An Integrated Experimental Environment for Distributed Systems and Networks (full report)

Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad
Mac Newbold, Mike Hibler, Chad Barb, Abhijeet Joglekar

University of Utah

`www.cs.utah.edu/flux   www.emulab.net`

May 2002

Abstract

Today there exist three environments in which to perform experimental network and distributed systems research: network emulators, simulators, and live networks. The continued use of multiple approaches highlights both the importance and inadequacies of each. Emulab is an experimentation facility that seamlessly integrates these approaches. Emulab's primary goals are *ease of use*, *control*, and *realism*. Unlike the constituent experimental platforms it leverages, Emulab achieves these goals simultaneously.

Metric	Simul.	Emul.	Live Net	Emulab
Ease of Use	✓	ModelNet?		✓
Performance		✓	✓	✓
Repeatability	✓	✓		✓
Packet-Level Control	✓			✓
Coarse-Grain Control	✓	✓		✓
Scalability	varies	w/ModelNet	varies	✓
Param. Space Explor.	✓	ModelNet?		✓
Reuse of Models	✓	ModelNet?		✓
Real Links			✓	✓
Real Routers			✓	✓
Real Hosts		✓	✓	✓
Real Applications		✓	✓	✓
Real Users			✓	

Table 1: Characteristics of Experimental Platforms

1 Introduction

The diverse requirements of network and distributed systems research are not well met by any single experimental environment. Competing approaches remain popular because each covers a different point in a space defined by levels of *ease of use*, *control*, and *realism*. Discrete-event simulation, as exemplified by *ns* [42], and live network experimentation represent two extreme techniques. Emulation [3, 40, 33, 34] is a hybrid approach that subjects real hardware, protocols, and workloads to a synthetic network environment.

Emulab offers a complementary alternative to existing experimental environments. It is a large software system and set of tools, that when deployed on an appropriate cluster of machines, provides a time- and space-shared platform for research, education, or development in distributed systems and networks. Emulab's primary contribution is the seamless *integration* of the above seemingly disparate techniques in a manner that preserves the *control* and *ease of use* of simulation, without sacrificing the *realism* of emulation and live network experimentation.

Figure 1 enumerates characteristics of traditional approaches. Each offers unique benefits, thus guaranteeing their continued importance. For example, simulation presents a controlled, repeatable environment. However, its high level of abstraction may be inappropriate, for example, when studying the effects of interrupt-induced receiver livelock on a heavily-loaded system. Live networks achieve realism, but surrender repeatability and the ability to modify or even monitor internal router behavior. Single-node WAN emulators such as Dummynet [40] introduce artificial delays, losses, and bandwidth constraints to real applications in a controlled manner, but require tedious manual configuration.

Emulab spans simulation, emulation, and live network experimentation by integrating them into a common framework. This framework provides integrated abstractions, services, and namespaces common to all three environments, such as node and link allocation and naming, mapping them into domain-specific mechanisms and internal names. In this way Emulab masks much of the heterogeneity of the various approaches. Emulab subsumes the above techniques by automatically allocating simulated, emulated, or actual wide-area network links, thus obviating manual configuration. While Emulab provides most of the benefits of each individual technique, it is much more than a simple sum of services. Emulab's integration means that tools such as topology and traffic generators that were originally targeted for only one domain are often useable across all three. Furthermore, a particular experiment is not confined to a single experimental technique; it may include simulated, emulated, and wide-area resources.

Versions of this rapidly evolving system have been deployed at the University of Utah since April 2000, to provide a shared, Internet-accessible, research and education facility open to the community. Emulab's success has prompted other institutions to adopt it to build similar facilities, and federation is planned.

1.1 Design Principles

Emulab achieves integration through four primary design principles:

Transparency: An Emulab experiment can consist of simulated, emulated, and wide-area links and nodes. These realizations are essentially transparent to the user, since links and nodes share a common namespace and are consistently specified in *ns* syntax.

Virtualization: Emulab virtualizes IP addresses, hosts, and links. This level of indirection allows for control and configuration of these resources, as well as their efficient time-sharing. It also affords greater scalability by allowing the seamless multiplexing of virtual devices.

Automation: Experiment creation involves a large number of steps including, for example, configuration of network interfaces, routing tables, and switches, reloading disk images, exporting file trees, and configuring traffic generators. Emulab removes the tedium of manual configuration through automation. An integrated event system and the Turing-complete language, Tcl, that underlies the *ns* interface, help provide arbitrary programmatic control.

Efficiency: Emulab was designed to make efficient use of physical resources and to enhance experimenter productivity. It manages the shared use of physical resources to provide their greatest possible utilization, while ensuring inter-experiment isolation. Emulab performs experiment creation, swapping, and termination in a few minutes, enabling an interactive style of use.

2 Resources

As its name suggests, Emulab was originally conceived as an emulation platform. The flexibility of its design has helped it to grow from an emulated "Internet in a room" to a "slice of the Internet." It now supports a diverse set of physical node and link types. Nodes and links are virtualized in the sense that they may be allocated and controlled largely independently of their physical realization. Virtual nodes may be instantiated from a large set of local nodes, from a smaller set of distributed nodes, or within *ns* simulation. Distributed nodes are currently provided by the RON [6] testbed. Virtual links may map directly to local-area or wide-area links or may be emulated by Dummynet.

2.1 Virtual Nodes

Local Nodes: The Utah Emulab currently contains 168 PCs that can function as edge nodes, traffic generators, or routers. Each machine has five 100Mb Ethernet interfaces: one is on a dedicated control and data acquisition network, and the others are for arbitrary use by experiments. At each PC node, local memory and disk provide ample room for local computation and logging of monitoring data.

Distributed Nodes: By subsuming the RON testbed, Emulab now has 16 nodes at remote sites, including nodes connected via Internet2, DSL, and cable modems. Though their shared usage model and administrative policies limit Emulab's control over them, they support many of the key features of local nodes. For example, Emulab sets up accounts, provides convenient access via distribution of ssh keys, and automates traffic generation.

Simulated Nodes: Network simulation [11] has been widely used to rapidly design, evaluate, and validate new protocols as well as study existing protocol behavior. Though simulation often abstracts a good deal of detail [24, 19], it can provide scalability beyond the limits of physical resources. Virtual simulated nodes can be multiplexed on one physical node. Emulab integrates simulation through *ns*'s emulation facility, *nse* [17]. This allows simulated nodes, links, and traffic to be subjected to live application traffic.

2.2 Virtual Network Links

Local-Area Links: Emulab can easily exploit the large number of local nodes and wealth of available bandwidth to realize a switched LAN topology. Rapid and automated configuration of operating systems makes Emulab an attractive platform for kernel development and research within local-area networks.

All local nodes are connected using high-end switches, that function as a "programmable patch panel." To allow arbitrary topologies within Emulab and provide security to Emulab users, we employ Virtual LANs (VLANs). VLANs are a switch technology that restricts traffic generated within a VLAN to other machines in that VLAN. This technology can be used to define subnets within an experiment as well as to protect users from others' stray traffic. Separate switches are used for the control and experimental networks to provide isolation of control traffic from experiment-generated traffic.

Emulated Links: Emulab uses Dummynet [40] to emulate wide-area links within a local-area environment. A Dummynet node lies between two physical nodes and enforces queue and bandwidth limitations, introducing delays and packet loss. Dummynet nodes act as Ethernet bridges, so they are transparent to experimental traffic.

Wide-Area Links: If experimenters specify no particular links to wide-area nodes, they obtain the fully-interconnected "natural" Internet. However, if they specify such links, we set up IP tunnels so that distributed nodes can use "private" IP addresses, maintaining our principle of virtualization. In conjunction with our automated routing setup, an overlay network is automatically created to the experimenter's specifications. These tunnels also allow transparent communication between wide-area nodes and experimental interfaces on our local testbed nodes, so that

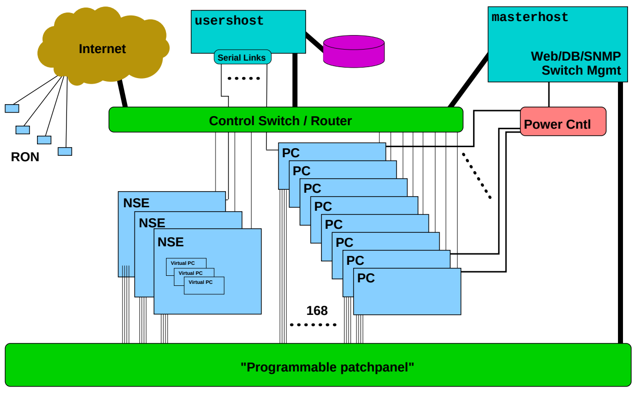

Figure 1: Emulab Architecture

networks can easily be constructed that contain both "live" Internet links and emulated Emulab links. This ensures that distributed nodes may be seamlessly treated as local nodes with respect to traffic generation, routes, and IP addresses.

Simulated Links: Emulab's deployment of *ns* makes a vast wealth of simulation infrastructure accessible to an emulated or distributed experiment. Emulab can leverage *ns*' rich and diverse protocol suite, varied statistical models, or support for wireless devices. Through its integration with Emulab, *nse* can be used to simulate a large-scale network within an emulation. For example, the NSWEB model [44] is considered to be a very accurate web workload model based on SURGE [9] that can be used to generate large number of web traffic flows. The close interaction between simulation and live protocols presents an opportunity to validate *ns*' abstractions.

2.3 Planned Extensions:

Though these physical realizations have proven successful, virtualization ensures Emulab is not bound to them. Plans are underway to incorporate additional resource types. We are constructing a WAN emulator based on the Intel IXP1200 network processor [28] that can more scalably implement congestion, route flapping, route asymmetry, router queuing delays, and packet dropping policies. Secondly, we plan to incorporate the powerful ModelNet [43] network emulation platform, which should offer greater scalability for wide-area flows. Such integration offers ModelNet's benefits, automatically controlled and configured through Emulab's existing interfaces.

3 Experiment Life Cycle

An experiment is Emulab's central operational entity. It represents a network configuration, including switch VLAN mappings and path characteristics; node state, including operating system images; and database entries, including event traces and traffic generators to be instantiated on nodes. The intended duration of an experiment ranges from a few minutes to many days. Emulab places a premium on efficient experiment creation and termination so that these latencies are not a barrier to interactive experimentation. When interaction is not required, Emulab can fully automate the process by scheduling and executing batch experiments in the background as resources permit.

As we proceed, we develop an analogy between an experiment and a Unix process. This metaphor illustrates the life cycle of an experiment and Emulab's role in automating and controlling the procedure. Emulab compiles an *ns* specification to synthesize a hardware realization of the virtual topology. The specification is first parsed into an intermediate representation that is stored in a database and later allocated and "loaded" onto hardware. During experiment execution, Emulab provides interfaces and tools for experiment control and interaction. Finally, Emulab may preempt and "swap out" an experiment.

3.1 Accessing Emulab

Emulab employs a small set of administrative nodes to provide a secure interface, as depicted in Figure 1. `masterhost` is a secure server for many of our critical systems, including the web server, database, and switch management.

To minimize administrative overhead, Emulab employs a hierarchical structure for authorization. To begin a new *project*, a "leader," e.g., a faculty member or senior student, submits information through a straightforward web interface. Once the project has been approved by Emulab staff, authority and accountability is delegated to the project leader.

The web interface provides a universally-accessible portal to Emulab. Needing only a standard web browser, an experimenter may create or terminate an experiment, view the corresponding virtual topology, or configure various node properties. The simplicity of this interface ensures that neither manual configuration nor bureaucratic delays are a barrier to experimentation.

Having created an experiment, experimenters may log directly into their allocated nodes or may log in to `usershost`, which serves as a centralized point of control. This node is currently also `fileserver`, which exports home and project directories across an experiment and stores operating system images.

3.2 Specification

Just as program text is the concrete specification of a runtime process, an *ns* script written in Tcl configures an Emulab experiment. This choice facilitates validation and comparison since *ns*-specified topologies and traffic generation can be seamlessly reproduced in an emulated or wide-area environment. For the large community of researchers well-versed in *ns*, it provides a graceful transition from simulation and an opportunity to leverage existing scripts. Since Tcl is a general-purpose programming language, a researcher is empowered with looping constructs, conditionals, and arbitrary functions to drive experiment configuration and execution.

```
set ns [new Simulator] ;# Create the simulator
source tb_compat.tcl   ;# Add Emulab commands
$ns rtproto Static     ;# Automatic routing

set source [$ns node] ;# define new nodes
set router [$ns node]
set dest   [$ns node]

# Connect source to router and router to dest
$ns duplex-link $source $router 10Mb 0ms RED
$ns duplex-link $router $dest 1.5Mb 20ms DropTail

tb-set-node-os  $source FBSD45-STD ;# Set OS
tb-set-hardware $dest pcinet  ;# Request wide-area node

$ns run   ;# "run" on Emulab
```

Figure 2: A linear topology with routing and a wide-area node

Emulated nodes and links enjoy full implementation transparency. By default, *ns* syntax used to specify nodes and links is interpreted by a parser that configures interposed Dummynet nodes to effect a virtual topology. To instead incorporate distributed nodes, an experimenter need only specify a corresponding node type, as shown in Figure 2. A simulated topology can be embedded within a physical topology by wrapping standard *ns* syntax in a `make-simulated` block.

Emulab's automation extends beyond virtual topology configuration to encompass dynamic aspects of an experiment, including traffic generation. Cross traffic is important for studying the behavior of protocols in the face of congestion. Any constant bit rate (CBR) traffic flow identified via standard *ns* syntax automatically instantiates traffic sources and sinks using the TG Tool Set [31]. Simulated FTP and Telnet flows are rendered using *ns*' emulation facility, *nse*. This mechanism injects traffic generated by models, such as the tcplib telnet distribution, into a live network.

Program objects allow arbitrary applications to be attached to an *ns* node. A program object may be started and stopped at any point during an experimental run. Though outside standard *ns* syntax, this mechanism greatly simplifies remote execution.

Experimenters unfamiliar with *ns* syntax may create topologies graphically via the "NetBuild" Java GUI. This, in turn, automatically generates an *ns* configuration file. Alternatively, a standard topology generator such as GT-ITM or BRITE may be used to generate an *ns* script that is subsequently passed to Emulab. This highlights one of the primary benefits of integration: application of tools intended for one experimental domain, in this case simulation, to another. Experiment creation is the only step requiring experimenter intervention; all subsequent phases are handled by Emulab.

3.3 Parsing

To realize an experimental configuration, Emulab uses a compiler that accepts *ns* as a source language. Portability goals motivate the componentizing of a traditional compiler into front and back ends. While a traditional compiler only targets one architecture within a given run, Emulab must target multiple, heterogeneous physical resources simultaneously. For example, a single experiment may incorporate simulated, emulated, and wide-area links.

Front-end compilation is performed by a Tcl/*ns* parser. The parser recognizes the subset of *ns* relevant to topology and traffic generation. Written in Tcl, it operates by overriding and interposing on standard *ns* procedures. Emulab executes the script in the context of these new definitions. As such, the script enjoys the full complement of Tcl's features and syntax. Unrecognized *ns* commands are ignored, while *ns* syntax configuring links and traffic sources and sinks triggers the overloaded procedures.

An Emulab-specific library defines procedures for controlling aspects outside of *ns*' domain, including configuring a node's operating system and specifying its hardware type. These procedures are not required, since Emulab supplies default values in their absence. A stub library defines null procedures so that the same script may be executed both on Emulab and within *ns*.

Both overloaded and Emulab-specific procedures populate the database. This relational database stores information about hardware, users, and experiments. The database, in part functioning like a compiler's intermediate representation (IR), presents a consistent abstraction of heterogeneous resources to higher layers of Emulab and to experimenters. For example, the front-end database representation of distributed and emulated nodes differ only in a type tag. The database provides a single name space for all experimental entities. Thus, in most cases, experimenters can interact with them using the same commands, tools, and naming conventions regardless of their implementation. As an example, nodes of any type may host traffic generators, despite the fact that the traffic may flow over links simulated by *ns*, emulated by Dummynet, or provided by a distributed testbed.

3.4 Global Resource Allocation

The global resource allocation phase is responsible for binding abstractions created during previous stages to physical entities. It corresponds to the resource allocation performed during back-end compilation and the name binding done during linking.

A simulated annealing algorithm, `assign`, maps a target configuration onto physical resources. The algorithm queries the database to obtain the intermediate representation of the target configuration as well as the set of available nodes, switches, and links. On the current hardware base `assign` finishes in less than 15 seconds. If the (randomized) algorithm fails to find a solution on the first run, it is repeated a number of times. If still unsuccessful, the failure is reported to the user, any residual experimental state is reaped.

Based on the output of `assign`, Emulab reserves nodes and links and updates the database with the resource mappings, user-specified node names, delay, bandwidth, and loss parameters, and operating system image. To exploit parallelism, Emulab employs optimistic concurrency measures. Therefore, reservations may fail due to conflicts caused by race conditions. If a reservation fails, `assign` is again invoked and the process is repeated until reaching successful allocation or a threshold number of failures.

To ensure consistent naming across instantiations of an *ns* configuration, Emulab virtualizes IP address and host names. This level of indirection is necessary since a configuration is unlikely to be mapped to the same physical resources upon re-creation. While experimenters are free to manually assign IP addresses, this task is most often left to Emulab. Despite using a randomized algorithm to assign actual physical nodes and links, Emulab deterministically names nodes and links across experiment creations.

3.5 Local Self-Configuration

Emulab must configure resources after their assignment to physical hardware. This is driven by the nodes themselves, but entirely controlled by state stored centrally in the database. We have found that managing *node state* is one of the most crucial aspects in meeting our goals. For robustness and security reasons we keep the nodes free of persistent configuration state. At boot time they are in their only fully-known state, and at that time configure themselves out of the database.

Emulab ensures that a clean disk image is installed on every node before experiment swapin or creation; this presented a significant performance challenge. The conflicting policy and performance requirements of disk loading are described later, in section 4.2.

In parallel, cooperative nodes are issued a Unix `reboot` command via `ssh`; any nodes that fail to reboot in a timely manner are power cycled. The PCs' BIOS are configured to have their network cards use Intel's PXE [39] bootstrap protocol. Each node's PXE ROM contacts `masterhost`, loading a first level kernel as directed by the database. This first level kernel might be the fast disk image loader outlined above, a memory based operating system, or typically, a larger second level bootstrap program. This second level loader again contacts the database to determine the next step, either booting from an on-disk partition or downloading an OSKit [21] kernel. This multiphase approach permits flexible configuration and customization of the OS that runs on each node in an experiment. The system then waits for the nodes to come back up. If a node does not come alive in a timely manner it is assigned to the system pseudo-experiment "down" for later manual examination.

The default installations of FreeBSD and Linux have slightly modified initialization sequences that invoke a node configuration script, called the Testbed Master Control Client, TMCC. By communicating with a daemon that fronts the database, it obtains the information required to configure interfaces, host names, Dummynet delays, users, groups, RPM packages, and startup scripts to be run. It NFS-mounts the specified project's tree and users' home directories from `fileserver`.

Through the TMC daemon, a primitive node synchronization mechanism is available to user scripts. Each node has an associated "ready" flag in the database. Nodes can declare themselves ready and retrieve the count of ready nodes and the total number of nodes in their experiment.

3.6 Experiment Control

Traditional operating systems offer simple job control over local processes, including an ability to start, stop, and resume execution. Distributed operating systems and batch queue systems extend this mechanism to apply across a network. To provide complete accessibility to remote experimenters and zero penalty for remote use, Emulab is faced with a similar challenge. Like these systems, and like simulators, Emulab must control distributed processes such as traffic generators running throughout an experiment. To this end, Emulab employs a distributed event system. Unlike the above systems, Emulab must expose hardware resources to remote users. Through virtualization, Emulab provides distributed access to local resources, such as serial lines, and extends control mechanisms to links.

If virtualization imposed a common, high-level interface to physical resources, Emulab would limit an experimenter's power and expressiveness over these entities. Emulab provides abstractions via tools and *ns*' high-level syntax. However, experiments are not restricted to such interfaces. Such unchecked power allows an experimenter to unwittingly corrupt their assigned resources. Emulab's ability to restore an experiment's state from the database and reload disk images from a repository protect the experimenter from accidents.

Emulab currently supports an event system that allows users and programs to remotely control activity on the nodes of a testbed. Our control infrastructure uses events for activities such as executing programs or scripts, notifying device drivers of changes to characteristics of the simulated network, and tracking important system events (e.g., node or link failures).

One aspect of control that we have found to be highly valued by experimenters is "root" access, or superuser privileges. While this does increase the need for high security to protect experimenters from each other and protect the world from their experiments we have found root access to be critical. Emulab is one of the only facilities where researchers can reasonably obtain root access on remote nodes.

Every node in Emulab is connected to a control network, separate and isolated from the network that is used for experimental traffic. This provides three important virtues: more reliable control, cleaner experimental data, and greater security.

All nodes are connected to serial lines for console interaction and power controllers so that they can be rebooted by experimenters, even if they crash or get "wedged." Unless their program needs to use a display or mouse, Emulab does not penalize remote experimenters—with only minor exceptions, remote users have as much control over nodes as they would with local machines, and in many ways, they have more, since our tools provide easy access to these facilities. For example, console lines are virtualized so that an experimenter need not be logged into the host where the serial line is attached; all consoles are securely available from any Unix machine.

3.7 Preemption

Traditional operating systems preempt and schedule processes for better system throughput and CPU utilization. Because Emulab is a shared facility, efficient utilization is also a priority. Therefore, it supports the ability to "swap out" and later re-instantiate an experiment. The time scales of scheduling quanta and process context switch times ensure that Unix preemption can generally occur without upsetting interactive use. Preemption in Emulab is complicated since the larger time scales inherent in experiment setup and teardown can not occur without noticeable disruption to experiments and results. This additional constraint means that Emulab does not enjoy a Unix scheduler's freedom to arbitrarily preempt processes. Instead, we have designed a facility to detect idle periods before swapping out an experiment.

Emulab nodes are often under-utilized despite being assigned to experiments. Although commonly due to negligence, users are reluctant to relinquish nodes assigned to an idle experiment. Determining idleness in Emulab is difficult; the indicators used in standard clusters are not sensitive enough, since an active experiment may be doing nothing sending a single network probe every 5 minutes. Indicators we monitor include activity in the experimental network, use of pseudo-terminal devices (indicating interaction with the node), and CPU load averages. Our idle detection system gathers data on these three aspects of activity. Eventually we will use this system to drive automatic swapout of idle experiments.

During "swap out", Emulab stores the virtual topology, host names, and general setup of an experiment in the database. "Swap in" reconstitutes this state on physical resources after invoking `assign`. Since nodes currently retain neither their disk nor their memory state, an NFS file system is used for persistence.

4 Issues

Here we explore a number of issues which were particularly important in achieving Emulab's goals.

4.1 Mapping of Virtual to Physical Resources

4.1.1 Mapping Local Resources

In a testbed of appreciable size and finite inter-node bandwidth (i.e., a practical and economical one), we must assign the user's virtual nodes and links to their physical counterparts. That is, Emulab ensures the physical hardware will support the emulated traffic flows without introducing any bottlenecks, with their attendant experimental artifacts. As a trivial example, consider the physical network on the left in Figure 3. If we wish to emulate the virtual topology shown to its right, then we must pick a physical realization which groups A, B, and C together on one switch, and D, E, and F on the other switch; any other configuration will attempt to send excess bandwidth across the inter-switch link.

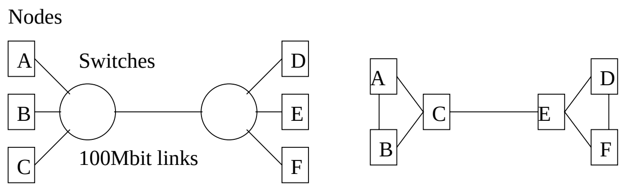

Figure 3: A trivial six-node partitioning problem

We call determining the relationship between these two networks the *testbed mapping problem*. This problem is trivial in the six-node example, but in the general case, it is NP-hard (by reduction to the multiway separator problem, or the minimum-degree graph partitioning problem [22]).

To maximize the utility of the testbed, we should create an assignment in interactive time; in the same way that researchers use the *ns* simulator, interactive use of the testbed creates a qualitative difference in the user's experience, not just a quantitative one. Interactive use encourages "what if" experiments and rapid adaptation to results, rather than the slow tedium of a batch-processed system.

Our testbed mapper, called `assign`, searches for an optimal assignment of virtual nodes to physical nodes. It attempts to minimize the bandwidth used between switches, and to minimize the total number of switches affected by one user's virtual network, so as to retain the largest possible degree of usability for other concurrent users of the testbed. Specifying *features* allows users to obtain particular processor types, link types, or nodes last used by the same project. We examine the performance of `assign` in Section 5.2.

4.1.2 Mapping Wide-Area Resources

The designer of an experimental topology which includes distributed nodes may assign specific desired bandwidth and latency characteristics between these nodes. When this experiment gets mapped to physical resources, the experimenter expects that the resulting physical topology will resemble the one requested as closely as possible. However, unlike the highly configurable connections inside Emulab, connections between distributed nodes traverse the Internet through uncontrolled links. Fortunately, on the Internet, n physical nodes (ideally) offer $n * (n - 1)$ links between them—each one with its own, probably unique, characteristics. Because of this, an ideal mapping of experiment nodes onto physical nodes can be reasonably close to experiment specifications.

However, finding this ideal mapping is a far from simple task. For non-trivial experiments, an exhaustive search is out of the question. To provide a best-effort solution in a short amount of time, we implemented a genetic algorithm.

Solutions deemed "fit" for purposes of the genetic algorithm had a low (sum of roots) latency error, a low (sum of logs of ratios) bandwidth error, and avoided the assignment of multiple experiment nodes onto the same physical node. The experimenter may decide to base the mapping on long-term observed physical characteristics of the network, or on recently gathered statistics that better reflect the sometimes dynamic nature of the Internet. The algorithm completes its task of mapping experiments to our current distributed node set in a few seconds at most. Performance of this system is discussed in further detail in Section 5.2.

4.2 Disk Reloading

An important feature of testbed control is the ability to reload disks automatically. Disk reloading is the most reliable and efficient way to install a new OS image.

Policy: In the policy for disk reloading we see a tension between the latency of typical experiment creation, overall Emulab throughput, Emulab system complexity, node robustness, and experiments' security. Our policies have evolved over time, driven by our tools, pressure on resources, and experience.

To ensure a consistent, safe, and secure OS installation, each node in a new experiment requires that a clean disk image be loaded, sometime after the previous experiment terminated. However, disk reloading remains one of the most time-consuming aspects of experiment creation and swapin, although over time we have reduced it to less than 100 seconds. Currently, Emulab's policy is that upon experiment termination, each node's disk is reloaded with the default image containing both FreeBSD and Linux. This works well since most users request one of these OS's, and if there are sufficient free nodes, the disks are reloaded in the background, ready to go for the next swapin.

A troubling effect occurs, however, in the common case of a single experimenter creating and tearing down very similar experiments, in quick succession; this frequently also happens with the batch queue. The nodes are not available for the few (typically wasted) minutes while reloading, which is exactly when the user requests a similar number of nodes for their next experiment. In the past, to avoid this anomaly we delayed reloading for 100 seconds, reallocating the unreloaded node only to an experiment in the same project as the previous experiment. However, that has robustness vulnerabilities.

Users can specify an alternative disk image or partition, including a user-provided image, holding an arbitrary OS. In that case, the background disk reloading was entirely wasted. Automated analysis of historical and ongoing experiment creation and swap patterns would be one way to attack this challenge.

Process: The procedure for disk reloading follows the initial steps described in Section 3.5: the PXE BIOS loads the initial bootstrap which in turn loads a small, memory-based FreeBSD system used to run the disk loader client. This client contacts an instance of the disk loader server, downloading, uncompressing and writing out the disk image. After completion, the node reboots from the newly installed image.

We currently provide a small set of images containing various versions of Linux and FreeBSD, but in the future will likely provide others, including Windows 2000. Custom disk images can be used to boot an unsupported OS, to load a newer (or older) version of a supported OS, or to install a specialized version of an existing image on multiple nodes.

Disk reloads occur on each experiment creation or swapin, so performance is a major concern. Although image creation is much less frequent than image loading, its performance is reasonably important. Once Emulab made it quick and easy for users to save the entire state of their disks in a few minutes with a few mouse clicks, they became much more willing to swap out their experiments. However, since disk loading is far more frequent and is in the critical path of experiment creation and swapin, we have focused on that.

The Emulab disk loader, termed "Frisbee" (the flying disk) uses three main techniques to improve performance from Emulab's initial 29 minutes per image. First, it carefully overlaps block decompression and device I/O. Second, it uses a domain-specific compression algorithm that can identify and skip "empty" areas of a disk such as filesystem free blocks and swap partitions, while using standard `zlib`-based compression for the remainder. This algorithm exploits the fact that many disks contain large swap partitions and mostly-empty filesystems. For example, one of our standard FreeBSD images for a 3GB partition is over 80% unused, and reduces to 156MB using Frisbee image compression, versus 473MB using naive `zlib` compression. In addition to saving network bandwidth when transferring the file, the filesystem-specific compression enables the Frisbee decompression program to skip, rather than zeroing, the unused blocks when writing out the disk image. This turned out to be very important: once we had done standard compression and implemented a multicast mechanism, DMA to the disk became the bottleneck. For the aforementioned disk image, Frisbee would write out 550MB of actual decompressed data rather than the full 3GB. The third optimization technique used by Frisbee is to multicast the compressed images to clients, dramatically reducing the required server bandwidth and improving scalability. The result is that a standard FreeBSD image that requires 88 seconds to load onto a single node, requires an average of only 97 seconds per node to load on 80 nodes.

4.3 Testing

Emulab presents unusual testing challenges for two reasons: i) It is inherently coupled to physical artifacts which, unlike software state, cannot be cloned. This makes full test and regression runs impossible. ii) Its mission is to provide a public evaluation platform for arbitrary programs. This mission simultaneously puts a premium on accuracy and precision, while presenting a fundamentally unknowable workload. Combined, these two reasons also mean that Emulab must run continuously, even as its software radically evolves.

We have countered with the following procedures. First, we have created an 8-node version of the Emulab testbed, Minibed. It serves some, but not all, of our testing needs. As an independent version of Emulab, Minibed is also key in developing federated Emulabs.

Second, we have integrated support for testing throughout the Emulab software suite. In addition to the normal operating mode, all of our software supports a "test mode" in which any operations that normally affect hardware are prevented. It allows us to make duplicate installations of Emulab databases and software, including web interfaces and daemons, and to run tests of the software without requiring exclusive access to hardware. We also have incorporated a "full-test mode," in which we can reserve hardware in the master Emulab database, and use that hardware in conjunction with the duplicate database and software. One feature that makes this possible is database-driven, node-specific redirection to alternate daemons and databases.

Third, we developed a comprehensive regression test suite that is run nightly, and optionally at compile time. However, Emulab's recent jump in complexity is starting to exceed the capability of these point tests. Therefore, we are creating a few end-to-end regression testing programs that exploit Emulab's automation and programmability to sweep through "all" Emulab variables (14 today), currently resulting in 232 measured values—an incomplete set. The key to managing this apparent complexity is to measure only a single easily-obtained value, such as the end-to-end throughput on a single TCP connection. The experiment is configured so that every value of every parameter will indirectly affect the measured value. We expect these programs, with tuning, to be sensitive regression tests of end-to-end Emulab function.

5 Efficiency

In this section we evaluate Emulab's performance, including detailed analysis of the main challenges to efficiency. We include comparisons to the measured costs of manually setting up small experiments, and an analysis of the improved resource use provided by Emulab's time- and space-sharing.

5.1 Experiment Creation and Swapping

We timed the creation of experiments of various sizes, both with and without disk loading at experiment creation-time. Most experiments fall in the former category, as our default dual-OS disk images prove sufficient for most experimenters. Loading at experiment creation time is only necessary when a custom disk image is requested, since Emulab reloads each node with the default image as soon it is freed from an experiment. Experiment creation duration is essentially equal to the swapin duration of a swapped-out experiment. This equality is due to the insignificant cost of parsing the *ns* input file and creating our IR, compared to the cost of realizing a topology.

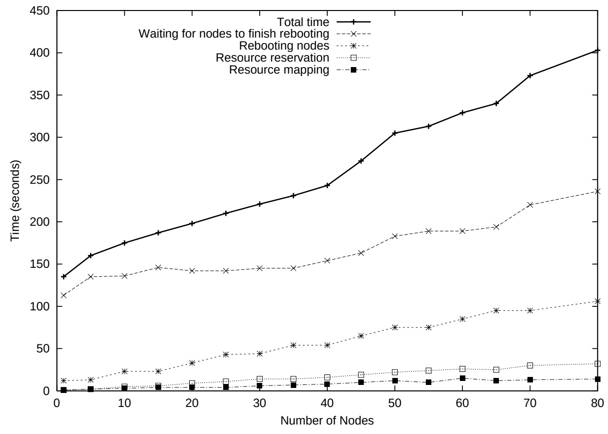

Figure 4: Time to create an experiment without disk loading. Stages shown account for 99% of the total, which is the top line. The other stages of experiment creation account for the remainder.

Figure 4 shows the time it takes to create experiments without disk reloading. The most time-consuming stages of experiment creation are shown on the graph. First, a mapping to physical resources is done by our `assign` program. Then, those resources are reserved into the experiment, and setup, such as cleaning console logs, is performed. Nodes are rebooted in parallel, in groups of eight, so as not to overstress network resources. Finally, we wait for all nodes to come back up before reporting to the experimenter that their experiment is ready. Throughout the process, we use parallel operations as much as possible to reduce setup time. For example, though it takes non-negligible time, our setup of VLANs for the experiment does not show up as a cause of experiment creation time scaling, because it is done in parallel with the longer step of rebooting nodes.

The single-node experiment takes 135 seconds, of which the majority is spent rebooting the node and waiting for it to return to multiuser mode. Each additional node adds a marginal cost of approximately 3.4 seconds.

With automatic disk loading, experiment creation takes longer, as shown in figure 5. A single node experiment takes 351 seconds with an average-size disk image. However, creation time still scales linearly, and the marginal cost for adding more nodes is still just 4.8 seconds, on average.

When rebooting large numbers of nodes, occasionally some fail to come up properly—such is the nature of commodity PC hardware. Emulab watches for such occurrences, and through issuing a "ping of death" and perhaps a power cycle, attempts to bring up nodes that fail to boot initially. As a result, experiment creation occasionally takes longer, but experiments rarely fail due to transient hardware faults.

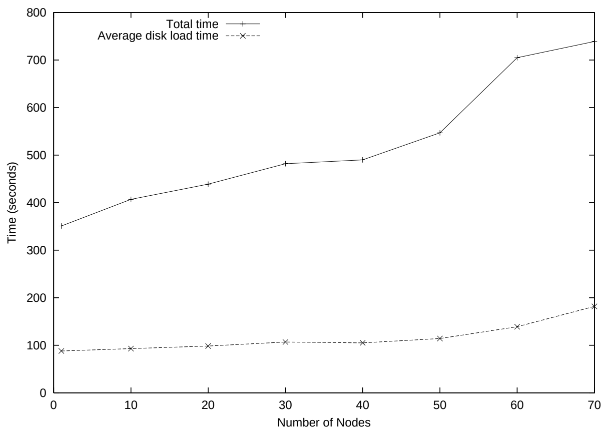

Figure 5: Time to create an experiment with disk loading. The amount of time taken up by the disk loading itself is also shown.

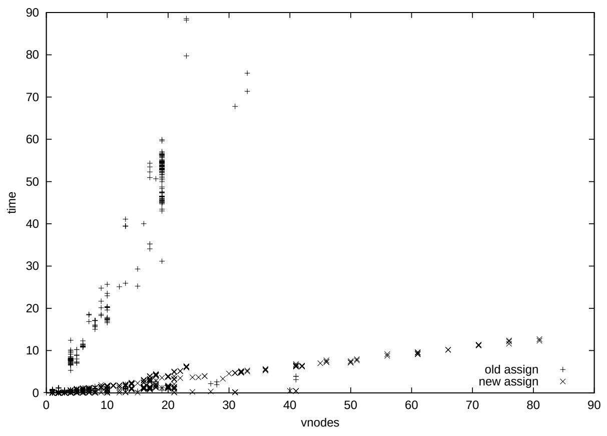

Figure 6: Performance and Scaling of `assign`

5.2 Virtual to Physical Mapping

Mapping of virtual to physical resources must be done efficiently or it could dominate experiment creation time. We now explore the performance of our two mapping programs, one for mapping Emulab-local hardware and one for mapping distributed resources.

Mapping to Emulab-local Resources: Emulab has kept detailed logs of every successful experiment since June 2001, and in less than a year has accumulated data on over 2000 experiments. From this data we have collected performance information for each experiment. Analysis shows that a reliable indicator of the difficulty of a mapping problem is the number of virtual nodes (vnodes) the user requests. The performance is indicated by the runtime of `assign`. We added a general notion of resource equivalence classes to `assign` in December 2001; Figure 6 demonstrates the resulting improvements. The new version takes less than 13 seconds on even the largest topologies, and less than 5 seconds for most experiments.

Mapping to Wide-area Resources: We conducted two experiments to test the performance of the "WAN solver" genetic algorithm. The first mapped a wide variety of experimental topologies onto our current set of 16 WAN nodes, while the second mapped a variety of experimental topologies onto a synthetic set of 256 WAN nodes.

For the first set of statistics, we chose 48 (node-count, link-count) pairs to represent a cross-section of experiment complexities. For each of these pairs, we ran hundreds of tests on automatically generated topologies. Figure 7 shows the average time the solver took to find its final solution for each complexity. Interestingly, mappings using all 16 nodes were found much faster than mappings using most, but not all, of the nodes. The results show that for modestly sized experiments, the solver does not contribute noticeably to the total experiment setup time, nor is it prohibitively slow for experiments involving most of the currently available nodes.

For the second set of statistics, we generated a physical topology of 256 WAN nodes, as well as an experimental topology for each of a variety of experiment complexities. The results of running the solver on each topology show that as experiment complexities grow, mapping takes significantly more time. All experiments with 32 nodes, as well as all sparse topologies, mapped in a few minutes, but many dense topologies using most of the nodes took three hours or more! Similar to the above, topologies using every node mapped relatively quickly, in minutes. These results show that the solver will scale well in the medium-term, but as the WAN hardware base grows dramatically, a more optimized algorithm running in parallel on multiple processors may be required to provide interactive setup times.

5.3 Disk Loading

Here we further analyze disk loading performance, one of the main costs in experiment creation and swapin. To isolate the cost of disk loading using Frisbee, we pre-create experiments of increasing sizes and manually reload the disks after the experiment is finished setting up. For each run, all nodes start loading their disks simultaneously. As Frisbee uses multicast distribution of images, one would expect it to scale gracefully, and this is reflected in Figure 8. From a base time of 88 seconds to load a 3GB (180MB compressed) image on a single node, to an average of 97 seconds per node for 80 nodes. The anomalous data point for 60 nodes (104 seconds) is likely due to transient load on the image server, but needs to be investigated.

5.4 Scaling of Simulated Resources

We explored the extent to which simulated nodes can be multiplexed onto physical nodes to increase experiment scale: it appears that integrating simulated nodes into Emulab provides a scaling factor on the order of 100.

We ran an instance of *nse* simulating many constant bit rate UDP flows, between pairs of nodes on links of 2Mbps/50ms, running at 2Mbps. To measure *nse*'s ability to keep pace with real time, and thus with live traffic, a 2Mbps/50ms 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 850 MHz

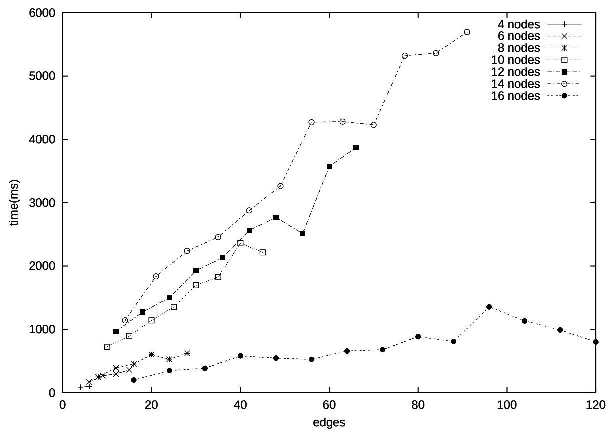

Figure 7: Average time for WAN solver to find a final solution for a variety of experimental topology complexities (node and edge counts.)

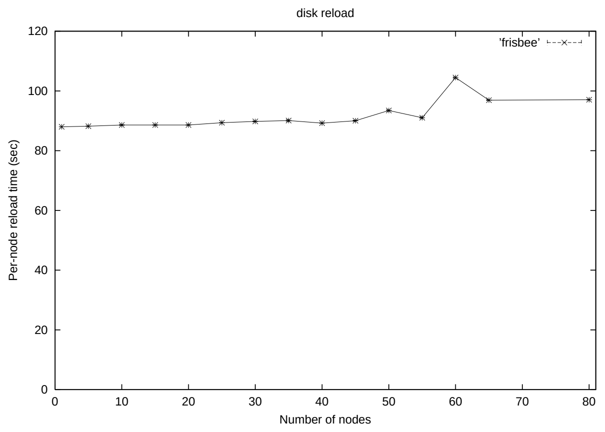

Figure 8: Scaling of "Frisbee" disk loader with increasing number of nodes.

PC, we were able to scale the number of simulated flows up to 105 simulated links and 210 simulated nodes, while maintaining the full throughput of the live TCP connection. Past 105 simulated links, the throughput dropped precipitously. We also measured *nse*'s TCP model on the simulated links: the performance dropped after 90 simulated links. (We have improved the performance of *nse* by a factor of two by making some modest changes to its scheduler.)

We also did preliminary validation of *nse* vs. real nodes: *nse* reported TCP throughput 3% higher than a physical Emulab node.

5.5 Comparison to Manual Configuration

After a time, it becomes easy to take Emulab's ease of use for granted. To add quantitative estimates to our many anecdotal accounts of Emulab's value in configuring experiments, we had a student with significant Linux system administration experience create experiments manually. This process included wiring machines together, installing OSs, and configuring software. His first (learning) trial was a simple two-node setup with an intervening Dummynet node. This took 10.4 hours, and the student estimated it would have taken a full week had not experienced Emulab staff been present to assist. The second time through he went much faster, but still took nearly two hours. Emulab can build a comparable setup, including the installation of a custom disk image, in six minutes.

He then set up a "dumbell" topology consisting of four end hosts communicating through two routers, with a single traffic generator. This time we assumed the nodes already had functional OS installations, so we measured system configuration, routing setup, and traffic generation. After subtracting the time spent learning the new tasks, he finished in 3.25 hours. The equivalent Emulab time is less than three minutes.

To give a feeling for the scale involved in automating Emulab's type of configuration, we counted the number of operations involved in Emulab's experiment swapin process, approximately 20% of the entire experiment creation process. We counted operations comparable to steps one would take in manual configuration, such as creating or updating configuration files, interacting with the database, and calculating the mapping of a topology onto the physical network. We found 233 separate operations just in this portion of experiment configuration.

5.6 Time and Space Sharing

We analyzed Emulab's historical logs for the last 12 months to derive quantitative estimates of the value of time-sharing (experiment swapping) and space-sharing. Although the behavior of both users and Emulab management would change without such features, the estimate is still revealing. Without Emulab's ability to time-share its 168 nodes, a testbed of 1064 nodes would have been required to provide equivalent service. Similarly, without space-sharing, 19.1 years would be required.

6 Validation

In this section we present data that helps validate Emulab's behavior. We start with low-level and point aspects of the system, progressively moving to more comprehensive evaluations.

6.1 Switch Effects

One of Emulab's fundamental mechanisms is the flexible and accurate emulation of a connection between any two (local) nodes. The flexibility is attained by having all nodes connect to a programmable switch and by inserting "WAN emulator" nodes between pairs of hosts to achieve desired bandwidth, delay, and packet loss characteristics. In the following two sections we evaluate the accuracy of our emulation.

To characterize the delay introduced by the Cisco switches we measured the round-trip time (RTT) of packets sent between two nodes, first with the nodes directly connected via Cat5 crossover cables, and then with the nodes connected via a VLAN in the switch.

packet length	RTT direct (μs)	RTT switch (μs)	per-byte overhead
64	63.4	92.1	0.22
256	102.2	176.7	0.15
512	143.0	266.7	0.12
1024	225.1	445.2	0.11
1518	297.1	601.3	0.10

Table 2: Measurement of packet round-trip time (RTT)

The two nodes run minimal OSKit kernels, so that scheduling and other overheads found in a full multi-processing kernel could be eliminated. The first node runs the `request` kernel which serially transmits simple UDP packets and waits for a reply. The second node runs the `reply` kernel which reverses the MAC and IP addresses and sends the packets back through the same interface. `request` records the time at which it transmits and receives each packet, producing the resulting round-trip time.

Table 2 summarizes the results of the two tests for various packet sizes. While in percentage terms the difference is significant, in absolute terms it is quite reasonable. Crossing the diameter of our switch topology at its longest point, which entails crossing two trunk links, adds 9μs(10%) to the round-trip time for a 64-byte packet.

Another switch-related concern is that *switch management*, such as setting up VLANs, not affect unrelated experimental traffic. To check this, we sent constant high-speed packet streams between nodes, while setting ports up/down and creating/removing VLANs. Those are the frequent operations associated with creation and termination of experiments. In all cases, the resulting per-packet mean round-trip time and standard deviation were the same as for control streams which experienced no concurrent switch configuration.

6.2 WAN Emulator Node Effects

There are two concerns with using off the shelf PC's and a general purpose OS as WAN emulator nodes. One is performance: whether the machine can keep up with the network when the emulated links are operated at full speed. The more important issue is how accurate are the emulated delays, bandwidths, and packet loss rates.

As a capacity test, we generated streams of UDP round-trip traffic between two nodes, with and without an emulator node in between. We found that, for 1518-byte packets, the emulator node has no adverse effect: either configuration can easily saturate a 100Mb link. With 64-byte packets, the two nodes can exchange 45000 packets (2.9MB) per second when connected directly versus 30000 (1.9MB) when joined by an emulator node. Note that these are round trip measurements, so the packet rates with and without the emulator node are really twice the numbers reported above.

We then ran another series of experiments to characterize the emulator node, by sending traffic at the node's full capacity, and measuring how faithfully it emulates link delay, bandwidth and packet loss rate. Tables 3, 4 and 5 summarize these experiments.

packet length	expected RTT (ms)	observed RTT (ms)	% error	90% confidence interval
5 ms delay				
64	10	10.036	0.036	10.034 - 10.037
1518	10	10.293	2.93	10.261 - 10.325
300 ms delay				
64	600	601.271	0.21	601.26 - 601.27
1518	600	602.487	0.41	602.485 - 602.488

Table 3: Measured accuracy of delay emulation as a function of packet size and link delay. The link bandwidth was unlimited and the link loss rate was 0.0. The sample size was 10000 for 64 byte packets and 1000 for 1518 byte packets.

link bandwidth	observed bandwidth	% error	std. dev.
75 Mbps	74.87 Mbps	0.17	0.028
10 Mbps	9.97 Mbps	0.30	0.004

Table 4: Measured accuracy of bandwidth emulation as a function of link bandwidth. The link delay was 0ms and link loss rate was 0.0. Traffic consisted of 1518-byte packets at 8Kpps.

link loss rate	observed loss rate	% error	std. dev.
0.01	0.0101	1	0.002
0.5	0.5018	0.36	0.001

Table 5: Measured accuracy of packet loss rate emulation as a function of link loss rate. The link bandwidth was unlimited and the link delay was 0ms. Traffic consisted of 1518-byte packets at 8Kpps.

6.3 Event System

The Emulab event system consists of four components: a master event server, per-experiment event schedulers, a dynamic event generation tool, and various event clients. The event messaging model and master event server are taken from Elvin [41], a notification and messaging service. While Elvin supports a flexible, content-addressable messaging model, Emulab defines a small set of fixed message types, using `elvind` to handle event subscriptions and distribute notifications according to those subscriptions. Since `elvind` is only an event dispatcher, we have an event scheduler to queue events until their trigger time, at which point they are sent to the event server for immediate delivery to interested parties. The event scheduler starts by extracting static, ns-specified events from the database but it also handles dynamically generated events from the event generation tool. The event generation tool is a simple command-line tool, using a syntax similar to that in the ns file, for generating various events at specified times. Current clients of the event system include traffic generators, a delay node control agent and a general remote execution facility.

Since events are scheduled and queued centrally on `masterhost` and distributed at the time of firing, it is important that events be delivered in a timely fashion to ensure event firing accuracy. For individual node events, we expect the *latency* of event delivery to be on the order of the one-way trip time between `masterhost` and the node. For events that are to occur simultaneously on multiple nodes, there is an additional issue to consider. Since our Elvin master event server does not currently use multicast for event distribution, there will be some *skew* between the first and

node	event delay (ms)	stdev	clock offset (ms)
pc39	0.637	0.136	0.441
pc24	0.177	0.140	0.677
pc16	0.936	0.142	0.316
pc32	1.124	0.133	0.173
pc27	1.091	0.136	0.249
pc21	1.225	0.137	0.152
pc30	1.223	0.135	0.217
pc25	1.016	0.138	0.301
pc31	1.362	0.138	0.120
pc33	1.963	0.139	-0.143
pc38	1.300	0.140	0.186
pc26	1.814	0.139	-0.032
pc28	2.207	0.136	-0.279
pc34	0.403	0.134	0.603

Table 6: Event notification delay as measured between event dispatcher and client, averaged over 10 runs. Nodes in the table are listed in the order in which the event was dispatched to highlight skew. "Clock offset" is the `ntp` measured offset between the node's local clock and the reference clock on `masterhost` (the source of the event).

last nodes receiving the event.

To measure the latency and skew of event delivery, the per-experiment event scheduler was modified to record a timestamp prior to passing an event to `elvind` for dispatch. A special event client was then run on all nodes within an experiment. This client received events, extracting the time stamp and comparing it with the local current time. After compensating for clock skew between the node and `masterhost`, the result is the delivery latency. An experiment was created with 15 nodes and an event "broadcast" to all nodes for their capture. The event was sent 10 times and the per-node times averaged. Table 6 summarizes the results, with one node omitted due to an unacceptably inaccurate clock. As delays are on the order of the resolution of clock synchronization, the results are highly variable but clearly quite small. A general trend can also be seen, with delay increasing by around 2ms between the first and last nodes notified, demonstrating that there is a modest skew.

6.4 Validation Against the Wide-Area

In this section we outline two macrobenchmark comparisons between real Internet nodes and corresponding configurations on Emulab. The first example also demonstrates Emulab's flexibility and the transparency of its WAN link specification.

Distributed Multiplayer Game: Besides validation between Emulab's distributed nodes and its local nodes, in this experiment we also demonstrated Emulab's ability to map requested WAN links into the best-matching live Internet links. We have 5 synthetic clients of the standard multiplayer game "DOOM." The communication protocol used by this version of DOOM is simple (and stupid). At a target rate of 30 times per second, each client sends unicast packets to all other clients, doing so only once it has received all the prior period's packets from its peers.

	Live Internet			Emulated		
	tics	stddev	retransmits	tics	stddev	retransmits
Fast	29	0.00	1.10	28	0.67	1.10
Slow	21	0.73	1.70	21	0.52	2.80

Table 7: Median "tic" rates and packet retransmission counts achieved by "DOOM" clients, both on live Internet and emulated links. Numbers are repeated both for nodes with uniformly fast links, and with some slower links mixed in.

We evaluated DOOM on four network configurations, making at least four repeated runs on each. We specified the desired latency and bandwidth of the 10 links that create a fully-connected graph between all five clients.

For the first configuration, by giving a node type of `pcvremote` in the *ns* file, we specified that the nodes be wide-area ("`remote`") and they be virtual nodes ("`v`"), i.e., multiplexed onto some set of the physical remote (RON) nodes. Emulab's mapping phase, the genetic algorithm described in Section 4.1.2, found the best-matching Internet links from among the wide-area nodes that still had available virtual node "slots." (If we had preferred coarser-grained specification of the wide-area experiment, instead of giving link characteristics we could have requested virtual nodes by wide-area (last mile) subtype, such as "`pcvinet2`" (Internet2) and "`pcvintl`" (international).)

For the second configuration we used the same link specification, but by changing just one line (inside a Tcl loop) that set the node type, we obtained Emulab-local nodes and emulated links. The third and fourth configurations were analogous to configurations one and two, but we requested some substantially slower links, with the goal of demonstrating an application-visible effect.

The results were good, as presented in Table 7. The two key metrics in DOOM are "tic rate" and packet retransmission. Tic rate in this example is affected primarily by latency, and represents the rate at which progress is made in the system—a higher tic rate indicates faster progress. Packet retransmission rates are governed by bandwidth and packet loss rate; there are typically only a handful of retransmitted packets per trial.

Wide-Area Database Replication: Researchers at Johns Hopkins University are studying group communication mechanisms for wide-area replication of databases. In the course of their research they compared results from the CAIRN wide-area network [12] to their results on local Emulab hardware, with Emulab links emulating the delay and bandwidth characteristics they observed in CAIRN. As reported in their technical report [5], their application-level measurements of communication characteristics matched well. Emulab offered them two other advantages over CAIRN: due to the control offered by Emulab, they were easily able to study the effects on their system of varying network characteristics. Secondly, they could obtain nodes all of the same type, unlike on CAIRN. In summary, after obtaining baseline data from the Internet, Emulab's control and repeatability let them better study the effect of network characteristics on their system.

7 Case Studies and Experience

7.1 TCP Dynamics

Network simulators, such as *ns*, have proven invaluable in studying TCP behavioral dynamics [19]. Because of abstractions such as lack of a CPU utilization model and one-way protocols with simplified window and ACK behavior, simulation should be validated empirically. Ironically, the potential for bugs and unspecified design parameters mean that real implementations do not necessarily define valid behavior, either. Nevertheless, the notion of "deviant behavior" [16] allows us to simultaneously gain confidence in the validity of simulation and the correctness of implementation.

The *ns* maintainers run nightly regression tests [32]. Emulab's transparent ability to parse *ns* scripts means these scripts can instead be used to validate *ns* behavior against emulation. Further, the tests may drive regression testing of a kernel implementation or a comparison across several implementations. This section presents preliminary results that show the feasibility of automating this process. The study of low-level, fine-grained TCP dynamics shows Emulab's flexibility in modulating a virtual network at various scales.

This approach is a general and powerful means of testing or validating the dynamic behavior of network protocols as realized in kernels or within simulation. Our framework executes a test script within *ns* and parses output trace files to determine where to generate traffic, which packets are dropped, and which links suffer losses. It then configures a network topology via Emulab's event system and passes a list of target drop packets to the correct Dummynet node. Again via the event system, the framework starts a program object to record packet traces and finally invokes the traffic generators.

Figure 9 shows a simple test from the *ns* validation suite that drops a single packet in a TCP New Reno stream. The *ns* and FreeBSD 4.5 senders detect a Triple Duplicate ACK and perform a Fast Retransmit immediately. By contrast, we discovered that FreeBSD 4.3 does not retransmit until triggered by a timer expiration, which greatly degrades throughput.

This example highlights the potential of automating comparison of simulation and emulation. The apparent visual similarity of the graphs is supported by statistics. FreeBSD 4.5 achieves a mean bandwidth of 50232.4 Bps over 10 experiments, with a standard deviation of 4.09. *ns* achieved a mean throughput of 48090 Bps.

Secondly, this example shows the merit of automated regression testing applied to large, evolving software systems. The behavior in FreeBSD 4.3 is caused by an uninitialized variable. A thorough application of the full suite of TCP tests may well uncover additional bugs. Such subtle bugs would be exceeding difficult to detect and reproduce without Emulab's fine-grained control. Our initial work in automating TCP regression testing has also led to the discovery of a discrepancy between the New Reno Fast Retransmit [27] implementation in FreeBSD 4.5 and the corresponding RFC [18].

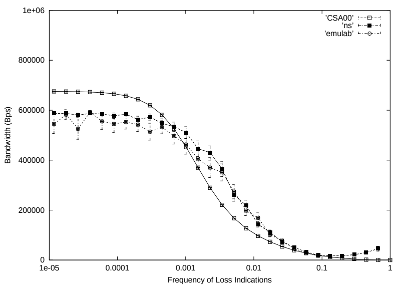

Figure 11: Comparison of Simulated, Emulated, and Analytic TCP Performance

Figure 10 depicts another scenario from the *ns* test suite that drops three packets during New Reno Slow Start. Again, *ns* reacts gracefully to the losses, retransmitting each in a timely fashion. The last loss under FreeBSD 4.5 induces a retransmission only after a timeout. This TCP implementation confounds the "`send_high`" and "`recover`" variables discussed in the New Reno Fast Recovery RFC [18], using instead a single "`snd_recover`" variable. This results in overly eager invocation of the False Fast Retransmit algorithm [27], which suppresses Fast Retransmission.

7.2 TCP Latency Modeling

Analytic TCP models [37, 13] tend to be both heavily parameterized and heavily dependent on network assumptions, including independent packet loss, specific queue disciplines, and zero scheduling or buffering overhead. The former feature often motivates parameter-space exploration through simulation, while the latter calls for validation on real hardware. These two goals are generally at odds with one another, forcing an experimenter to make concessions to one or the other. By leveraging Emulab's automation and control, we were able to explore a parameter space consisting of 50 iterations each of 23 loss rates. After experiment creation, the results were collected without experimenter intervention.

These results are plotted alongside their analytic and simulated counterparts in Figure 11. In all cases, 1MB is transferred across a link with a 50ms delay. Data points for simulation and emulation represent averages of 50 iterations and are plotted with 90% confidence intervals.

Simulation and the analytic model exhibit average errors of 7% and 19% respective to emulation. The discrepancy between *ns* and the analytic model is consistent with the findings of Cardwell et al. The simplicity of emulating a large range of parameters allows an experimenter to gauge

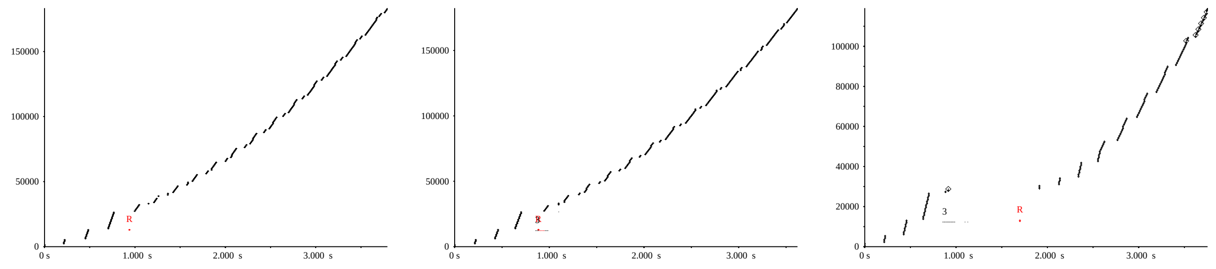

Figure 9: New Reno One Drop Test: (a) *ns* (b) FreeBSD 4.5 (c) FreeBSD 4.3

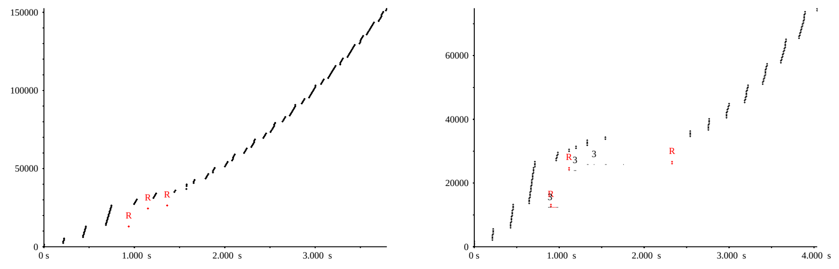

Figure 10: New Reno Three Drop Test: (a) *ns* (b) FreeBSD 4.5

an analytic model's sensitivity to its assumptions. For example, we could easily study the model's applicability to RED queues, which are inconsistent with the model's assumption of drop-tail behavior.

7.3 The Armada I/O Framework

This example from others' research demonstrates the value that Emulab's *automation* brings to experimental evaluation; it is further described in a technical report [36]. Automation enables easy exploration of large parameter spaces, as one can do in simulation.

Dartmouth's Armada[35] system is designed to improve I/O in computational grids. The results it obtains depend highly on the bandwidth, latency, and packet loss rate of the wide area network connecting the nodes or LANs involved. They ran experiments that tested every possible combination of 7 bandwidths, 5 latencies, and 3 application parameter settings on four different configurations of the Armada system on a set of 20 nodes, performing a total of 420 different tests in 30 hours, averaging 4.3 minutes each. Because Emulab supports such high levels of automation, this was all done inside one experiment.

7.4 The "Average" Project/User

Emulab currently has 201 users in 58 different projects, with a few users belonging to more than one project. 49 of those projects have been active enough to configure one or more experiments. Excluding our own Emulab development work, the most active project has created over 400 experiments, and three others have created more than 100. Three of our most consistent users have each had 8–12 nodes in almost constant use for 6–8 months at a time.

Almost half of the projects using Emulab are working on distributed systems research, and about 40% on networking research. The remainder is split between Active Networks research, instructional use for classes, and use of Emulab as a computational cluster. Clearly, Emulab is successful as a tool, and has wide appeal among several areas in the field. The wide variety of projects using Emulab illustrate how its features are important for these projects. For instance, educational users of Emulab (primarily OS and networking classes at remote universities) have enjoyed the isolation and protection we offer, which makes Emulab one of the only places where they can safely give their undergraduate students root access. Network researchers, on the other hand, find Emulab's ability to set up arbitrary network topology to be vital.

7.5 The "Average" Experiment

Emulab is in a unique position to characterize the needs of network and distributed systems experiments. We have a sample of over 2000 experiments, presumably representative of large classes of research in these areas. From this sample we've extracted information that will be useful in recognizing the experimenters' requirements. For example, the most frequent experiment size was two nodes, even though many more nodes are typically available. We attribute this to development and debugging work in preparation for larger scale experimentation. The distribution of experiment sizes is multi-modal, with another mode at 15–20 nodes, and smaller modes near 30 and 40 nodes. We believe that another significant mode would be at much larger scales, such as those typical of peer-to-peer overlay networks, in in the hundreds or thousands of nodes. We are currently analyzing our logs to determine the distribution of numbers of links—which is confounded by the high incidence of LANs that experimenters specify.

7.6 Non-IP Protocols

Ideally, Emulab's local nodes should be usable by experiments involving protocols that are not IP-based, and that was one of our original goals. However, there are two known issues that limit the ability to do this. First, node names are virtualized at the IP level: IP addresses within an experiment remain constant across swapins. The ideal solution would be to virtualize at the link layer, but that requires additional interaction and synchronization with the Emulab switch fabric. An easier solution is to abstract Emulab's notion of an address and continue to virtualize above the link layer. The second problem is that the standard tools used within Emulab are primarily, if not exclusively, IP-based. This includes the OSes, Linux and FreeBSD, routing agents, DNS, DHCP, Dummynet, and the tunneling code. Some may be easily extended to work with arbitrary protocols, but in general, providing complete replacements would be a significant task.

8 Related Efforts

Emulab uses well-known experimental infrastructure as building blocks, particularly *ns* [42], the *nse* emulation facility [17], Dummynet [40], and the RON wide-area testbed [6]. Emulab's novelty is its ability to integrate these techniques in a manner that provides ease of use, control, and realism. For example, while *nse* provides the mechanism that allows simulated traffic generators to affect real links, establishing such a scenario without Emulab requires manual configuration.

ModelNet [43] is a large-scale network emulation facility that utilizes a gigabit cluster serving as a router core to deliver packets to edge nodes. ModelNet most closely approaches Emulab's automatic configuration of physical resources by distilling a target topology specified in any number of high-level formats. However, as ModelNet focuses on scalability, it does not extend this automation to consistent control over heterogeneous physical and simulated resources. Like other emulation mechanisms, including trace modulation [34] and NIST Net [33], ModelNet is a complimentary approach that can be deployed in place of or alongside Dummynet.

Trace modulation [34] recreates observed end-to-end characteristics of a wireless network. Interposing trace modulation instead of Dummynet would provide support for wireless links in Emulab.

Over the past decade there have been a large number of additional network emulation efforts. These include hitbox [3], the Ohio Network Emulator (ONE) [4], and Rice's support for evaluating their web-oriented kernel and OS optimizations [38, 8]. One of the earliest and largest uses of emulation was a twelve node deployment at USC in 1994, used to study TCP Vegas [3]. They cite an emulator effort at Bell Labs [29], which apparently started to build a more general emulator. Another category is represented by Michigan's "Orchestra" fault-injection system [15]. Although there have been exceptions, typically these single node emulators were tailored for a specific research application, were (or are) not well supported or widely distributed, and have not been widely adopted by external users.

The "Access" project [7] originated the vision of a distributed set of testbeds. They explored the feasibility of building a set of small testbeds, distributed over many sites around the world. The Access vision overlapped with Emulab in our shared emphasis on completely replaceable node software, our operational model of a Web-accessible master control host, and provision of power control and serial console lines. However, we differ in many other respects. Fundamentally, Access did not intend to provide an emulation facility, did not have our emphasis on ease of use, nor did it intend to offer integration. They did recognize a need we identified only later, for real wide-area links for some experimenters.

Network simulators successfully isolate protocol dynamics but may do so at the expense of accuracy. Therefore, results from simulators such as *ns* [42] and GloMoSim [1] may not be valid indicators of deployed performance [20]. Brakmo and Peterson [10] highlight differences between simulated and implemented TCP protocols. They present *x*-sim, a simulator which avoides inaccuracies by embedding actual protocol code. However, this approach requires the non-standard *x*-kernel.

Heidemann et al. describe means of gaining confidence in network simuation [25]. The integration of simulation within a live experimental environment provides another validation opportunity. This idea leverages the simultaneous integration of simulated nodes with physical nodes to provide greater scale through simulation tempered by the realism of emulated or wide-area nodes. Physical nodes are integrated with simulation to detect possible inaccuracies. For example, if simulation is sufficiently accurate for a given application, simulated and emulated nodes should exhibit similar behavior under careful scrutiny. Secondly, this hybrid approach should be largely indistinguishable from pure simulation.

Emulab's design and the problems encountered along the way have relevance outside the domain of network experimentation. In particular, Emulab offers a unique perspective on cluster management. Most existing systems, including GLUnix [23], PBS [26], Condor [30], LSF [46], and Ganglia [14] mask a multitude of physical nodes behind the illusion of a single virtual interface. This virtualization is provided by *software*– often in user-level libraries. Emulab instead virtualizes *hardware*. This distinction has important consequences. For example, because of their relatively high level of interposition, these systems generally can not checkpoint processes that fork or perform IPC. An as yet unimplemented design of checkpointing under Emulab seamlessly copes with forked process and IPC because it captures and recreates a node's entire hardware state. Océano [2] provides an application-hosting "computing utility powerplant" by similarly virtualizing hardware resources. This project also employs VLANs and automatic node reconfiguration. However, reconfiguration is driven by Service Level Agreements. Since the system is not designed as an experimental testbed, it does not expose virtual network resources to user control.

9 Conclusion

We have presented Emulab– software that provides an experimental platform for local- and wide-area network research. Emulab leverages the strengths and diversity of emulation, simulation, and live wide-area networks to reach levels of ease of use, realism, and control unsurpassed by other experimental environments. Emulab gives experimenters access and control over virtual nodes and links, integrated behind a consistent and familiar *ns* interface. It manages the topology to allow a packet to traverse simulated, emulated, and wide-area links along its path from one virtual host to another.

We have begun designing extensions to Emulab that will integrate a new class of virtual nodes to extend its near effortless setup, configurability, and repeatability into the mobile, wireless domain [45]. Emulab will exploit a large, dense set of wireless devices via mobile, passive couriers that move predictably in time and space. We will use two types of couriers: students moving from class to class with radio-equipped PDAs, and city and campus busses with wireless PCs. Both exhibit predictable movement patterns. To accommodate purely static experiments, Emulab will consist of a dense deployment of wireless devices throughout campus, indoors and out. Experimenters will specify their desired network and, for mobile experiments, its nodes' movements; our software will select the subset of nodes that best match.

To accomodate larger-scale environments, we are examining the federation of single-site Emulab instances. From an experimenter's perspective, federated resources will look much like current wide-area virtual nodes and links. However, integration of these resources presents new challenges and opportunities. A centrally-administered federation is faced with a scalability barrier due to economic, market, and political limitations. Further, indirection through a single, remote manager threatens availability by placing control of local resources elsewhere. By constrast, our vision is analogous to the Internet, in which wildly physically diverse, autonomously developed and administered networks interconnect only at their boundaries to from an "InterNetwork," yet expose interior nodes by direct addressability. The sheer flexibility, scale, and autonomous administration of such a federation addresses the current needs of overlay and peer-to-peer network research and creates the very conditions most conducive to the rise of new "killer apps".

Acknowledgements

The authors are grateful for discussion with and implementations contributed by Dave Andersen, Chris Alfeld, and Kirk Webb. Special thanks to Dave for offering up MIT's RON testbed to Emulab's management. Russ Christensen's manual configuration of small-scale testbeds demonstrated Emulab's automation. The efforts of Alastair Reid, Tim Stack, and Parveen Patel in running experiments and of Eric Eide in editing are much appreciated. John Regehr's and other reviewers' comments on earlier drafts helped shape this paper. Thanks also to Ron Oldfield for contribution of Armada results, and to all of our users who provided feedback.

The authors are indebted to Nicolas Christin for suggesting areas of networking research that would benefit from emulation validation. Neal Cardwell and Stefan Savage helped in understanding the analytic models and results from their work on TCP modeling. Mark Tinguely provided continued clarifications of the FreeBSD TCP implementation and its adherence to relevant RFCs.

Finally, we are deeply grateful to our many sponsors, especially the National Science Foundation under grant ANI-0082493, DARPA/Air Force under grant F30602-99-1-0503, and Cisco Systems.

References

[1] GloMoSim: Network Simulator, UCLA. http://www.isi.edu/nsnam/ns/.

[2] The Océano Project. http://www.research.ibm.com/oceanoproject/.

[3] 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, Aug. 1995.

[4] M. Allman, A. Caldwell, and S. Ostermann. ONE: The Ohio Network Emulator. Technical Report TR–19972, Ohio University Computer Science, Aug. 1997.

[5] Y. Amir et al. Practical Wide-Area Database Replication. Technical report, Johns Hopkins University, 2002. http://www.cnds.jhu.edu/-pub/papers/cnds-2002-1.pdf.

[6] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proc. 18th SOSP*, pages 131–145, Oct. 2001.

[7] T. Anderson. A Case for Access: A High Performance Communication and Computation Environment for Wide Area Distributed Systems, Networking, and Applications Research. http://www.cs.-washington.edu/homes/tom/access/.

[8] G. Banga, J. Mogul, and P. Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proc. of the 1999 USENIX Annual Technical Conf.*, June 1999.

[9] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[10] L. S. Brakmo and L. L. Peterson. Experiences with Network Simulation. In *Proceedings of ACM SIGMETRICS'96*, May 1996.

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

[12] CAIRN: Collaborative Advanced Internet Research Network. http://-www.isi.edu/CAIRN/.

[13] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP Latency. In *Proc. of INFOCOM '00*, Mar. 2000.

[14] B. Chun and M. Massie. Ganglia Cluster Toolkit. http://-ganglia.sourcefourge.net/docs/ganglia.pdf.

[15] S. Dawson, F. Jahanian, T. Mitton, and T.-L. Tung. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *Proc. International Symposium on Fault-Tolerant Computing*, June 1996.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proc. 18th SOSP*, Oct. 2001.

[17] K. Fall. Network Emulation in the Vint/NS Simulator. In *Proc. of the 4th IEEE Symposium on Computers and Communications*, 1999.

[18] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. Internet Request For Comments 2582, Internet Engineering Task Force, Apr. 1999.

[19] S. Floyd and V. Paxson. Difficulties in Simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4):392–403, Aug. 2001.

[20] S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE/ACM Transactions on Networking*, 9(4), August 2001.

[21] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for OS and Language Research. In *16th SOSP*, pages 38–51, Oct. 1997.

[22] M. R. Garey and D. S. Johnson. *Computers and Intractability: A guide to the theory of NP-completeness*. W. H. Freeman, 1979.

[23] D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson. GLUnix: a Global Layer Unix for a Network of Workstations. *Software— Practice and Experience*, 28(9), July 1998.

[24] J. Heidemann, N. Bulusu, J. Elson, C. Intanagonwiwat, K. Lan, Y. Xu, W. Ye, D. Estrin, and R. Govindan. Effects of Detail in Wireless Network Simulation. http://www.isi.edu/˜johnh/PAPERS/-Heidemann00d.html, Sept. 2000.

[25] J. Heidemann, K. Mills, and S. Kumar. Expanding Confidence in Network Simulation. *IEEE Network Magazine*, pages 58–63, Sept/Oct 2001.

[26] R. L. Henderson and D. Tweten. Portable batch system: Requirements specification. Technical report, NAS Systems Division, NASA Ames Research Center, 1995.

[27] J. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *SIGCOMM '96*, Aug. 1996.

[28] IXP1200. http://developer.intel.com/design/network/products/-npfamily/ixp1200.htm.

[29] A. M. Lapone, N. F. Maxemchuk, and H. Schulzrinne. The Bell Laboratories Network Emulator. Technical Report BL0113820-930913-64TM, AT&T Bell Labs, Sept. 1993.

[30] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor— A Hunter of Idle Workstations. In *Proc. of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[31] P. E. McKenney, D. Y. Lee, and B. A. Denny. *Traffic Generator Software Release Notes*. SRI International and USC/ISI Postel Center for Experimental Networking, Jan. 2002. http://www.postel.org/tg/.

[32] The Network Simulator ns-2: Validation Tests. http://www.isi.edu/-nsnam/ns/ns-tests.html.

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

[34] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-Based Mobile Network Emulation. In *Proc. of SIGCOMM '97*, pages 51–61, Cannes, France, Sept. 1997.

[35] R. Oldfield and D. Kotz. Armada: A parallel file system for computational grids. In *Proc. of IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 194–201, Brisbane, Australia, May 2001. IEEE Press.

[36] R. Oldfield and D. Kotz. Using the Emulab network testbed to evaluate the Armada I/O framework for computational grids. Technical report, Dartmouth College, May 2002. ftp://ftp.cs.dartmouth.edu/-pub/raoldfi/armada/oldfield:armada-emulab-tr.pdf.

[37] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *Proc. of SIGCOMM '98*, Sept. 1998.

[38] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proc. of the Third Symposium on Operating Systems Design and Implementation*, pages 15–28, New Orleans, LA, Feb. 1999.

[39] PXE Preboot Execution Environment Specification Version 2.1. ftp://download.intel.com/ial/wfm/pxespec.pdf.

[40] L. Rizzo. Dummynet and Forward Error Correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, New Orleans, LA, June 1998. USENIX Association.

[41] B. Segall and D. Arnold. Elvin Has Left the Building: A Publish/Subscribe Notification Service with Quenching. In *Proc. of the 1997 Australian UNIX and Open Systems Users Group Conference (AUUG '97)*, Brisbane, Australia, Sept. 1997.

[42] The VINT Project. *The* ns *Manual*, Apr. 2002. http://www.isi.edu/-nsnam/ns/ns-documentation.html.

[43] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. Submitted for publication. http://www.cs.duke.edu/˜vahdat/-ps/modelnet.pdf, May 2002.

[44] J. Wallerich. Design and Implementation of a WWW Workload Generator for the NS-2 Network Simulator. http://www.net.uni-sb.de/˜jw/nsweb/, Aug. 2001.

[45] B. White, J. Lepreau, and S. Guruprasad. Wireless and Mobile Netscope: An Automated, Programmable Testbed. http://-www.cs.utah.edu/l̃epreau/emulab-wireless.ps. Submitted for publication in Mobicom'02., Mar. 2002.

[46] S. Zhou, J. Wang, X. Zheng, and P. Delisle. Utopia: A Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software— Practice and Experience*, 23(2), 1993.