us to capture arbitrary method calls without any knowledge about them. Using the above features, we are able to construct the OTcl code to be given to different instances of *nse*. Note that most of the constructs are regenerated as they were specified by the experimenter while others such as links are transformed into `rlinks` if the incident nodes are mapped to different simulator instances. For user-specified events specified with the `at` method of the `Simulator` class, our overridden `at` method makes an attempt to map the event to a simulation object, which is further mapped to the simulator instance to which it needs to be delivered. The events thus mapped are thus stored in the database. OTcl code is not generated for user-specified events since they will be delivered via Emulab's central event scheduler as described in section 3.1.1 The following points outline a list of steps performed by the *nse* parser:

1. Concatenate all `make-simulated` blocks and store it in the database during first parse along with topology info.

2. Perform mapping using topology information from the database.

3. Initialize mapping info from the database in the *nse* parser (OTcl based).

4. Source the code in the `make-simulated` block into the parser creating objects based on overridden classes such as Simulator, Node, Agent etc., that we have defined. The base class object is created for any class name with a prefix of one of the above special classes. The actual name of the subclass is stored and will be used later to regenerate Tcl code. Objects of unrecognized classes are ignored.

5. Unrecognized global Tcl procedures are ignored. Note that unrecognized methods for the special classes mentioned above are all captured using the `unknown` method for the respective classes.

6. The last step is to generate OTcl code in this order of simulation objects : `Simulator`, `Node`, `duplex-link`, `rlink`, `Agent`, `Application`. The code generated will have method invocations as well as initialization of instance variables.

Our approach works well within the bounds of careful specification, numerous counterexamples of experiment specification can be constructed where our approach for parsing when using either Emulab frontend parser or *nse* parser will fail or is not adequate. For example, if specified code had dependencies on variable values of a running simulator, our approach fails. Another example is if an experimenter specified an OTcl loop to create a large number of simulation objects, our code generation will unroll all the loops, potentially causing code bloat that may be beyond the limits of our system. Some of these limitations can be overcome with more work but others are difficult to do so without writing a complete interpreter that understands all of *ns*. However, the ideas presented in this thesis are not in any way weakened by these limitations.

3.2.4 Self-configuration of Simulation Host PCs

The PCs that run *nse* instances boot up and contact Emulab *masterhost* to customize the setup of the OS in order to serve the role that it was designated for, in an experiment. The *masterhost* retrieves information from the database and returns them in response. The PCs receive information such as network identity (DHCP), shared filesystem (NFS) mounts, user accounts and keys, hosts file, network interface information (both real and virtual) and routing. These tasks were already performed by Emulab for cluster PC self-configuration. Once all tasks are performed, the PCs report back to *masterhost* that they are ready. We have extended the self-configuration process to download OTcl configuration from the *masterhost*, configure and run the simulator if the PCs were to serve as simulation hosts. The simulation hosts are configured to run FreeBSD. The disk images with this OS were updated with our modified *nse* binary as well as a supporting OTcl script. Our changes also ensured that the PC reported to the *masterhost* that it is ready only after *nse* is configured and running.

3.3 Virtual Networking

In this section, we discuss the details of the virtual link mechanism that allows us to multiplex several virtual links on a physical link and the implementation of the multiple routing table. We leveraged the virtual link mechanism that was added (by others) in Emulab to support multiplexed virtual nodes. We added initial support for multiple routing tables in FreeBSD which was later extended and completed by others, again for use in supporting multiplexed virtual nodes.

3.3.1 Design Issues in Virtual Links and LANs

In a general context, virtual links provide a way of multiplexing many logical links onto a smaller number of physical links. In this light, virtual links can be used to provide a higher degree of connectivity to nodes, whether those nodes are virtual or physical. In the context of virtual nodes, our discussion of virtual links includes not only this multiplexing capability but also the network namespace isolation issues and the subtleties of interconnecting virtual nodes within, and between, physical nodes. The implementation of *nse* packet capture and injection dictates our design choice.

The interesting characteristics of virtual links are:

- Level of virtualization.
- Encapsulation.
- Sharing of interfaces.

- Ability to co-locate virtual nodes.

3.3.1.1 Level of Virtualization

Virtual link implementations can present either a virtual link layer by providing a virtual ethernet device or a virtual IP layer by using multiple IP addresses per physical interface. The former is more flexible, allowing traffic other than IP, but such flexibility may not be needed for some applications. A virtual ethernet device is needed to support *nse* since it uses a BPF device to capture live packets. Although it is possible to support demultiplexing of packets by using more specific filter rules, it may not be practical to generate a rule that encompasses all packet source–destination pairs that may traverse the link.

3.3.1.2 Encapsulation

Virtual links may or may not encapsulate their packets. Encapsulation is traditionally used either to transport nonstandard protocols over deployed networks (e.g., tunneling over IPv4) or to support transparent (to end node) multiplexing capability (e.g., 802.1Q VLANs). Encapsulation usually implies a decrease in the MTU size for the encapsulated protocol, which can affect throughput.

3.3.1.3 Sharing of Interfaces

The end point of a virtual link as seen by a virtual node may be either a shared interface device or a private one. This may affect whether interface-centric applications like tcpdump can be used in a virtual node. For the same reasons specified above, a private interface device per-link is better for *nse*.

3.3.1.4 Ability to Co-locate Virtual Nodes

Three factors related to the implementation of virtual links influence which, if any, virtual nodes in the same topology or virtual nodes in different topologies can be co-located on a physical node. First, if virtual links are implemented using IP aliases on shared physical interfaces, then there are restrictions on what addresses can be assigned to the interface. For example, two nodes in different topologies could not have the same

IP address or two nodes in the same topology could not be part of the same virtual LAN. Second, if virtual links use a shared routing table, then two co-located nodes cannot have different next hop addresses for the same destination. Third, even with private routing tables, virtual links that cross physical links must convey demultiplexing information so that the receiving physical node can use the correct routing table for forwarding decisions. Without this, physical nodes cannot host multiple virtual nodes for the same topology. This is known as the “revisitation” problem [53].

3.3.2 Virtual Network Interfaces

In order to support virtual links, we leverage Emulab’s implementation of a virtual Ethernet interface device (veth). The veth driver is a hybrid of a virtual interface device, an encapsulating device and a bridging device. It allows us to create unbounded numbers of Ethernet interfaces (virtualization), multiplex them on physical interfaces or tie them together in a loopback fashion (bridging) and have them communicate transparently through our switch fabric (encapsulation). Virtualization gives us per-link interfaces above the physical interface which we can use as the underlying interface for a BPF capture device. Bridging allows the correct routing of packets at the link level so that virtual interfaces only receive the packets that they should. Encapsulation preserves the virtual link information necessary to implement revisitation when crossing physical links, without making any assumptions about the switching fabric. However, our switch fabric learns link-layer addresses as packets cross the switch. They can also support several link-layer addresses for the same switch port. Thus, Emulab veth device also supports an alternative to encapsulation by using fake link-layer addresses.

3.3.3 Virtual Routing Table

We have adopted and extended the work of Scandariato and Risso [49] which implements multiple IP routing tables to support multiple VPN end points on a physical node. Routing tables are identified by a small integer routing table ID (rtabid). These IDs are the glue that bind together simulated nodes with `rlinks`, virtual interfaces and routing tables. Simulated nodes that have `rlinks` use a separate routing table with a unique rtabid to make sure packets injected by the node will use the correct routing table to find the next hop. Using a socket option to set the rtabid on a RAW IP `socket`, which is used to inject packets into the live network, a simulated node is able to ensure correct behavior.

3.4 Auto-Adaptation of Simulated Resources

A mapping of simulated resources to physical resources should avoid “overloading” any pnode in the system, which was discussed in detail in section 2.4. The workload to which an instance of *nse* is subjected is not easily determined statically in an integrated experiment, partly because an experimenter can generate arbitrary traffic without specifying its nature a priori. An overloaded pnode will result in simulation inaccuracies. In the case of simulated resources, these inaccuracies occur because the simulator is not able to dispatch all events in real-time. A similar issue also arises when multiplexing “virtual machine” type vnodes on pnodes. In order to solve this issue, we perform auto-adaptation when overload is detected by iterative mapping. Successive mappings use feedback data from running the experiment with prior mappings, until no overload is detected or we run out of physical resources. Such a solution for “virtual machine” type vnodes is discussed elsewhere [25]. In this section, we focus on performing iterative mapping for simulated resources.

The factors that make it feasible for us to perform auto-adaptation are:

- Fast mapping.
- Fast pnode reconfiguration

3.4.1 Fast Mapping

This was discussed in section 2.4. A mapping that takes hours is clearly too slow. Iterative mapping reduces the search space by remapping only the portions of the topology that were mapped to pnodes reporting an overload.

3.4.2 Fast Pnode Reconfiguration

Iterative mapping is affected by the speed of *reconfiguring* pnodes for the new mapping, both pnodes currently reserved to the experiment and new ones that may be allocated as more resources are needed. Current PC hardware can take long enough to boot that this starts to affect remapping time. Emulab in a recent optimization, now avoids doing full reboots by having unused pnodes wait in a "warm" state in the boot loader. This boot loader has the ability to boot into different disk partitions, and to boot different kernels within those partitions. Pnodes that were already part of the experiment are reconfigured without rebooting. This involves pushing all the Emulab client configuration data to the pnode, reconfiguring interfaces, routing tables, and a new simulation.

Initial mapping is guided by optimistic vnode co-locate factors per pnode type in Emulab. A more powerful PC supports a higher co-locate factor than a less powerful one. The co-locate factor is intended as a coarse grained metric for CPU and memory load on a pnode. In simulations with lots of traffic, the CPU bottleneck is typically reached much earlier than memory limits are reached. Also, if different amounts of traffic are passing through different vnodes, their resource consumptions will be different. Considering these problems, the co-locate factor we choose is based only on a pnode's physical memory. Based on feedback data obtained from running the simulations, we hope to quickly converge to a successful experiment if the initial mapping is too optimistic. A simulated vnode in *nse* consumes only moderate amounts of memory, allowing us to support a large co-locate factor. According to a study that compared several network simulators [39], *ns* allocated roughly 100KB per connection, where each connection consists of two nodes with two duplex-links that each add new branches to a "dumbbell" topology. Each connection consisted of a TCP source and sink on the leaves of the dumbbell. On a Emulab PC with 256–512MB of memory, a fairly large co-locate factor can be supported.

In order to determine "overload" and declare the simulation to be in "violation," we make use of the "slop factor" of a real-time simulation. The "slop factor" is the largest skew allowed between the simulated virtual clock and the real-time physical clock. We can have different meanings for "overload". One definition is to consider the simulation to be overloaded at a certain "slop factor" if the slop is exceeded even once. Another definition would consider a true "overload" in which the simulation is constantly exceeding the slop factor at a certain slop factor instead of exceeding the slop factor once or infrequently. In this thesis, we use the first definition where a slop factor exceeding even once is considered a violation. However, we explore different "slop factors."

When an overload is detected by a simulator instance, it reports all necessary information to Emulab *masterhost* via the event system. On receiving the first such event, a program on the *masterhost* is run that waits for several seconds, giving sufficient time for other pnodes to report overload if present. This program stores the feedback data into the database and begins remapping the experiment.

We outline two heuristics that we separately experiment with to guide auto-adaptation:

- Doubling vnode weights.
- Vnode packet-rate.

3.4.3 Doubling Vnode Weights

A coarse heuristic that we use is to double the weight of all the simulated nodes hosted on the pnode that reported an "overload" and remap the topology. These simulated nodes will then consume twice as many slots from the pnode co-locate factor as before. This process repeats untill no overload is detected or a vnode is mapped one-to-one to an overloaded pnode. If the overload is still present, it means that the experiment could not be mapped on Emulab hardware.

3.4.4 Vnode Packet-Rate

Simulation event-rate is proportional to the rate of packets that pass through a vnode or are generated by that vnode. This is because every packet typically causes roughly a constant number of events. For packet forwarding, even though events in *ns* occur in links, the cost of processing these events can be attributed to the vnode to which such links are connected. Because the Emulab mapper, `assign`, associates resource capacities with pnodes and resource use with vnodes, we use the rate of packets passing through a vnode as the cost. Based on packet-rate measurements we have taken (section 4.1), we set the pnode packet-rate capacities. This is a fine-grained heuristic compared to the previous one. Starting from an optimistic mapping, we can easily identify the vnodes that are “heavyweight,” allowing subsequent mappings to pack such vnodes less tightly.

Section 4.3 examines the results of using each of these heuristics listed above.

CHAPTER 4

EVALUATION

In this chapter, we present some results to establish the base performance of *nse* on Emulab hardware. We then present results that show the similarity (and differences) when multiple simulator instances are used. We collect packet traces and evaluate using following methods:

- Compare aggregate measures such as throughput.
- Compare packet traces for first-order statistics such as packet interarrival
- Multiscale analysis of packet traces
- Compare queueing behavior

Lastly, we present the results of our experiments with auto-adaptation using two different heuristics.

4.1 Base Performance of *nse*

4.1.1 Capacity and Accuracy

We have obtained some results that show the capacity, in packet processing rate per instance of *nse* on Emulab PCs, and the accuracy of *nse* as a link emulator. During this evaluation, a number of problems and bugs were uncovered with *nse* that we have since solved.

As a capacity test, we generated streams of UDP round-trip traffic between two nodes, with an interposed 850Mhz PC running *nse* on a FreeBSD 4.5 1000HZ kernel. A maximum stable packet rate of 4000 packets per second was determined over a range of packet rates and link delays using 64-byte and 1518-byte packets. Since these are round trip measurements, the packet rates are actually twice the numbers reported. With this capacity, we performed experiments to measure the delay, bandwidth and loss rates for representative values. The results measured recently are summarized in Tables 4.1, 4.2 and 4.3. These tables also report corresponding results of the Dummynet emulator for comparison [60].

The high error rates that we see in uniform loss rate measurements are present even in pure simulation and is suspected to be a bug in *ns*.

Table 4.1. **Delay:** Accuracy of observed Dummynet and *nse* delay at maximum packet rate as a function of packet size for different link delays. The 0ms measurement represents the base overhead of the link. Adjusted RTT is the observed value minus the base overhead.

delay	packet	observed Dummynet			adjusted Dummynet	
(ms)	size	RTT	stdev	% err	RTT	% err
0	64	0.177	0.003	N/A	N/A	N/A
	1518	1.225	0.004	N/A	N/A	N/A
5	64	10.183	0.041	1.83	10.006	0.06
	1518	11.187	0.008	11.87	9.962	0.38
10	64	20.190	0.063	0.95	20.013	0.06
	1518	21.185	0.008	5.92	19.960	0.20
50	64	100.185	0.086	0.18	100.008	0.00
	1518	101.169	0.013	1.16	99.943	0.05
300	64	600.126	0.133	0.02	599.949	0.0
	1518	600.953	0.014	0.15	599.728	0.04

delay	packet	observed *nse*			adjusted *nse*	
(ms)	size	RTT	stdev	% err	RTT	% err
0	64	0.233	0.003	N/A	N/A	N/A
	1518	1.572	0.030	N/A	N/A	N/A
5	64	10.226	0.016	2.26	9.993	0.07
	1518	11.575	0.058	15.75	10.003	0.03
10	64	20.241	0.023	1.21	20.008	0.04
	1518	21.599	0.071	8.00	20.027	0.14
50	64	100.239	0.024	0.24	100.006	0.006
	1518	101.617	0.078	1.62	100.045	0.05
300	64	600.244	0.029	0.04	600.011	0.002
	1518	601.612	0.078	0.27	600.040	0.007

Table 4.2. **Bandwidth:** Accuracy of observed Dummynet and *nse* bandwidth as a function of packet size for different link bandwidths

bandwidth (Kbps)	packet size	observed Dummynet bw (Kbps)	observed Dummynet % err	observed *nse* bw (Kbps)	observed *nse* % err
56	64	56.06	0.11	55.013	0.023
	1518	56.67	1.89	56.312	0.556
384	64	384.2	0.05	384.015	0.004
	1518	385.2	0.34	384.367	0.096
1544	64	1544.7	0.04	1544.047	0.003
	1518	1545.8	0.11	1544.347	0.022
10000	64	10004	0.04	N/A	N/A
	1518	10005	0.05	10000.519	0.005
45000	1518	45019	0.04	45001.092	0.002

Table 4.3. **Loss:** Accuracy of observed Dummynet and *nse* packet loss rate as a function of packet size for different loss rates

packet loss rate (%)	packet size	observed Dummynet loss rate (%)	observed Dummynet % err	observed *nse* loss rate (%)	observed *nse* % err
0.8	64	0.802	0.2	0.818	2.29
	1518	0.803	0.3	0.809	1.15
2.5	64	2.51	0.4	2.469	1.22
	1518	2.47	1.1	2.477	0.92
12	64	12.05	0.4	11.88	1.00
	1518	12.09	0.7	11.98	0.21

4.1.2 Scalability of Traffic Flows

In order to evaluate the scalability in the number of flows that a single instance of *nse* could support on a Emulab PC, we simulated 2Mbps constant bit rate UDP flows between pairs of nodes on 2Mbps links with 50ms latencies. To measure *nse*'s ability to keep pace with real time, and thus with live traffic, a similar link was instantiated inside the same *nse* simulation, to forward live TCP traffic between two physical Emulab nodes, again at a rate of 2Mbps. On an 850MHz PC, we were able to scale the number of simulated flows up to 150 simulated links and 300 simulated nodes, while maintaining the full throughput of the live TCP connection. With additional simulated links, the throughput dropped precipitously. We also measured *nse*'s TCP model on the simulated links: the performance dropped after 80 simulated links due to a higher event rate from the acknowledgment traffic in the return path.

4.1.3 Sensitivity to Different Slop Factors

The "slop" factor is the largest skew allowed between the simulated virtual clock and the real-time physical clock. It is a configurable parameter provided to the real-time simulation. If the skew exceeds the slop factor, a simulation is deemed to be in "violation." In order to determine how the scalability of the number of flows is affected by the slop factor, we simulated TCP traffic over 2Mbps links with 50ms latencies between pairs of nodes on an Emulab PC running at 850Mhz. In this experiment, we do not have any external traffic, although the cost of a Unix system call is still incurred since the real-time scheduler makes a call to `select()` for the presence of live packets. In Table 4.4, we report the number of TCP flows at which a violation is detected at different slop factors. We also report the run-time when the same workload is run under pure simulation in *ns*. Under *ns*, the workload we generated had a run-time of 42.46 seconds. We see that the slop of 100μs and 1ms is exceeded at a small number of flows even though pure simulation runs much faster than real-time. A 10ms slop factor provides us with a better tolerance for simulator clock skews as we scale the pure simulation to when it runs slightly faster than real-time. A possible reason for these observations is that the changes of exceeding 1ms slop is high because the OS that runs the real-time simulation runs at 1000HZ scheduling intervals.

Table 4.4. Sensitivity of slop factor on the number of simulation flows

Slop Factor	Number of flows when skew exceeds slop factor	Event Rate	Wall-clock time when run in *ns*
100μs	1	942	$<$ 1 second
1ms	3	2826	1 second
10ms	66	62176	41 seconds

4.2 Validation of Distributed *Nse*

When simulated resources in an integrated experiment are mapped to multiple PCs, some of the flow endpoints also get mapped to different *nse* instances on different PCs. To determine how similar are packet flows inside a single instance of *nse* compared to the ones that cross physical (switched) links, we perform the following experiment:

The basic experimental setup consists of two simulated nodes connected by a T1-like duplex-link of 1.544Mbps bandwidth and 25ms latency.[1] Traffic is generated using `Agent/TCP` which is an abstract implementation of the BSD Tahoe TCP protocol [3]. About 75MB of simulated data bytes are transferred over this connection in one direction. This gives us a trace of about 50,000 data packets and about the same number of ACK packets in the reverse direction. In the rest of this section, a TCP-sink is the endpoint which receives DATA packets while a TCP-source refers to the one that sends DATA packets (and receives ACKs). The simple model described above is useful in establishing a lower bound on the difference between absolute repeatable simulations and emulations using distributed *nse*.

The above setup is realized under the following scenarios:

1. Both simulated nodes are in one instance of *nse*, i.e., pure real-time simulation. Whenever we use *RTSIM* anywhere in this section, we mean this scenario. Unless *nse* falls behind real-time due to an intensive workload, these results are the same as that of pure simulation. We have verified that all RTSIM results reported in this section exactly match pure *ns* simulation (i.e., running in discrete virtual time) which runs faster than real-time for this workload.

2. Each simulated node is in a different instance of *nse* on two different PCs connected via a 100Mbps switched Ethernet link. Each instance of *nse* simulates one node and the outgoing link to the other node. The physical 100Mbps link is simply

[1]This latency of the link roughly models traversing an uncongested, intracontinental piece of the terrestrial Internet [50].

used as a transport for the encapsulated simulator packets.[2] We will refer to this scenario in this section by *DIST-RTSIM*. Figure 3.2 illustrates the setup in detail.

3. The above scenario is replicated to have 60 simulated **T1** links mapped to the same 100Mbps switched Ethernet link. Each instance of *nse* is handling approximately 7646 packets per second which is within the stable capacity of *nse* on this hardware, as reported in section 4.1. Note that these are encapsulated packets roughly about 100 bytes in size resulting in 6–7% utilization of a 100Mbps Ethernet link. The simulated nodes on each end for these links are mapped to two different *nse* instances running on two PCs. This setup is useful in identifying the effects of multiplexing packets from independent virtual links over the same physical link. We will refer to this scenario in this section as *DIST-RTSIM-60*

The platform on which we run *nse* is a 850Mhz Pentium-III PC with FreeBSD 4.9 for all three tests listed above. We now present comparisons between RTSIM, DIST-RTSIM and DIST-RTSIM-60.

4.2.1 Comparison of Aggregate Measures

Table 4.5 shows how the aggregate throughput for the TCP flow described in the setup above compares between RTSIM, DIST-RTSIM and DIST-RTSIM-60. For the latter case, of the 60 flows, we show flows with both best and worst percentage errors from the expected value. For the experimental setup described above, the expected value is 1.544Mbps. As we see below, the difference between all three experiments is imperceptible at the aggregate level.

4.2.2 Comparison of Packet Interarrivals

Comparing packet interarrivals provides us with a better insight into the effect of the OS, network device and the physical network on the packet traffic. Tables 4.6 and 4.7 compare the mean, standard deviation and 95% confidence interval for the mean,

[2]The size of the encapsulated packets does not always depend on the simulated packet size since packet data is not typically included. In the experiments performed here, the encapsulated packet size including the IP and Ethernet headers was about 100 bytes.

Table 4.5. Throughput comparisons between RTSIM, DIST-RTSIM and DIST-RTSIM-60

Experiment	Throughput (Kbps)	Percentage Error (%)
RTSIM	1543.767	0.0151
DIST-RTSIM	1543.669	0.0214
DIST-RTSIM-60 (best)	1543.782	0.0141
DIST-RTSIM-60 (worst)	1543.761	0.0153

Table 4.6. Packet Interarrival comparisons at the TCP-sink between RTSIM, DIST-RTSIM and DIST-RTSIM-60

Experiment	Mean (μs)	Standard Deviation (μs)	95% Confidence Interval for the mean
RTSIM	7846.84	312.05	7844.16 – 7849.53
DIST-RTSIM	7846.77	314.22	7844.07 – 7849.48
DIST-RTSIM-60 (**best**, using `select()`)	7846.80	395.65	7843.39 – 7850.21
DIST-RTSIM-60 (**best**, using `kqueue()`)	7846.79	323.42	7844.01 – 7849.58
DIST-RTSIM-60 (**worst**, using `select()`)	7846.85	644.87	7841.29 – 7852.40
DIST-RTSIM-60 (**worst**, using `kqueue()`)	7846.82	395.51	7843.41 – 7850.22

for the packet data at the TCP-sink and TCP-source, respectively. For the experiment DIST-RTSIM-60, the *best* data point corresponds to the flow that has the least variance and correspondingly the *worst* data point is for a flow with the highest variance. In all these results, unless mentioned otherwise, we use the values of the simulator clock – just before the packets are delivered to the traffic agent to compute interarrival times.

Results from the DIST-RTSIM experiment are very close to RTSIM providing us with some confidence about the similarity of results between the two. However, both the best and the worst case results for DIST-RTSIM-60 have a higher standard deviation compared to the other two cases. In order to determine the source of this variability, we looked at kernel timestamps provided by the Berkeley packet filter (BPF). These timestamps are stored by the kernel for every packet stored in the BPF buffer and is

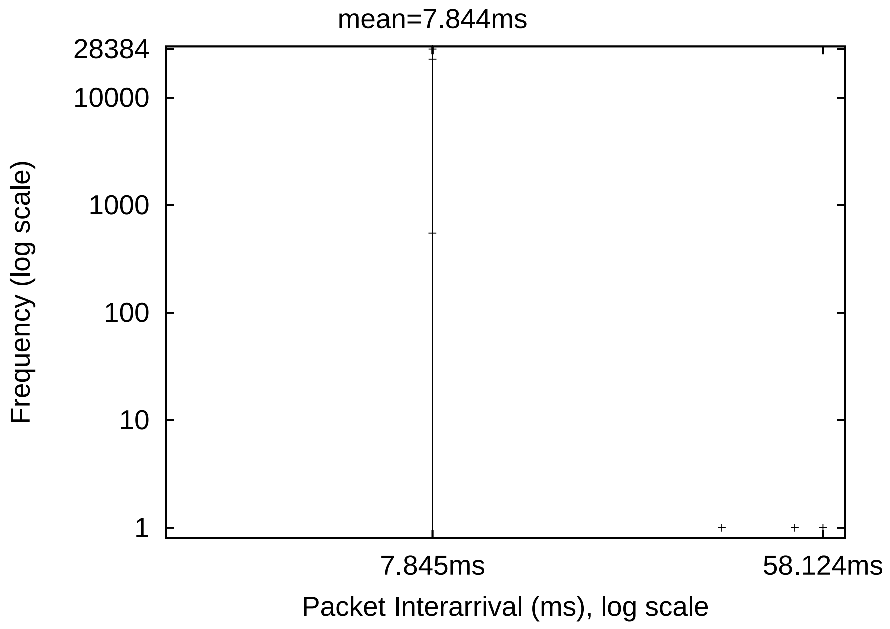
Table 4.7. Packet Interarrival comparisons at the TCP-source between RTSIM, DIST-RTSIM and DIST-RTSIM-60

Experiment	Mean (μs)	Standard Deviation (μs)	95% Confidence Interval for the mean
RTSIM	7846.84	312.05	7844.16 – 7849.53
DIST-RTSIM	7846.86	314.01	7844.16 – 7849.56
DIST-RTSIM-60 (**best**, using `select()`)	7846.88	498.50	7842.59 – 7851.18
DIST-RTSIM-60 (**best**, using `kqueue()`)	7846.87	345.42	7843.89 – 7849.84
DIST-RTSIM-60 (**worst**, using `select()`)	7846.87	852.46	7839.53 – 7854.21
DIST-RTSIM-60 (**worst**, using `kqueue()`)	7846.87	522.96	7842.36 – 7851.37

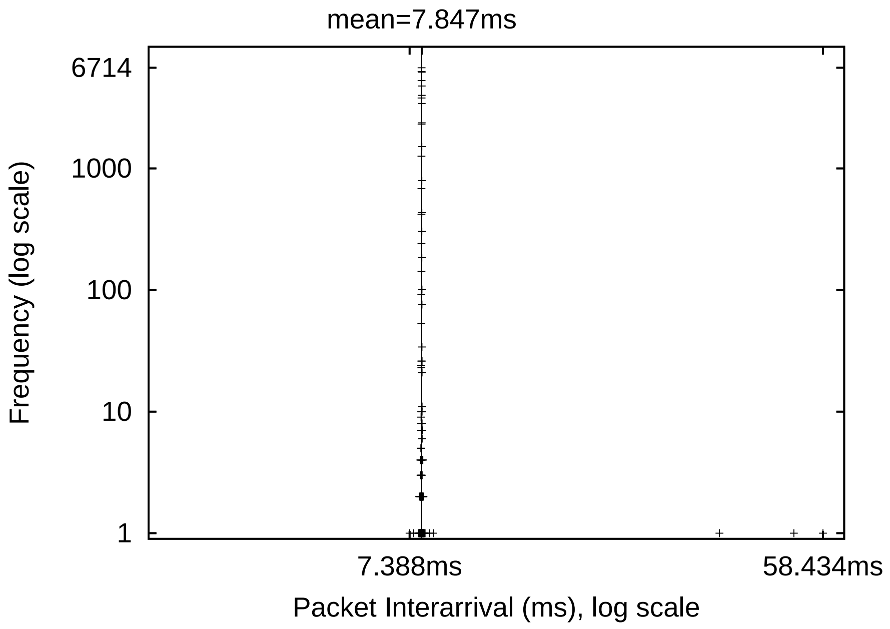
made available to the user-space program– *nse* in this case– that reads the BPF buffer. The interarrival distribution using these timestamps for the TCP-sink gives us a best case standard deviation of 321.34μs and a worst case of 359.93μs among the 60 flows. Therefore, the majority of the variability could be attributed to the overhead of reading packets from the kernel buffer. In particular, the Unix `select()` system call– which is used to check for I/O readiness– is not very efficient when examining large numbers of file descriptors. The TCP-source ACK-packet interarrivals are affected further more because of encountering the above effects at both TCP-sink and TCP-source endpoints. We evaluated an alternative method, namely FreeBSD `kqueue()` [34], for I/O readiness which is known to be more efficient.[3]

In Figures 4.1, 4.2, and 4.3, we present the frequency plots at the TCP-sink when only one flow is present, as well as, when 60 flows are present using `select()`, and `kqueue()`, respectively. Similarly, in Figures 4.4, 4.5 and, 4.6, we present frequency plots for the packet interarrivals at the TCP-source.

[3]However, as we found out, `kqueue()` support for BPF devices is fairly recent as well as buggy. This was confirmed by a FreeBSD bug report [2]. The data reported in Tables 4.6 and 4.7 were obtained after applying the fix for the aforementioned bug.

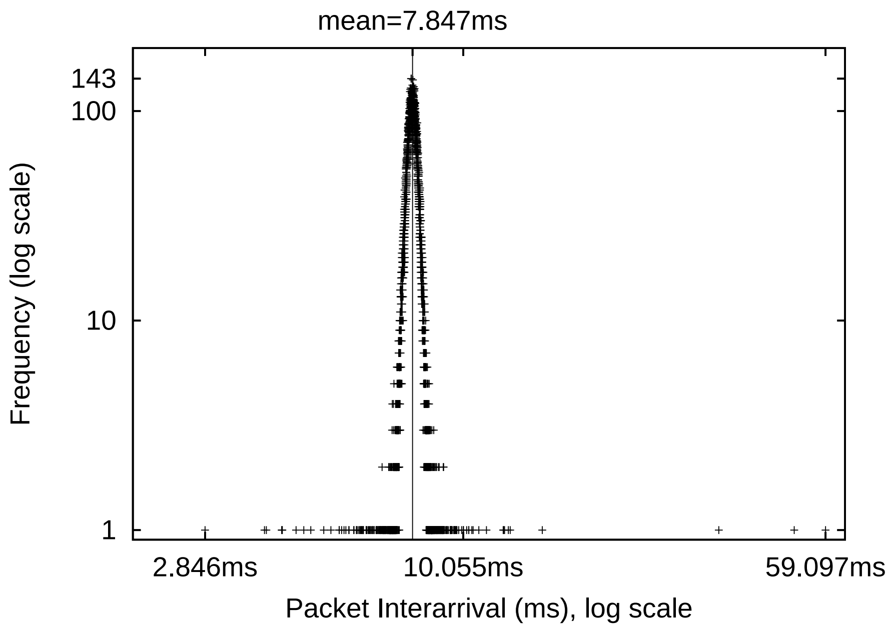


(a) RTSIM

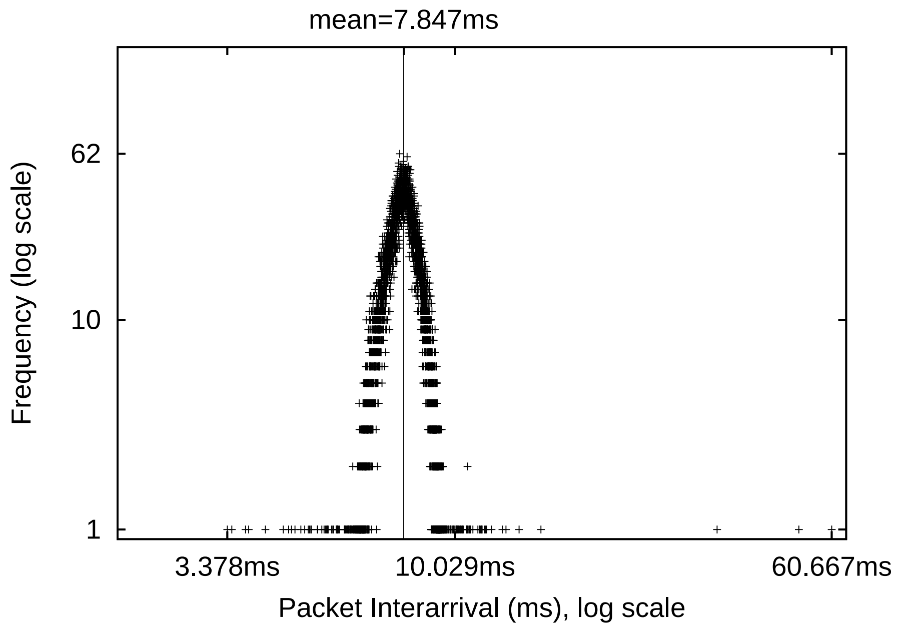


(b) DIST-RTSIM

Figure 4.1. Comparison of frequency of packet interarrivals at TCP-sink for one flow

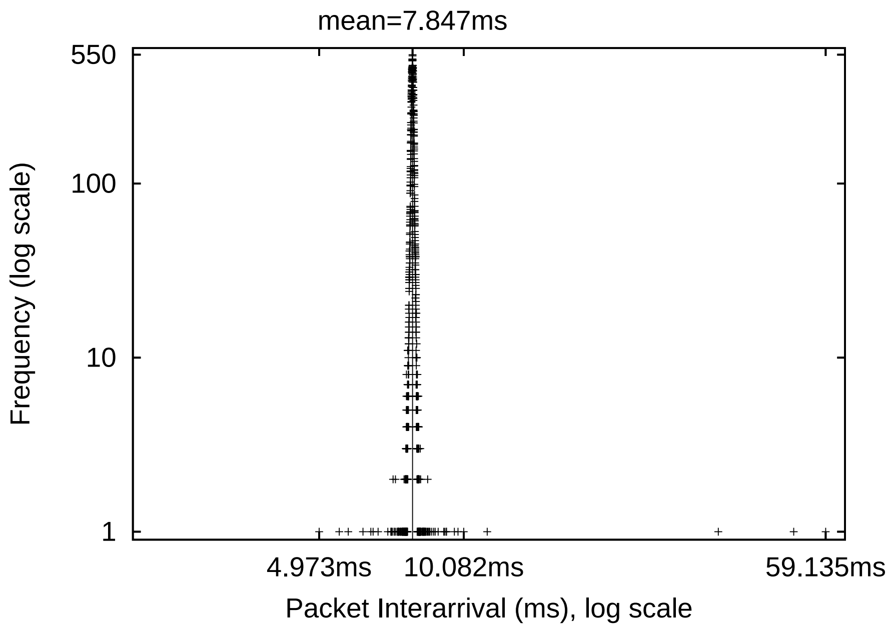


(a) DIST-RTSIM-60 (best, using `select()`)

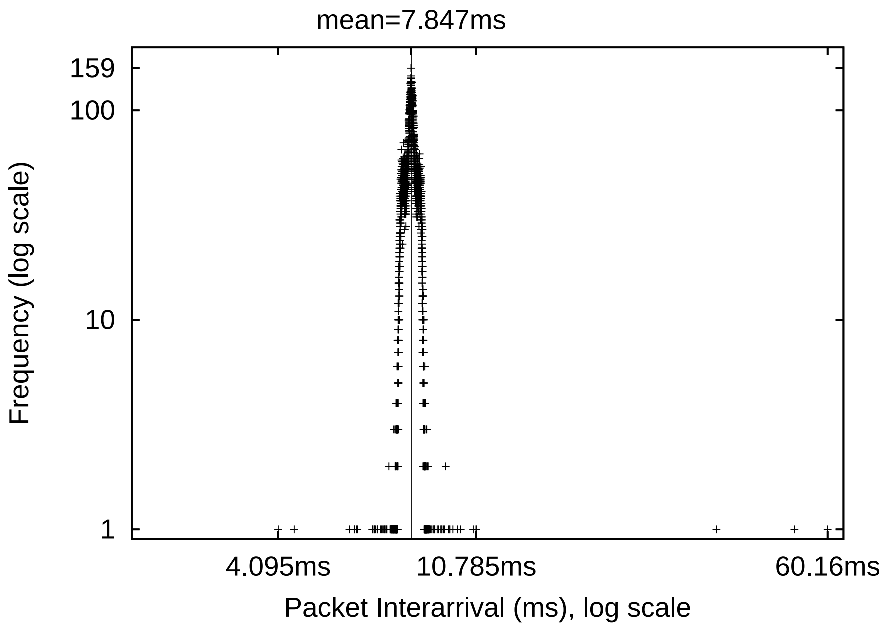


(b) DIST-RTSIM-60 (worst, using `select()`)

Figure 4.2. Comparison of frequency of packet interarrivals at TCP-sink using `select()` when 60 flows are present

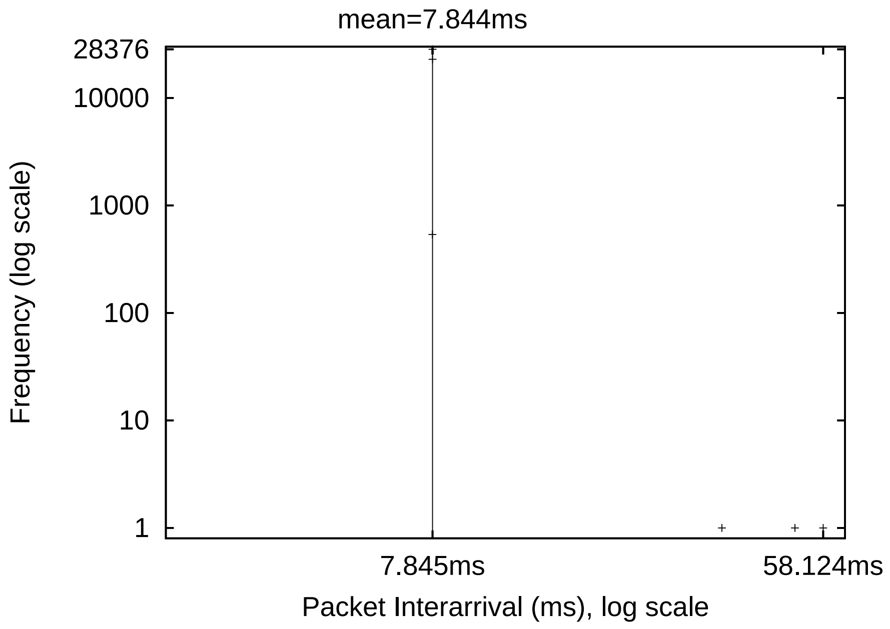


(a) DIST-RTSIM-60 (best, using `kqueue()`)

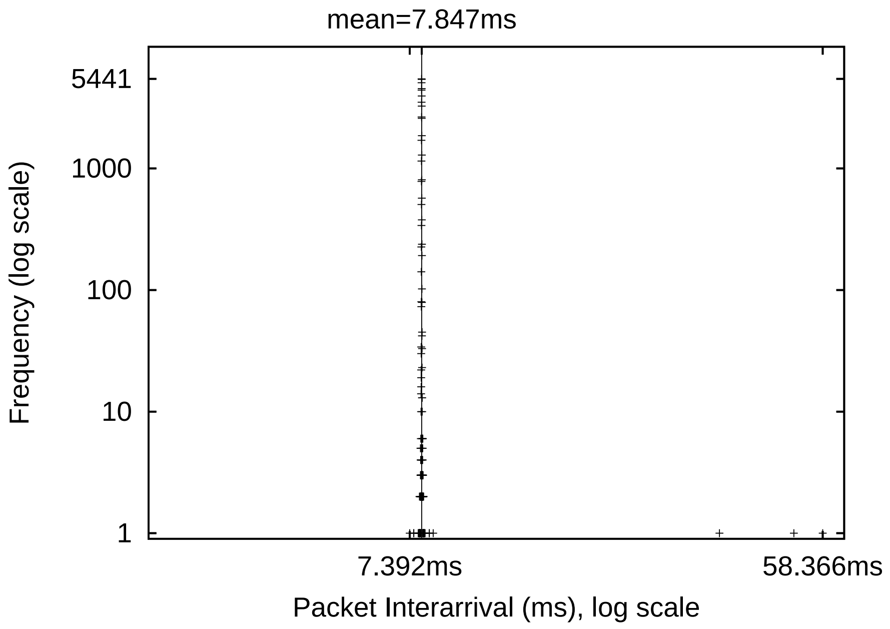


(b) DIST-RTSIM-60 (worst, using `kqueue()`)

Figure 4.3. Comparison of frequency of packet interarrivals at TCP-sink using `kqueue()` when 60 flows are present

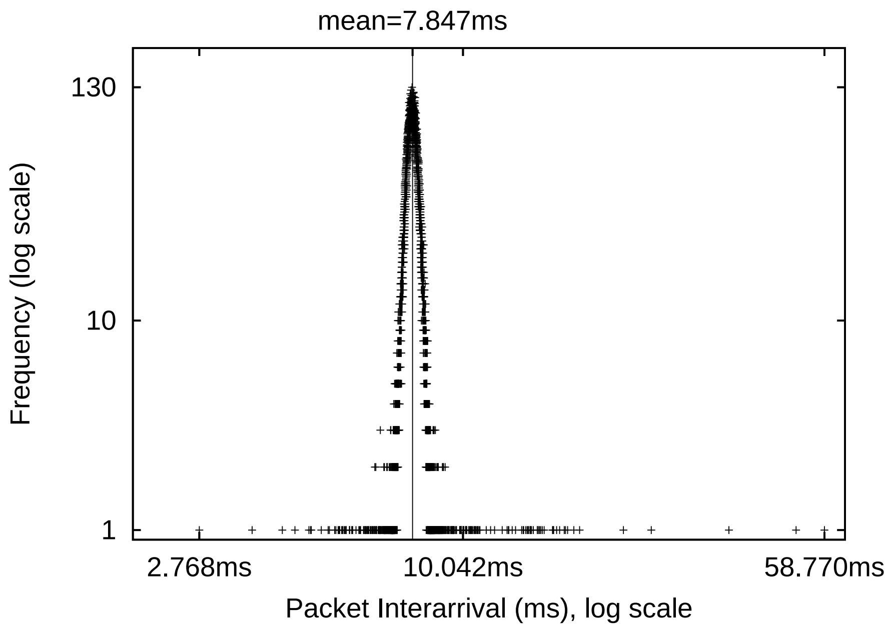


(a) RTSIM

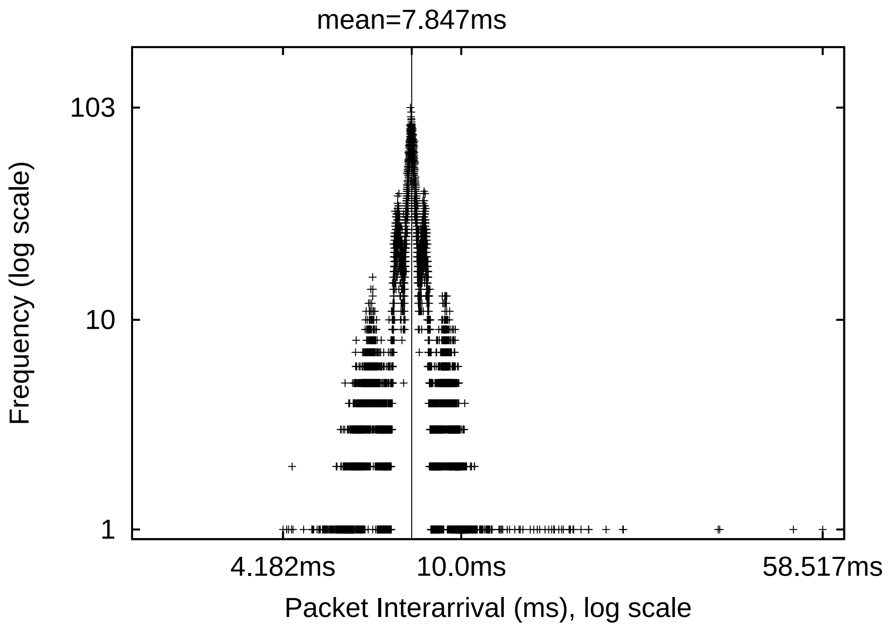


(b) DIST-RTSIM

Figure 4.4. Comparison of frequency of packet interarrivals at TCP-source for one flow

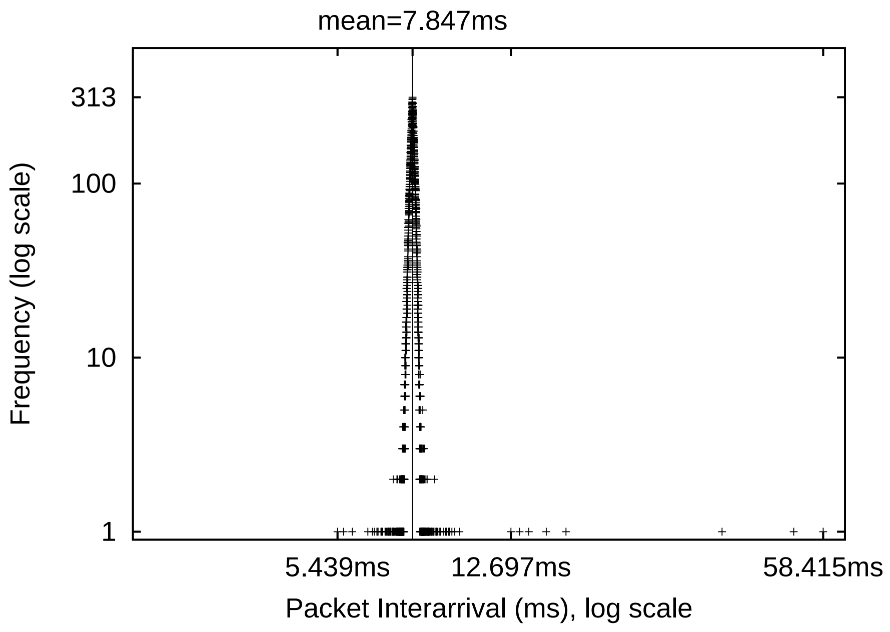


(a) DIST-RTSIM-60 (best, using `select()`)

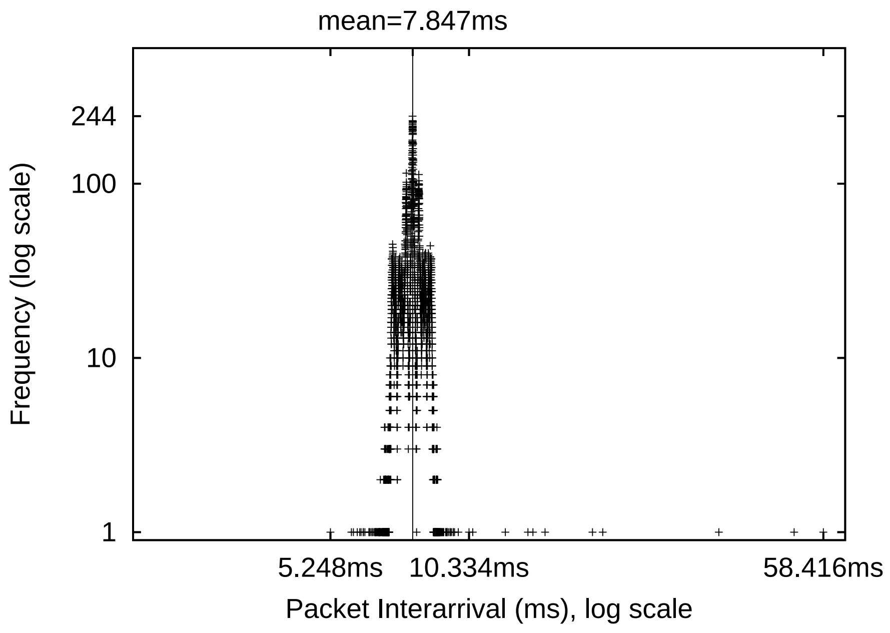


(b) DIST-RTSIM-60 (worst, using `select()`)

Figure 4.5. Comparison of frequency of packet interarrivals at TCP-source using `select()` when 60 flows are present



(a) DIST-RTSIM-60 (best, using `kqueue()`)



(b) DIST-RTSIM-60 (worst, using `kqueue()`)

Figure 4.6. Comparison of frequency of packet interarrivals at TCP-source using `kqueue()` when 60 flows are present

The variability of the packet interarrival progressively increases from RTSIM to DIST-RTSIM-60. It is also clear that using `kqueue()` is better than using `select()` when examining large numbers of file descriptors for I/O readiness.

4.2.3 Multiscale Analysis

It is well established that both wide-area and local-area data network traffic is bursty across a wide range of time-scales. This is explained by the notion of *distributional self-similarity*, which is that the correlational structure of the time-series for the traffic remains unchanged at varying time scales [33, 20]. A self-similar time-series exhibits bursts—- extended periods above the mean—- at a wide range of time scales. Several statistical techniques – of which we name one here: *time–variance* plots– are available to test for the presence of self-similarity.

Multiscale analysis techniques such as *time–variance plots* [16] are usually used to test the presence of self-similarity. However, such a plot has a useful property that it can sometimes show differences in traffic that simple aggregate measures or first-order statistical comparisons might fail to capture [32]. In this thesis, we use the time–variance plot to determine if RTSIM, DIST-RTSIM and DIST-RTSIM-60 have different properties across timescales even when the latter has higher packet-interarrival variance than the former.

4.2.3.1 Time-variance Plot

Let $X = (X_t : t = 0, 1, 2, ...)$ be a stationary time series. An example for X_t is a traffic rate process, i.e., the number of bytes seen in the last time interval. We define $X^{(m)} = (X^{(m)} : k = 1, 2, 3, ...)$ by averaging the original series X over nonoverlapping blocks of size m. For each $m = 1, 2, 3, ...$,

$$X_k^{(m)} = \frac{1}{m}(X_{km-m+1} + ... + X_{km}), k = 1, 2, 3, ... \tag{4.1}$$

We obtain the *time–variance* plot by plotting the variance of $X^{(m)}$ against m on a $log10 - log10$ scale. A straight line with a slope$(-\beta)$ where $-\beta$ is greater than -1 is often indicative of a self-similar process with “Hurst parameter” $H = 1 - \frac{\beta}{2}$ [16]. In other words, a self-similar time-series exhibits a slowly decaying variance.

Note that the aim of generating time–variance plots in this thesis is to visually compare any differences across time-scales and not to make any conclusions about the scaling properties of the simulation traffic in the experiment described above or its validity with respect to the real-world behavior.

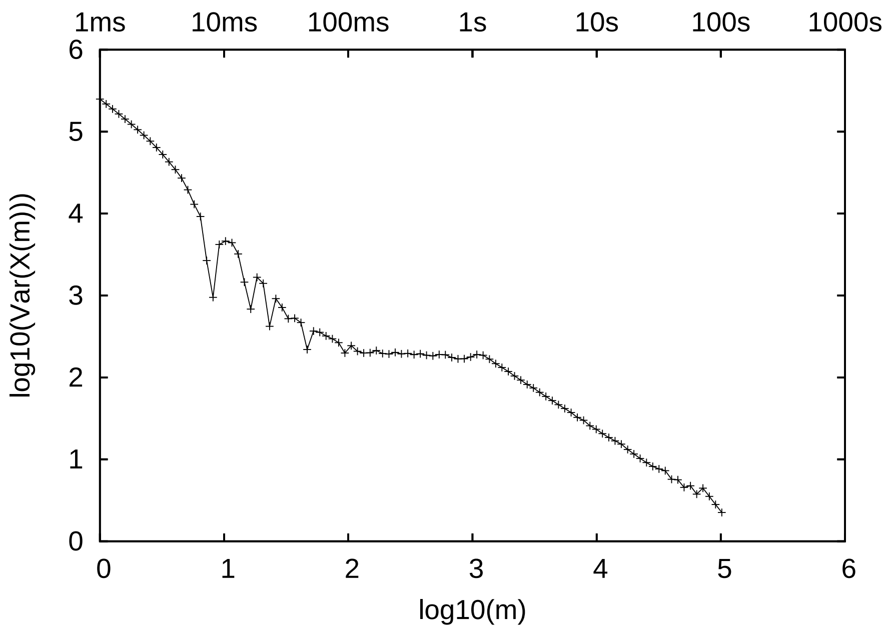
Figures 4.7, 4.8, and 4.9 plot the time–variance plots for the time-series data obtained at the TCP-sink. Similarly, Figures 4.10, 4.11, and 4.12 plot the time–variance plots for the time-series data obtained at TCP-source. Notice that the basic shapes for both the sink and source are quite similar modulo a constant shift in the variance (or roughly so) across all time scales.

The only perceptible difference between RTSIM and DIST-SIM-60 occurs at the first trough in the plot close to 10ms aggregation. At this point, the RTSIM time-series has less variance compared to DIST-SIM-60. The best values for both `select()` and `kqueue()` in DIST-SIM-60 are closer to RTSIM than the corresponding worst values. Also, `kqueue()` gets us closer to RTSIM than `select()` at this time-scale. Beyond this time-scale, the plots are all more or less identical. Thus, the variability that we saw in DIST-RTSIM-60 in Figures 4.2, 4.3, 4.5, and 4.5 do not have any noticeable effect at longer time-scales.

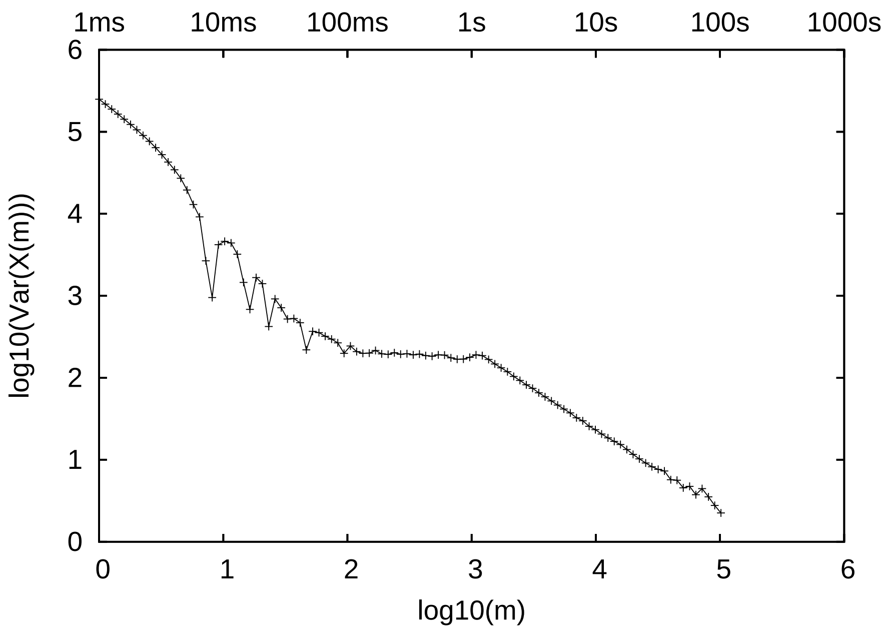
4.2.4 Comparison of Queueing Behavior

In sections 4.2.1 and 4.2.3, we compared the traffic rate characteristics of the distributed *nse* with one instance of *nse* using an experimental setup described in section 4.2. In this section, we analyze queueing behavior. The experimental setup for this comparison is described below:

The setup consists of two simulated nodes connected by a T1-like duplex-link similar to the experimental setup described in section 4.2. A total of six TCP flows and one UDP flow are instantiated on this link. The link queue size is set to 100 packet slots. The TCP flows are of the `Agent/TCP/Newreno` flavor in *ns*. Traffic due to one of the flows starts 0.5 seconds after the other five flows start.

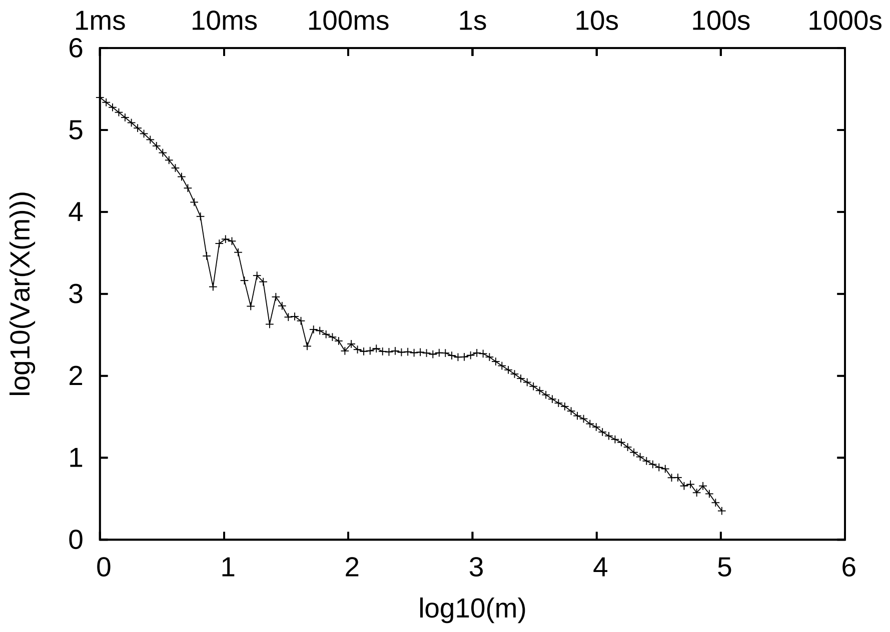


(a) RTSIM

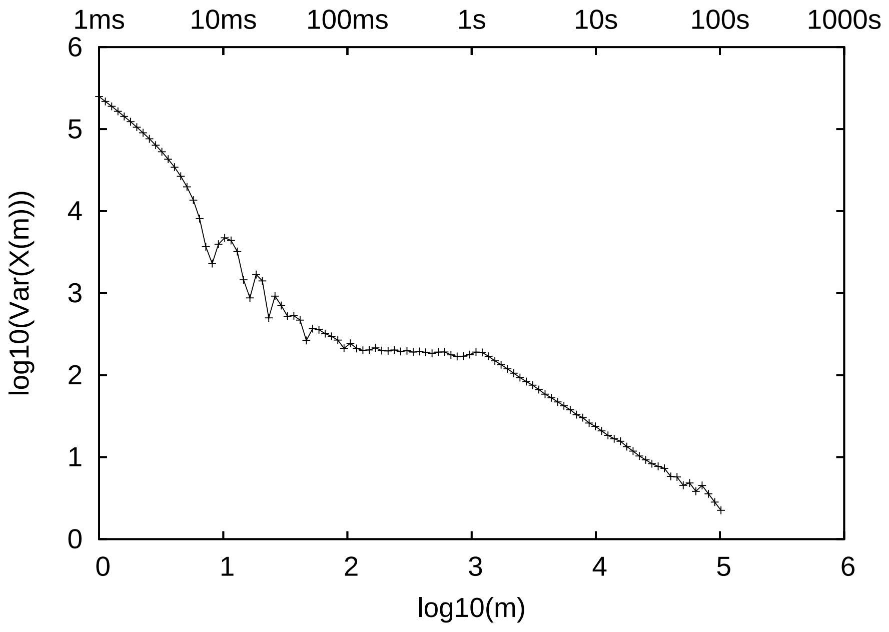


(b) DIST-RTSIM

Figure 4.7. Comparison of TCP-sink time-series for one flow

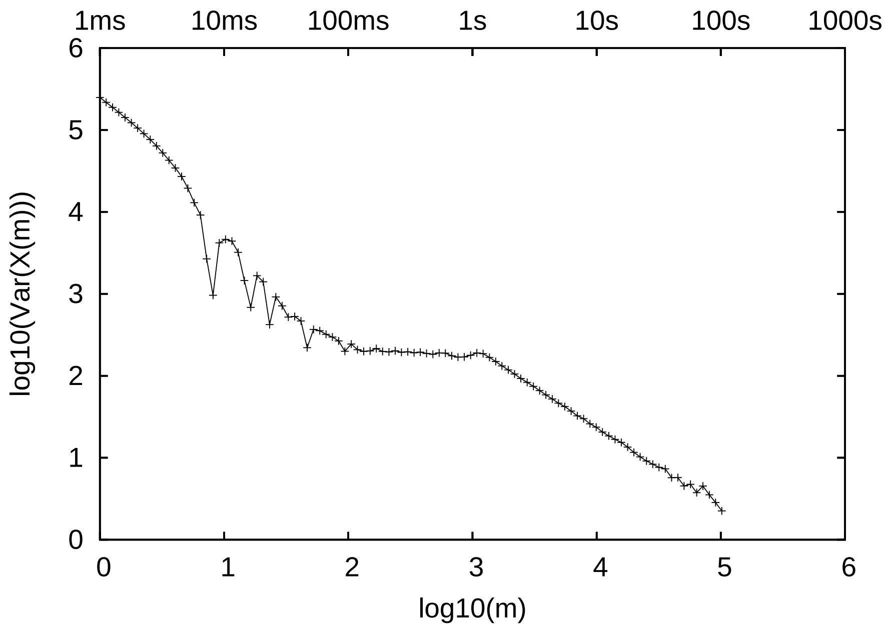


(a) DIST-RTSIM-60 (best, using `select()`)

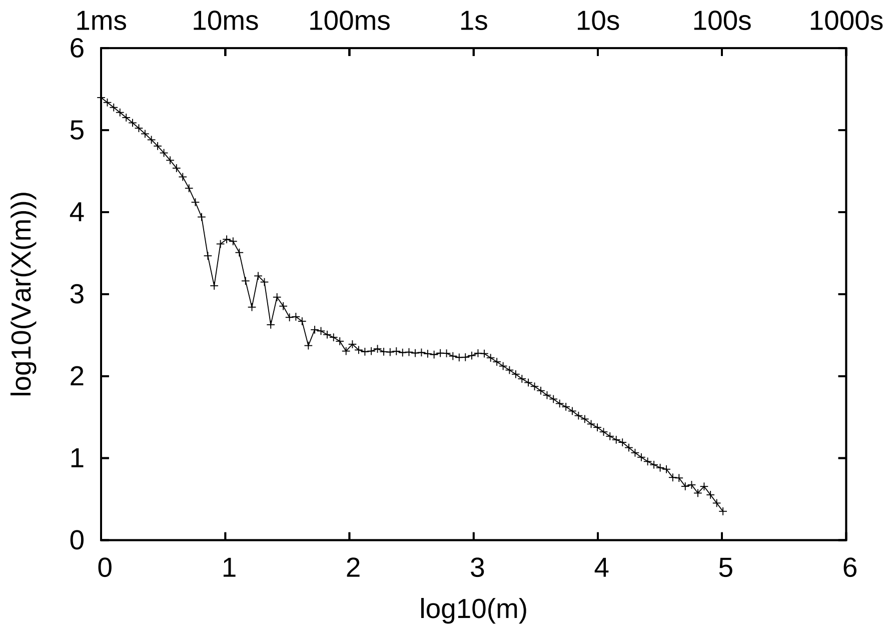


(b) DIST-RTSIM-60 (worst, using `select()`)

Figure 4.8. Comparison of TCP-sink time-series using `select()` when 60 flows are present

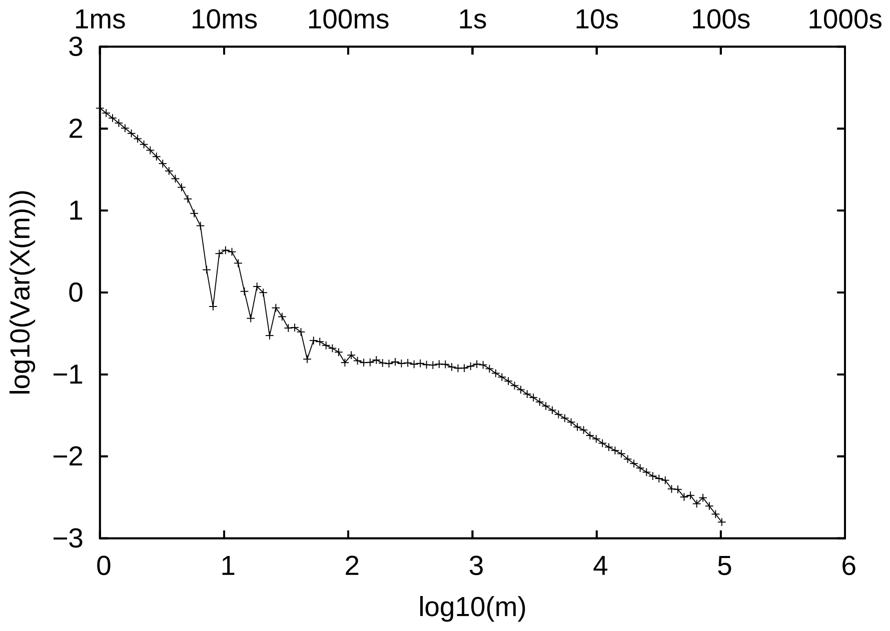


(a) DIST-RTSIM-60 (best, using `kqueue()`)

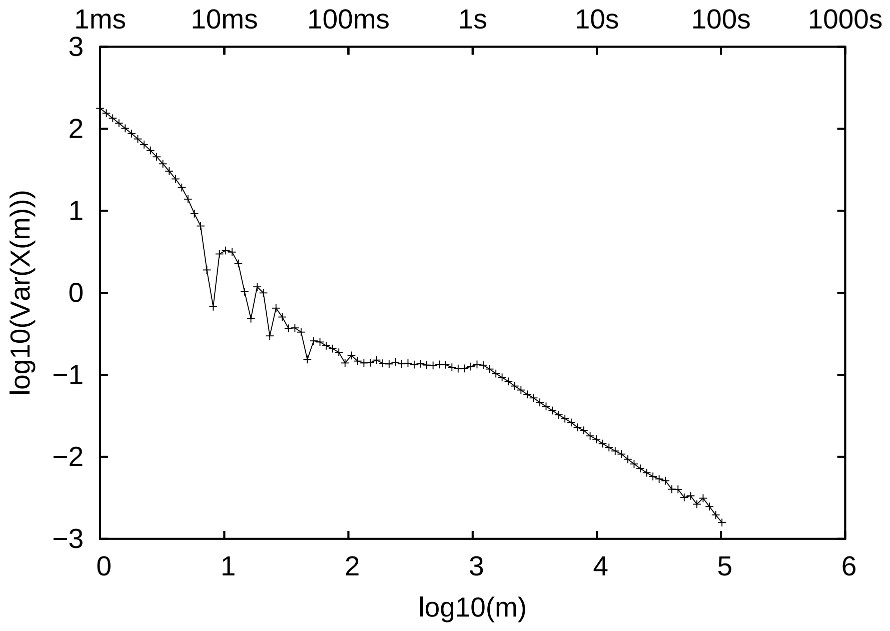


(b) DIST-RTSIM-60 (worst, using `kqueue()`)

Figure 4.9. Comparison of TCP-sink time-series using `kqueue()` when 60 flows are present

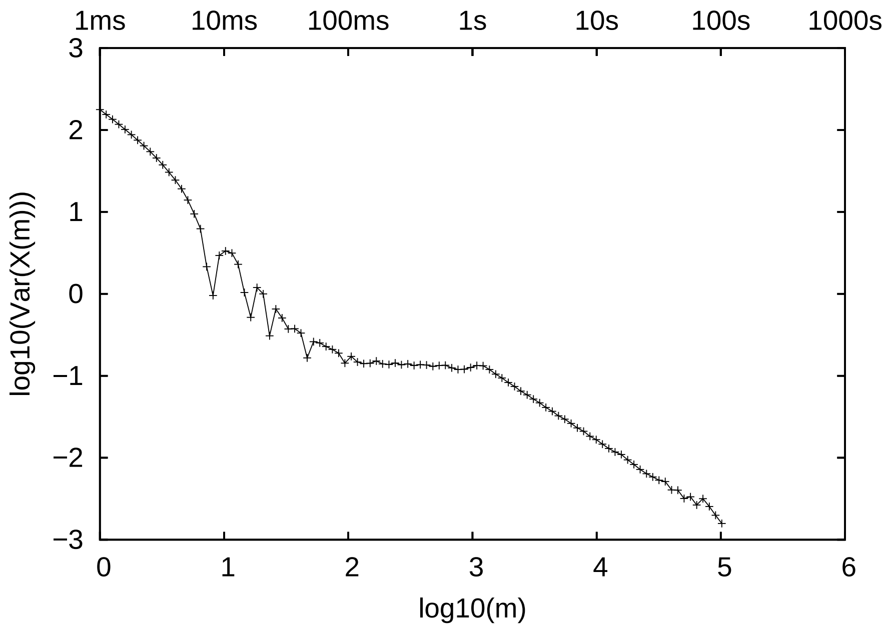


(a) RTSIM

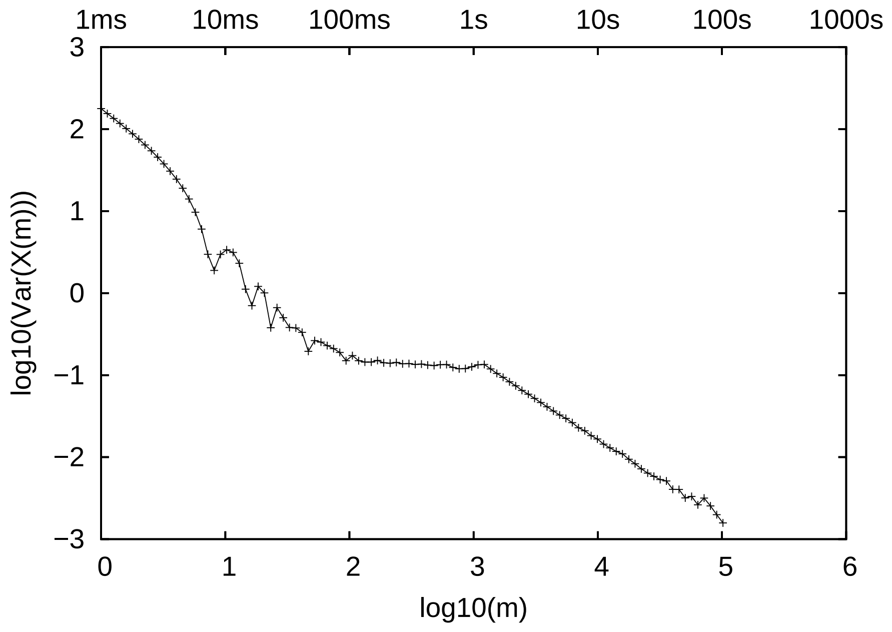


(b) DIST-RTSIM

Figure 4.10. Comparison of TCP-source time-series for one flow

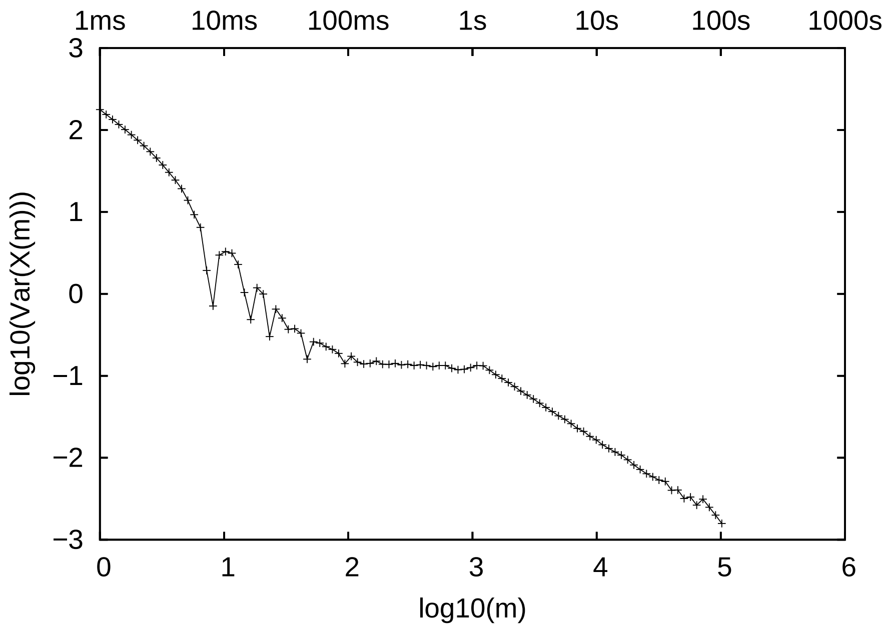


(a) DIST-RTSIM-60 (best, using `select()`)

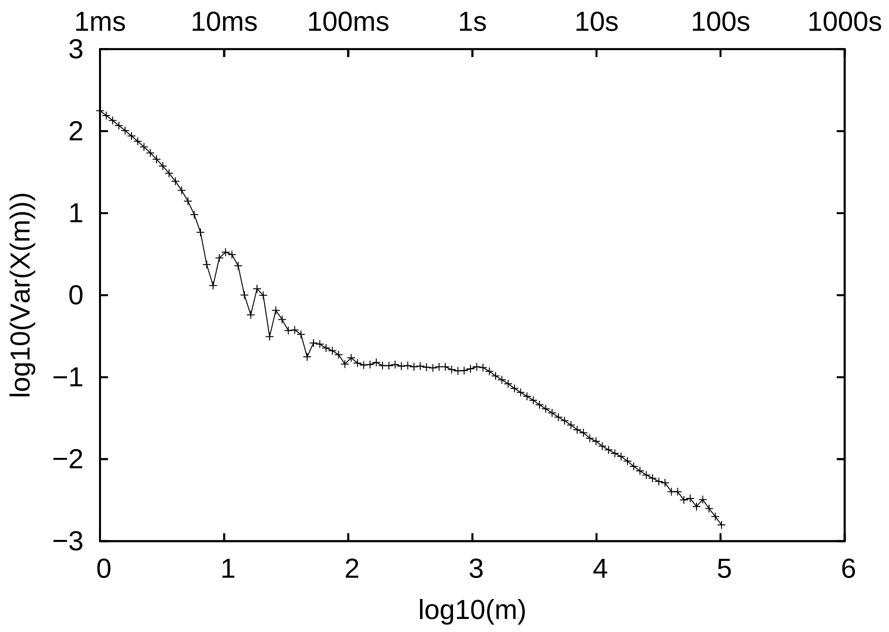


(b) DIST-RTSIM-60 (worst, using `select()`)

Figure 4.11. Comparison of TCP-source time-series using `select()` when 60 flows are present



(a) DIST-RTSIM-60 (best, using `kqueue()`)



(b) DIST-RTSIM-60 (worst, using `kqueue()`)

Figure 4.12. Comparison of TCP-source time-series using `kqueue()` when 60 flows are present

RTSIM-QUEUE denotes the experimental setup where both simulated nodes are in one instance of *nse*, i.e., pure simulation. *DIST-RTSIM-QUEUE* denotes the setup where the simulated nodes are in two instances of *nse* across a physical link.

In Figure 4.13, we show the aggregate queueing behavior, namely, instantaneous queue size sampled every 0.5 seconds, cumulative average queue size and packet drops in the last five seconds. At the aggregate level, the difference between RTSIM-QUEUE and DIST-RTSIM-QUEUE are small.

In Figure 4.14, we look at instantaneous queueing behavior for two individual TCP flows. RTSIM-QUEUE and DIST-RTSIM-QUEUE are noticeably different. Similarly, Figure 4.15 compares cumulative throughput for the same two individual TCP flows. Again, there is a noticeable difference between RTSIM-QUEUE and DIST-RTSIM-QUEUE. Note that TCP Flow #1 starts 0.5 seconds after Flow #2.

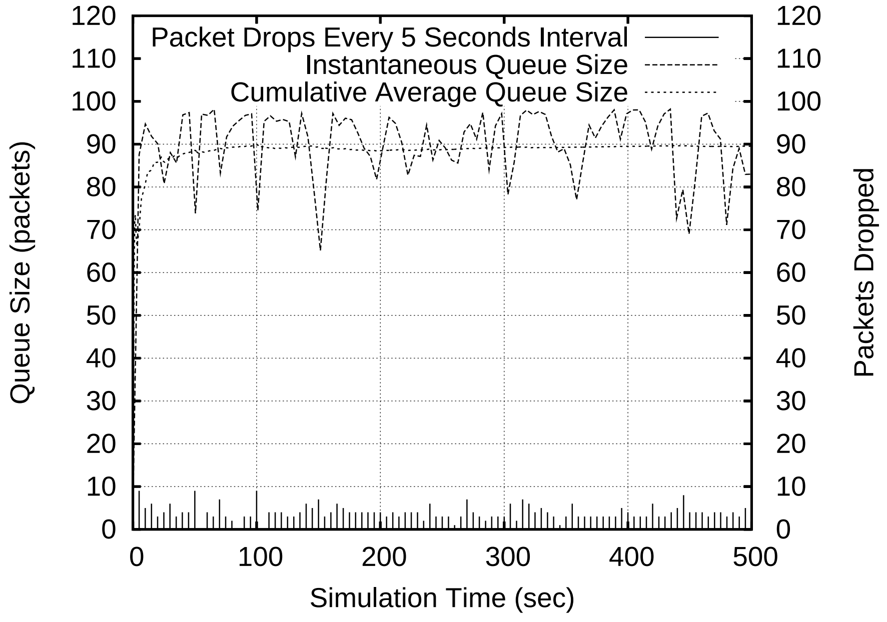
Thus, we can conclude that distributed *nse* significantly affects the individual queueing behavior of traffic flows when these flows are competing over the same link. Aggregate behavior is somewhat preserved.

4.3 Auto-adaptation of Simulated Resources

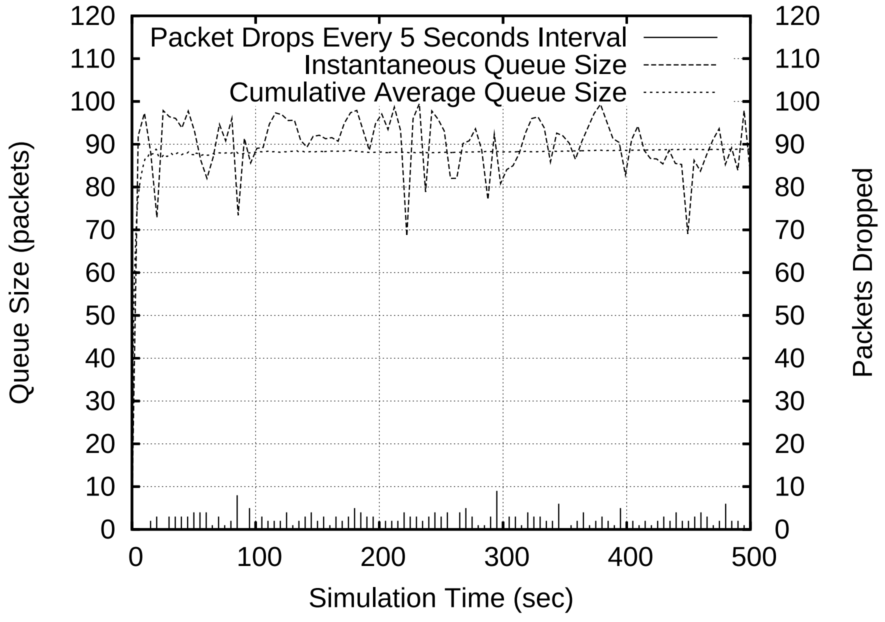
We evaluate the following auto-adaptation heuristics in this section, the details of which are discussed in section 3.4.

- Heuristic 1: Doubling vnode weights for vnodes mapped to overloaded pnode (s).
- Heuristic 2: Using packet-rate measurements per vnode during remapping.

The topology used for this experiment is illustrated in Figure 4.16. It is composed of 416 simulated vnodes and 436 links. It is composed of eight binary trees each containing 52 vnodes with the root of the trees connected to each other in a full mesh. We call the vnodes in a full-mesh *interior* nodes, other routers in the topology *border* nodes, and the vnodes on the edge *leaf* nodes. We have 200 leaf vnodes, 208 border vnodes and 8 interior vnodes in this topology. The access links (i.e., from leaf nodes) and the intermediate links are of 2Mbps bandwidth. The links in the full-mesh are made up of 10Mbps links.

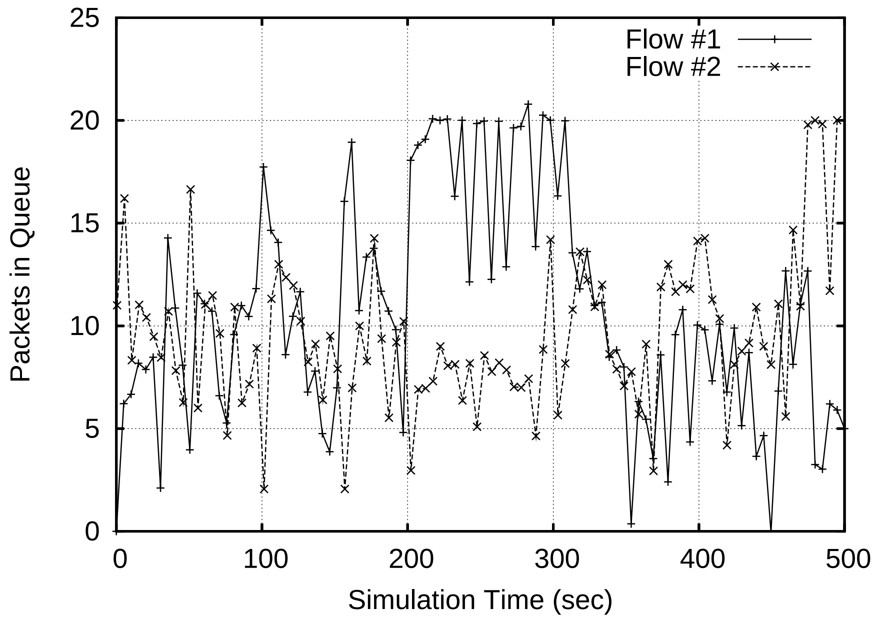


(a) RTSIM-QUEUE

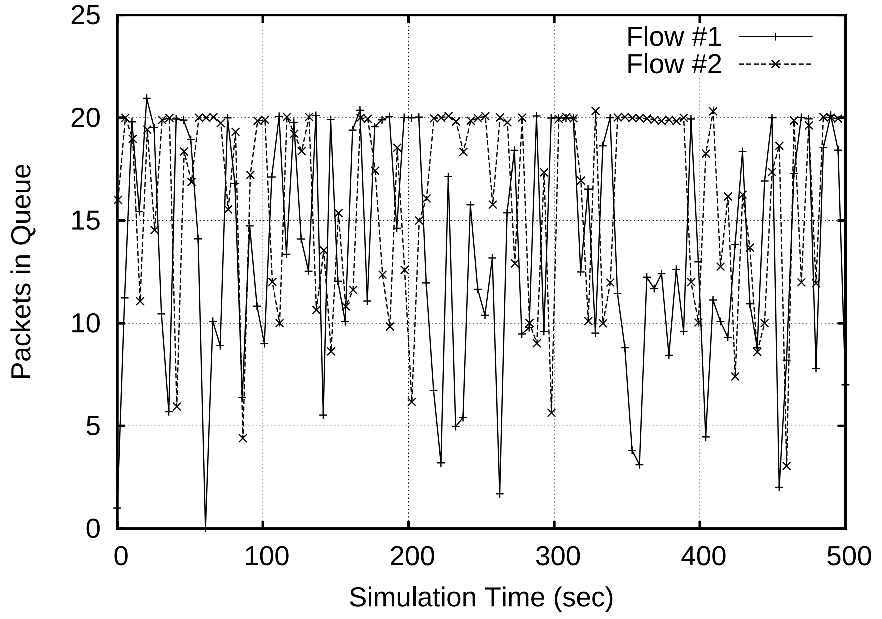


(b) DIST-RTSIM-QUEUE

Figure 4.13. Comparison of aggregate queueing behavior for six TCP and one UDP flows

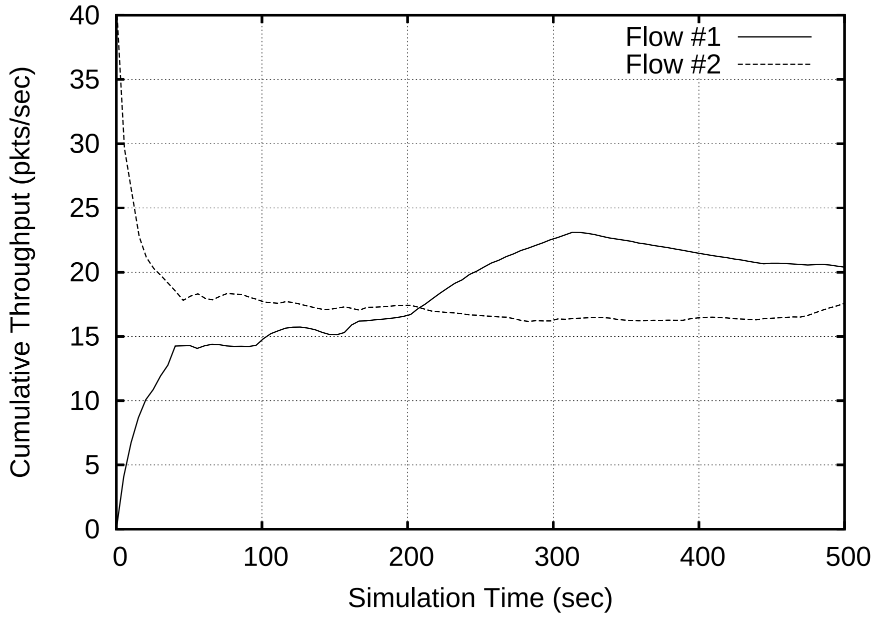


(a) RTSIM-QUEUE

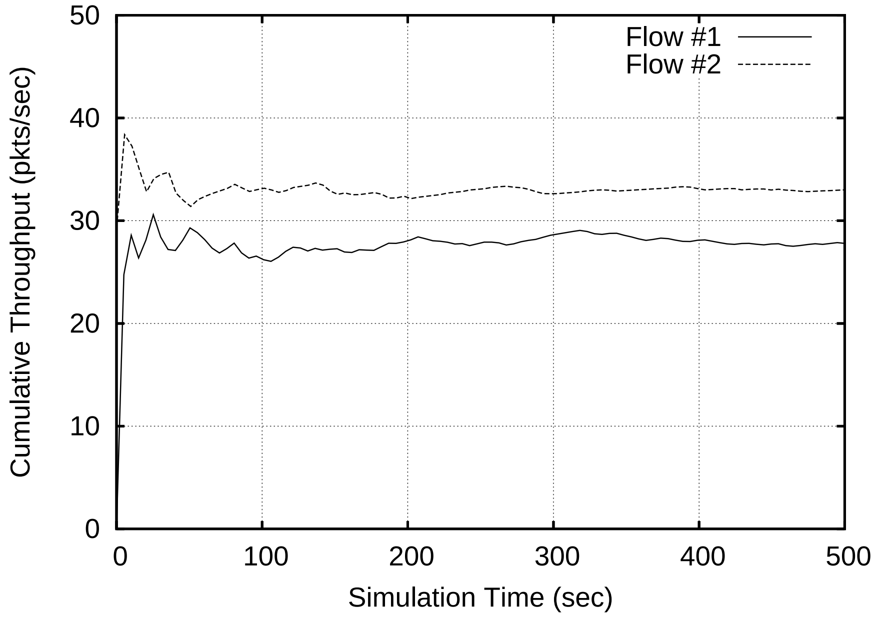


(b) DIST-RTSIM-QUEUE

Figure 4.14. Comparison of individual queueing behavior for two flows



(a) RTSIM-QUEUE



(b) DIST-RTSIM-QUEUE

Figure 4.15. Comparison of individual cumulative throughput for two flows

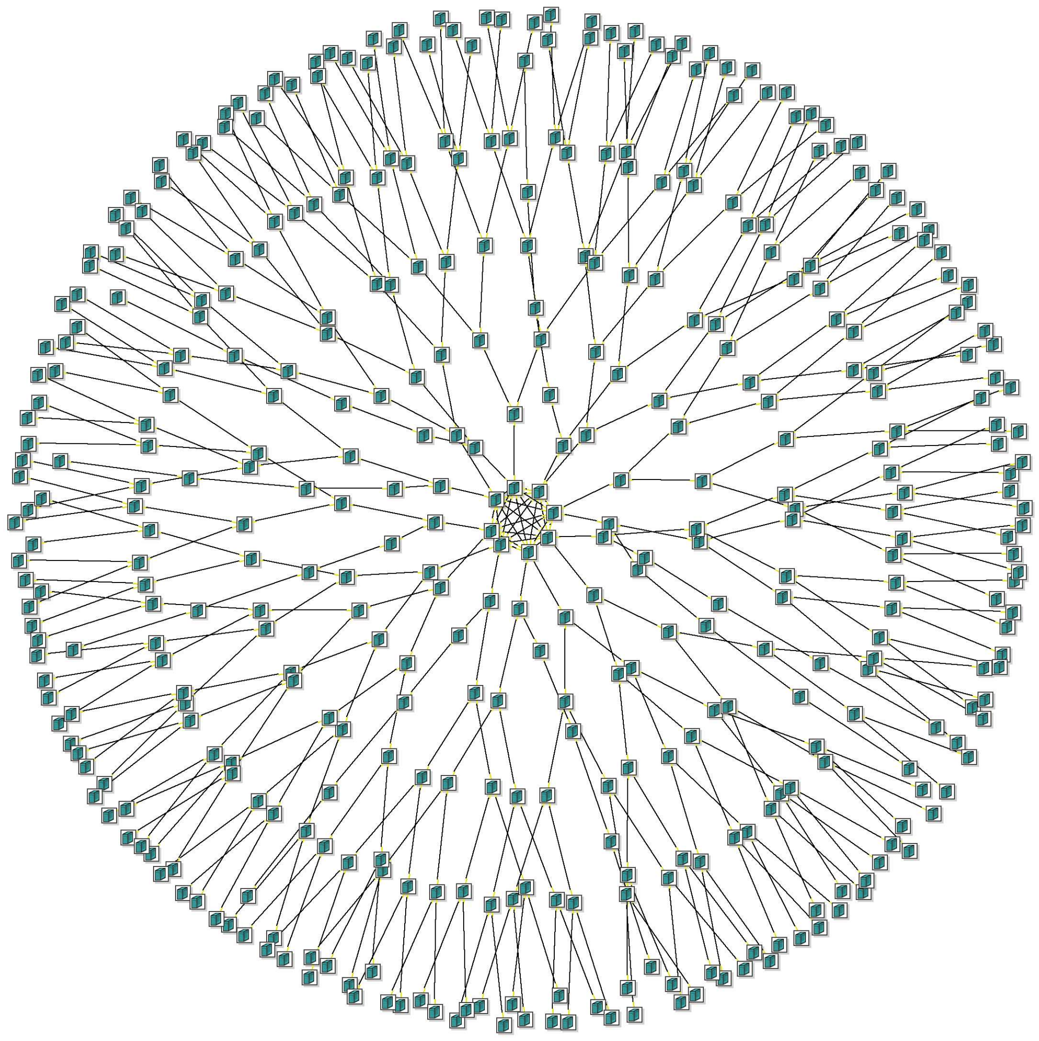


Figure 4.16. A 416 node topology used in evaluating auto-adaptation

Traffic is generated using *ns*'s `Agent/TCP` model. A total of 400 traffic flows are configured between leaf nodes out of which 200 of them are between pairs of vnodes in the same binary tree and 200 are between pairs of vnodes on different trees. We modeled the sending of 10,000 packets for each flow in pure simulation, i.e., in *ns*.

On an Emulab 850Mhz PC, the simulation is not able to run in real time. Tables 4.8 and 4.9 report information about running the above workload under *ns*. We report our measurements only after all the flows have completed sending the above number of bytes. Note that the wall-clock time reported is measured only from the start of simulation. It does not include the simulation configuration time. We also measured the aggregate packet rate passing through all the 416 simulated nodes in the topology to be 73,185 packets per second over 1310.79 seconds of simulation time. The peak packet rate numbers for the simulation could likely be higher than reported in the averages in Table 4.9.

In the evaluation of auto-adaptation, we do not introduce real PC vnodes in the topology since their presence simply increases the simulator traffic load and does not alter the methodology of auto-adaptation. On the other hand, because we only have simulated vnodes, it is possible to compare results with pure simulation.

We experimented with both 10ms and 100ms "slop factors". In the case of heuristic 1, a 10ms slop factor yielded a failed experiment. In other words, even for a one-to-one mapping, the slop was exceeded. With a slop factor of 100ms, we were able to get successful experiments for both heuristic 1 and 2.

In Table 4.10, we report the time taken for different phases of the experiment creation and swap-in process. Both heuristic 1 and 2 have approximately the same creation and swap-in times.

Table 4.8. "Pure" simulation data for the workload used in evaluating auto-adaptation

Simulation time (seconds)	Runtime (seconds)	Slowdown Ratio	Total Events processed	Events per runtime	Total simulated data transferred (MB)
1310.79	3018	2.30	165357063	54590	3814.7

Table 4.9. "Pure" simulation vnode packet rates for the workload used in evaluating auto-adaptation

vnode	Observed Packet Rate	
type	Lowest (Average Packets/s)	Highest (Average Packets/s)
Leaf	62	67
Border	68	895
Interior	1043	1068

Table 4.10. Auto-adaptation experiment creation and swap-in times

Experiment Creation Time	10 mins total 3 mins to run parser 7 mins for route computation
Experiment Swap-in Time	7–16 mins total 1–2 mins for mapping 20–40 seconds to generate OTcl sub-specification 1–4 mins *nse* startup time 2–9 mins for PC reboot/setup time 1 min for PC disk loading

4.3.1 Heuristic: Doubling Vnode Weights

For this evaluation, we set the co-locate factor to 512 letting `assign` map the above 416 node topology on to a small number of PCs. We choose 512 as it is a power of two. Since we double vnode weights, eventually a vnode whose weight is 512 will cause it to be mapped one-to-one. The actual co-locate factor that may be supported on Emulab PCs, if we only consider simulation memory use, is much higher. In that case, an experiment would take more mapping iterations for auto-adaptation. For this experiment, we also increase the *intra node bandwidth* to a large value in order to map the entire topology on one PC if possible and will likely cause run-time violation. Because of the randomized nature of mapping, we cannot ensure that two mappings are the same or similar in every respect. In Table 4.11, we report some characteristics of this auto-adaptation experiment.

Table 4.11. Results of an auto-adaptation experiment using heuristic 1

Total swap-in iterations	8
Number of PCs used in different iterations	$1 \rightarrow 2 \rightarrow 3 \rightarrow 6 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 12$
Maximum PCs reporting violation in an iteration	3
Iterations that had all PCs reporting violation	1
Maximum vnodes on a PC in the final iteration	10
Minimum vnodes on a PC in the final iteration	64
Weights for different types of vnodes in the final mapping	*Interior*: 64 *Border*: 8, 16, 32, 64 *Leaf*: 8, 16, 32, 64
Experiment Result	Final mapping has no “violations” at 100ms “slop” factor using 12 PCs

4.3.2 Heuristic: Vnode Packet Rate Measurements

We evaluate this heuristic by setting the packet rate capacity of PCs to 6000 packets per second so as to be well within the limits we reported in section 4.1.1. Packet-rates reported for any vnode in an iteration are used for the next mapping only if they are higher than previous iterations. In this manner, we take peak packet-rates in an iteration. The time used to compute the packet rate is the difference in time between the receipt of the last packet and the first packet by a vnode. In Table 4.12, we report some characteristics of this auto-adaptation experiment.

4.3.3 Summary of Auto-adaptation Experiments

The packet-rate measurements do not take into account short term bursts that may be present in the flow of packets. However, further study is required to determine if such bursts were present in our experiment and whether they were the cause of exceeding the 10ms slop factor. Relaxing the “slop factor” to 100ms provided us with a successful mapping after five iterations. Notice that the convergence using packet-rate measurements is much faster than doubling vnode-weights as we see in the mapping going from one PC

Table 4.12. Results of an auto-adaptation experiment using heuristic 2

Total swap-in iterations	5
Number of PCs used in different iterations	$1 \rightarrow 9 \rightarrow 10 \rightarrow 10 \rightarrow 10$
Maximum PCs reporting violation in an iteration	2
Iterations that had all PCs reporting violation	1
Maximum vnodes on a PC in the final iteration	75
Minimum vnodes on a PC in the final iteration	20
Packet-rates for different types of vnodes in the final mapping	*Interior*: 608–1041 *Border*: 53–555 *Leaf*: 45–145
Experiment Result	Final mapping has no “violations” at 100ms “slop” factor using 10 PCs

to nine PCs in a single iteration. However, the primary disadvantage of this method is that we need a fairly accurate measure of the packet rate capacity of running simulations on any PC. Overestimation of this capacity will cause this auto-adaptation heuristic to go into an infinite loop. In other words, a higher estimate of the pnode packet rate capacity than what can actually be supported will cause violations and remapping at the same vnode packet rates without a successful mapping.

Heuristic 1 is much better overall since we do not need to have much apriori information before mapping an arbitrary simulation workload and we are sure that the iterations will always terminate either with a successful mapping or a failed one. A combination of heuristic 1 and 2 may provide us with a better auto-adaptation solution. In other words, use heuristic 2 with coarse pnode packet rates that need not be accurate and then switch to heuristic 1 if the vnode packet rates do not change between successive iterations. However, we have not experimented with this method and leave it as future work.

CHAPTER 5

RELATED WORK

We can broadly classify the related work to this thesis into three categories: (a) Pure simulators that run in virtual time and strive to bring complete repeatability to real implementations or abstractions of networking entities, (b) network emulators that run in real-time and naturally forego fine grained repeatability to gain realism. (c) miscellaneous

5.1 Emulators

Dummynet [47] is a link emulator in the FreeBSD kernel that regulates bandwidth, latency, loss and some queuing behavior. This is used in Emulab for link emulation unless a simulated link is requested. ALTQ [13] is a queuing framework in BSD kernels that supports a large number of queuing disciplines.

Modelnet [55] is an emulation system focused on scalability. It uses a small gigabit cluster, running a much extended and optimized version of Dummynet which is able to emulate an impressively large number of moderate speed links. It has the added capability of optionally distilling the topology to trade accuracy for scalability. It is complementary to our work. Our work leverages *ns*'s rich variety of models and protocols.

5.2 Simulators

The x-sim [10] simulator provides an infrastructure to directly execute x-kernel [28] protocols in the simulator. It simulates links, routers and end-node protocols and provides logging mechanisms to capture the state of the simulation and post-process it.

pdns [46] is a parallel and distributed version of the *ns* simulator. It implements a conservative parallel simulation and has mechanisms for event distribution and simulation time synchronization. This system requires the experimenter to manually map a topology into its submodels that run on different processors and manually configure the global routing paths and IP addresses to run the simulation. This approach therefore is tedious, error-prone and can sometimes result in overall simulation slowdown compared to its serial version. The automated mapping that we develop is applicable to the mapping of a parallel simulation. Our work is in integrated experimentation whereas pdns is a pure simulation.

Dynamic Network Emulation Backplane [1] is an ongoing project that uses a dynamic library approach for capturing, synchronizing and rerouting network data from unmodified distributed applications over a simulated network. They also define an API for heterogenous simulators to exchange messages, synchronize simulation time and keep pace with real time in order to simulate a larger network, thus leveraging the strengths of different network simulators. Time-synchronization in the distributed simulation case has a high overhead and it remains to be seen whether it can be performed in real-time. Data are captured from unmodified applications by intercepting system call functions using a dynamic-library preloading approach. This however, is platform dependent as well as error prone due to duplication of code and specialized implementation of several system calls.

nsclick [38] embeds the click modular router [37] in ns-2, which allows a single click based protocol implementation to run over a simulated wired or wireless network as well as on a real node on a real network.

NCTUns [58] is a TCP/IP simulator that has descended from the Harvard network simulator [57]. This simulator virtualizes the OS's notion of time to be the simulation time. Using a combination of link simulation and tunnel devices, a network topology is built on a simulation host machine. Applications and the protocol stack are unmodified and therefore more realistic than traditional simulators.

umlsim [6, 54] extends user-mode Linux (UML) with an event-driven simulation engine and other instrumentation needed for deterministically controlling the flow of time as seen by the UML kernel and applications running under it. The current level of support allows network experimentation with a single link. The degree of multiplexing of virtual nodes is limited due to the use of a complete kernel. The current system also performs poorly and has much scope for improvement.

The Entrapid [27] protocol development environment virtualizes the kernel networking stack and user processes and moves them into user-mode. This enables building of network topologies with virtual nodes, links and application processes on a single machine.

5.3 Miscellaneous

The X-bone [52] is a software system that configures overlay networks. It is an implementation of the Virtual Internet Architecture [53] that defines “revisitation” allowing a single network component to emulate multiple virtual components, although in their context, a packet always leaves a physical node before returning on a different virtual link. In our system, multiple routing tables as well as the context of a virtual link are needed even when all the nodes and links of a virtual topology are hosted on one physical host. In the general sense, however, the issues they identify have a broader scope than what they have restricted themselves to, in their paper. Thus, virtual internets can be formed not just in the wide-area but also using a cluster such as the one employed in Emulab. Integrated network experimentation spans all three experimental techniques and we therefore believe that it is the most comprehensive form of virtual internet that we know of.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

Integrated network experimentation using simulation and emulation bridges the gap between each other to enable new validation techniques, such as traffic interaction, and performance comparisons of real protocol implementations against abstracted ones written for simulators. We have discussed many of the issues in realizing integrated network experimentation. We have solved some of them and provided an automated integrated experimentation environment in Emulab. By providing automation and programmability that is common in "pure" simulations to integrated experimentation with distributed real-time simulators, we enable "what if" experiments that were heretofore not practical to perform.

The results from section 4.2 show that in order to have packet flows in distributed real-time simulator instances be as similar to "pure" simulation as possible, experimental artifacts that occur due to OS interactions must be kept at a minimum. Using *time-variance* plots, we were able to conclude that the differences from "pure" simulation are only noticeable in the time-scale of the mean packet interarrival and do not change the behavior at large time-scales. Further research is required to make similar conclusions for a complex real-time simulation distributed over many PCs.

We explored two heuristics to guide auto-adaptation of mapping simulated resources on PCs in order for them to run in real-time. These heuristics utilize feedback data to repack simulated resources that were originally mapped in an optimistic manner. Although we are unable to make any conclusions about the quality of these heuristics due to failed experiments, we believe that this technique is attractive in general. Auto-adaptation is useful in other environments such as "virtual machine" based emulation experiments as well as mapping "pure" distributed simulation automatically to reduce overall simulation time. Faster remapping makes it practical to perform auto-adaptation experiments.

Overall, discrete-event packet-level simulation is computationally expensive. Better performance results than what we have presented in this thesis may be obtained by using modern PC hardware. It would also be interesting to see if recent techniques such as staged simulation [56] provide major improvements in performance and scaling of integrated experiments.

APPENDIX

SIMULATION CODE FOR THE COMPARISON OF QUEUEING BEHAVIOR

```
global ns fmon f1 f2
set ns [new Simulator]

$ns rtproto Static

#Open the processed trace file
set f2 [open packet-trace.dat w]

set n1 [$ns node]
set n2 [$ns node]

# bottleneck link
$ns duplex-link $n1 $n2 1.544Mb 25ms DropTail
$ns queue-limit $n1 $n2 100

# Used instead of trace-queue. NS tracing
# is disabled and a new custom class
# StorePktInfo keeps track of queue drop
# packets in memory
set sp [new StorePktInfo]
$sp store-pkts 10000
set dt [[$ns link $n1 $n2] set drophead_]
$dt target $sp

# Define a procedure that prints statistical data periodically
proc print_status {} {

    global ns fmon f2

    # Get current simulation time
    set curr_time [$ns now]

    # Get information from queue monitor
    set pdrop(0) [$fmon set pdrops_]
    set parri(0) [$fmon set parrivals_]
    set pdept(0) [$fmon set pdepartures_]
    set pcurr(0) [$fmon set pkts_]
    set pInt [$fmon get-pkts-integrator]
    set pqsize(0) [$pInt set sum_ ]

    puts -nonewline $f2 "$curr_time $pcurr(0) $pqsize(0) $parri(0) $pdept(0) $pdrop(0) "

    # Get information of a particular flow (by flow id)
    for {set j 1} {$j <= 2} {incr j} {

        # set a flow classifier
        set fcl [$fmon classifier];

        # select a particular flow
        set flow [$fcl lookup auto 0 0 $j]

        # get stats for this flow
        if { $flow != "" } then {
            set pdrop($j) [$flow set pdrops_]
            set parri($j) [$flow set parrivals_]
            set pdept($j) [$flow set pdepartures_]
            set pcurr($j) [$flow set pkts_]

            puts -nonewline $f2 "$pcurr($j) $parri($j) $pdept($j) $pdrop($j) "
        }
    }
    puts $f2 " "

    #Call this function again in future
    $ns at [expr $curr_time + 0.5] "print_status"
}
```

```
#Define a ’finish’ procedure
proc finish {} {
    # reference to global variables
    global ns f1 f2 sp

    #Flush the traces
    $ns flush-trace

    #Close the trace file
    close $f1
    close $f2

    $sp print-pkts
    #Exit simulator
    exit 0
}

for {set i 0} {$i < 6} {incr i} {
    set tcp($i) [new Agent/TCP/Newreno]
    $tcp($i) set fid_ [expr $i + 1]
    $ns attach-agent $n1 $tcp($i)

    set tcpsink($i) [new Agent/TCPSink]
    $ns attach-agent $n2 $tcpsink($i)
    $ns connect $tcp($i) $tcpsink($i)

    set ftp($i) [new Application/FTP]
    $ftp($i) attach-agent $tcp($i)
    if { $i == 0 } {
        $ns at 0.5 "$ftp($i) start"
    } else {
        $ns at 0.0 "$ftp($i) start"
    }
}

set udp0 [new Agent/UDP]
$udp0 set fid_ 100
$udp0 set packetSize_ 1440
$ns attach-agent $n1 $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 1440
$cbr0 set interval_ 0.04
$cbr0 attach-agent $udp0

set udpsink0 [new Agent/Null]
$ns attach-agent $n2 $udpsink0
$ns connect $udp0 $udpsink0
$ns at 0.0 "$cbr0 start"

set fmon [$ns makeflowmon Fid]
set dsample [new Samples]
$fmon set-delay-samples $dsample
set pktInt [new Integrator]
$fmon set-pkts-integrator $pktInt
$fmon reset
$pktInt set sum_ 0

set l0 [$ns link $n1 $n2]
$ns attach-fmon $l0 $fmon

$ns at 0.0 "print_status"

$ns at 1000.0 "finish"

$ns run
```

REFERENCES

[1] Dynamic network emulation backplane project. http://www.cc.gatech.edu/computing/pads/nms/.

[2] FreeBSD bug report: kqueue does not work with BPF when using BIOCIMMEDIATE or BIOCSRTIMEOUT. http://lists.freebsd.org/pipermail/freebsdbugs/2004-March/005856.html.

[3] Limitations in NS. http://www.isi.edu/nsnam/ns/nslimitations.html.

[4] J. S. Ahn and P. B. Danzig. Packet network simulation: speedup and accuracy versus timing granularity. *IEEE/ACM Transactions on Networking (TON)*, 4(5):743–757, 1996.

[5] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: Emulation and experiment. In *Proc. of SIGCOMM '95*, pages 185–195, Cambridge, MA, Aug. 1995.

[6] W. Almesberger. UML Simulator. In *Ottawa Linux Symposium*, 2003.

[7] S. Bajaj et al. Improving simulation for network research. Technical Report 99-702b, USC, March 1999.

[8] A. Boukerche and C. Tropper. A static partitioning and mapping algorithm for conservative parallel simulations. In *Proceedings of the Eighth Workshop on Parallel and Distributed Simulation*. ACM Press, 1994.

[9] R. Braden. Requirements for internet hosts – communication layers. Technical report, IETF, 1989. RFC 1122.

[10] L. S. Brakmo and L. L. Peterson. Experiences with network simulation. In *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, May 1996.

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

[12] R. Brown. Calendar queues: a fast 0(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[13] K. Cho. A framework for alternate queueing: Towards traffic management by pc-unix based routers. In *In Proceedings of USENIX 1998 Annual Technical Conference, New Orleans LA*, June 1998.

[14] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards realistic million-node internet simulations. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Jun. 1999.

[15] J. Cowie, D. Nicol, and A. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, 1(1):42 – 50, 1999.

[16] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[17] Elvin: Content based messaging. http://elvin.dstc..com/.

[18] K. Fall. Network emulation in the vint/ns simulator. In *Proc. of the 4th IEEE Symposium on Computers and Communications*, 1999.

[19] K. Fall and S. Floyd. Simulation-based comparisons of Tahoe, Reno and SACK TCP. In *Computer Communication Review*, volume 26, pages 5–21. ACM, July 1996.

[20] A. Feldmann, A. C. Gilbert, and W. Willinger. Data networks as cascades: Investigating the multifractal nature of internet wan traffic. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 42–55. ACM Press, 1998.

[21] R. M. Fujimoto. Parallel discrete event simulation. In *Communications of the ACM*, volume 33, pages 30–53, October 1990.

[22] S. Gadde, J. Chase, and A. Vahdat. Coarse-grained network simulation for wide-area distributed systems. In *Communication Networks and Distributed Systems Modeling and Simulation Conference 2002*.

[23] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. chan Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan. Effects of detail in wireless network simulation. In *Proc. of the SCS Multiconference on Distributed Simulation*, pages 3–11. USC/ISI, January 2001.

[24] J. Heidemann, K. Mills, and S. Kumar. Expanding confidence in network simulation. *IEEE Network Magazine*, 15(5):58–63, Sept./Oct. 2001.

[25] M. Hibler, L. Stoller, R. Ricci, J. L. Duerig, S. Guruprasad, T. Stack, and J. Lepreau. Virtualization Techniques for Network Experimentation. Technical report, University of Utah, May 2004.

[26] P. Huang, D. Estrin, and J. Heidemann. Enabling large-scale simulations: selective abstraction approach to the study of multicast protocols. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 241–248, Montreal, Canada, July 1998. IEEE.

[27] X. W. Huang, R. Sharma, and S. Keshav. The entrapid protocol development environment. In *Proc. of the 18th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM '99)*, volume 3, pages 1107–1115, 1999.

[28] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.

[29] D. W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, 1986.

[30] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *Proc. 2nd Intl. SANE Conference*, May 2000.

[31] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, 1998.

[32] K.-C. Lan. *Rapid Generation of Structural Model from Network Measurements*. PhD thesis, University of Southern California, 2003.

[33] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1):1–15, February 1994.

[34] J. Lemon. Kqueue – a generic and scalable event notification facility. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 141–153. USENIX Association, 2001.

[35] B. Liu, D. R. Figueiredo, Y. Guo, J. Kurose, and D. Towsley. A study of networks simulation efficiency: Fluid simulation vs. packet-level simulation. In *Proc. of the 20th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM '01)*, Apr. 2001.

[36] B. Liu, Y. Guo, J. Kurose, D. Towsley, and W. Gong. Fluid simulation of large scale network: Issues and tradeoffs. Technical Report UM-CS-1999-038, 1999.

[37] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. In *Symposium on Operating Systems Principles*, pages 217–231, 1999.

[38] M. Neufeld, A. Jain, and D. Grunwald. Nsclick: Bridging network simulation and deployment. In *Proc. of the 5th ACM International Workshop on Modeling, Analysis, and Simulation of Wireless and Mobile Systems (MSWiM '02)*, pages 74–81, Atlanta, GA, Sept. 2002.

[39] D. M. Nicol. Comparison of network simulators revisited. http://www.ssfnet.org/Exchange/gallery/dumbbell/dumbbell-performance-May02.pdf.

[40] D. M. Nicol. Scalability, locality, partitioning and synchronization in PDES. In *Proceedings of the Parallel and Distributed Simulation Conference (PADS'98)*, 1998. Banff, Canada.

[41] NIST Internetworking Technology Group. NIST Net home page. http://-www.antd.nist.gov/itg/nistnet/.

[42] B. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-based mobile network emulation. In *Proc. of SIGCOMM '97*, September 1997.

[43] A. Psztor and D. Veitch. PC based precision timing without GPS. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–10. ACM Press, 2002.

[44] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communication Review (CCR)*, 32(2):65–81, Apr. 2003.

[45] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM Computer Communications Review*, 2003.

[46] G. F. Riley, R. Fujimoto, and M. H. Ammar. A generic framework for parallelization of network simulations. In *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 1999.

[47] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Computer Communication Review*, 27(1):31–41, Jan. 1997.

[48] L. Rizzo. Dummynet and forward error correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, June 1998. USENIX Association.

[49] R. Scandariato and F. Risso. Advanced vpn support on freebsd systems. In *Proc. of the 2nd European BSD Conference*, 2002.

[50] T. Shepard and C. Partridge. When TCP starts up with Four packets into only 3 buffers. Internet Request For Comments RFC 2416, BBN Technologies, Sept. 1998.

[51] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. volume 32, pages 652–686. ACM Press, 1985.

[52] J. Touch and S. Hotz. The X-bone. In Third Global Internet Mini-Conference in conjunction with Globecom, Novemeber 1998.

[53] J. Touch, Y. shun Wang, L. Eggert, and G. G. Finn. A virtual internet architecture. Technical Report ISI-TR-2003-570, Information Sciences Institute, 2003.

[54] UML Simulator Project. http://umlsim.sourceforge.net/.

[55] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 271–284, Boston, MA, Dec. 2002.

[56] K. Walsh and E. G. Sirer. Staged Simulation: A General Technique for Improving Simulation Scale and Performance. In *ACM Transactions on Modeling and Computer Simulation (TOMACS), Special Issue on Scalable Network Modeling and Simulation*, April 2004.

[57] S. Y. Wang and H. T. Kung. A simple methodology for constructing extensible and high-fidelity tcp/ip network simulators. In *Proc. of the 18th Annual Joint Conf. of the IEEE Computer and Communications Societies (INFOCOM '99)*, volume 3, pages 1134–1143, 1999.

[58] S. Y. Wang and H. T. Kung. A new methodology for easily constructing extensible and high-fidelity tcp/ip network simulators. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 40(2):257–278, 2002.

[59] D. Wetherall and C.J.Linblad. Extending Tcl for Dynamic Object-Oriented Programming. In *Proceedings of of the Tck/Tk Workshop*, 1995.

[60] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

[61] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks (full report). http://www.cs.utah.edu/flux/papers/-emulabtr02.pdf, May 2002.