An Integrated Experimental Environment for Distributed Systems and Networks

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

School of Computing, University of Utah

www.flux.utah.edu www.netbed.org

Abstract

Three experimental environments traditionally support network and distributed systems research: network emulators, network simulators, and live networks. The continued use of multiple approaches highlights both the value and inadequacy of each. Netbed, a descendant of Emulab, provides an experimentation facility that integrates these approaches, allowing researchers to configure and access networks composed of emulated, simulated, and wide-area nodes and links. Netbed's primary goals are *ease of use*, *control*, and *realism*, achieved through consistent use of virtualization and abstraction.

By providing operating system-like services, such as resource allocation and scheduling, and by virtualizing heterogeneous resources, Netbed acts as a virtual machine for network experimentation. This paper presents Netbed's overall design and implementation and demonstrates its ability to improve experimental automation and efficiency. These, in turn, lead to new methods of experimentation, including automated parameter-space studies within emulation and straightforward comparisons of simulated, emulated, and wide-area scenarios.

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*. Packet-level discrete event simulation and live network experimentation represent two extremes. Simulation presents a controlled, repeatable environment. However, its level of abstraction may be too high to capture low-level effects such as the impact of interrupts under heavy load. Live networks achieve realism, but surrender repeatability and the ability to modify or even monitor internal router behavior. Emulation [1, 27, 36, 42] is a hybrid approach that subjects real applications, protocols, and operating systems to a synthetic network environment. While single-node WAN emulators, such as Dummynet [36], introduce artificial delays, losses, and bandwidth constraints in a controlled manner, they require tedious manual configuration.

Netbed complements existing experimental environments. It spans simulation, emulation, and live network experimentation by integrating them into a common framework. This integration brings the control and ease of use usually associated with simulation to emulation and live network experimentation without sacrificing realism. It gives users the individual benefits of simulation, emulation, and live network experimentation, configured and controlled in a consistent manner. Further, integration facilitates interaction, comparison, and validation across the three domains.

Netbed is a software system that provides a time- and space-shared platform for research, education, or development in distributed systems and networks. It leverages local nodes, allocated from clusters and temporarily dedicated to individual users, for emulation; this paper often refers to these as emulated nodes. Netbed also employs geographically-distributed nodes that are simultaneously shared amongst users; this paper frequently refers to such resources as wide-area nodes. Researchers access these resources by specifying a virtual topology graphically or via an *ns* script [40], causing Netbed to automatically configure a physical topology. An *experiment* is defined by this configuration and any run-time dynamics, such as traffic generation, specified via the general-purpose *ns* interface. When realizing the virtual topology, Netbed virtualizes host names, IP addresses, links, and nodes. Virtual nodes may be instantiated from a large set of local nodes, from a smaller set of distributed nodes, or within *ns* simulation. Virtual links may map directly to local-area links, may be matched to similar wide-area links, or may be emulated by interposing Dummynet nodes to regulate bandwidth, latency, loss, and queuing behavior.

Netbed's framework provides integrated abstractions, services, and name spaces common to all three environments, mapping them into domain-specific mechanisms and internal names. Netbed's operating system-like services include node and link allocation and naming, scheduling and idle experiment preemption, experiment "swapping," and disk image loading.

Given these services, an analogy between an experiment and a Unix process seems natural. This metaphor illustrates the life cycle of an experiment and Netbed's role in automating and controlling the procedure. The *ns* specification serves as the "program text," which Netbed compiles 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, Netbed provides interfaces and tools for experiment control and interaction. Finally, Netbed may preempt and swap out an experiment. Because Netbed gives experimenters run-time control over node and link characteristics and an ability to interpose traffic-shaping and monitoring nodes, we view the system as a virtual machine for heterogeneous node, link, and topology allocation and control. While traditional virtual machines target an architecture's instruction set, Netbed instead abstracts the network.

The analogy is not merely cosmetic; experiments derive key benefits from Netbed's design, namely automation and time- and space-efficiency. Experiment creation involves a large number of steps including, for example, configuring network interfaces and routing tables, installing operating systems, exporting file trees, and administering user accounts. Netbed removes the tedium of manual configuration through automation. Netbed 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. Netbed performs experiment creation and termination in a few minutes, enabling an interactive style of use. Attention to efficiency of disk reloading, resource allocation, and experiment creation maximizes time spent executing experiments and minimizes effort expended configuring them.

This paper makes the following contributions:

- It introduces the notion of a virtual machine for controlled network experimentation and shows how it integrates heterogeneous resources.
- It outlines the key obstacles to virtual machine efficiency and how they were overcome.
- It shows that Netbed's automation, efficiency, and services inspire qualitatively new methods of experimentation.
- It provides data validating Netbed's emulation capabilities.

Section 2 continues by outlining the heterogeneous resources managed by Netbed. Section 3 outlines the life cycle of an experiment, using the virtual machine analogy to describe the system's design, and Section 4 shows the benefits of this approach. Section 5 details the challenges overcome by Netbed's experiment services, including the mapping of virtual to physical resources and disk loading, and their efficiency. Section 6 validates the emulation facilities. Section 7 illustrates unique experimental techniques facilitated by Netbed. Finally, related work is addressed in Section 8 and Section 9 concludes.

2 Resources

As its original name, "Emulab," suggests, Netbed was conceived as an emulation platform. Through its flexible design, it has evolved to support a diverse set of physical node and link types. These resources are virtualized in the sense that they may be allocated and controlled largely independently of their physical realization.

Local-Area Resources: Netbed software currently controls two clusters: one at the University of Utah comprised of 168 PCs and another at the University of Kentucky containing 50 PCs. The two sites are configured in a nearly identical fashion. Any of these nodes can function as an edge node, a traffic generator, or a router. 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 node, local memory and disk provide ample room for computation and logging of monitoring data.

All local nodes are connected using high-end switches that function as a "programmable patch panel." To support arbitrary and isolated topologies and to provide security to Netbed users, we employ Virtual LANs. A VLAN is a switch technology that restricts traffic to the subnet defined by its members. We have verified empirically that our switches provide inter-VLAN performance isolation, in the face of both traffic and control operations (VLAN creation, deletion, and modification).

Netbed's local nodes and wealth of available bandwidth can be configured into switched LAN topologies. This, coupled with its rapid and automated configuration of operating systems, makes Netbed an attractive platform for kernel development and research within local-area networks. Root privileges, remotely accessible consoles, and remote power cycling help make kernel development convenient.

Emulated Resources: Netbed uses Dummynet and VLANs to emulate wide-area links within the local-area environment. A Dummynet node is automatically inserted between two physical nodes and enforces queue and bandwidth limitations, introducing delays and packet loss. Dummynet nodes act as Ethernet bridges and are transparent to experimental traffic.

Distributed Resources: Netbed integrates both the

MIT-owned testbed nodes first used for the RON [4] research, as well as nodes contributed by other organizations that run our special CD-based Unix configuration. These resources today provide Netbed with approximately 40 nodes at 30 different sites around the world, including nodes connected via Internet2, DSL, and cable modems. These nodes are valuable to experimenters performing Internet measurement or who require the characteristics of a live network. Experimenters may request a random set of nodes, specific nodes, nodes having a specific class of network connection (e.g., via a cable modem), or nodes connected via specified latencies, bandwidths, and loss rates. In the latter case, Netbed provides a best-effort mapping of a user-specified virtual topology onto physical distributed nodes.

Distributed nodes support many of Netbed's key features, including account establishment and automated traffic generation, subject to their particular policies and mechanisms. For example, distributed nodes typically have only one network interface, so do not have a physically separate control network. Due to their scarcity, by policy—not limitation of mechanism—distributed nodes currently are shared; multiple experiments may use a node simultaneously. Netbed provides some isolation between experiments through the FreeBSD Jail [18] mechanism, which provides a primitive form of virtual machine and restricts root privileges. Our modifications to Jail provide access to raw sockets, while preventing processes from spoofing IP addresses. Multiplexing is supported by providing a (currently fixed) number of jailed virtual machines per node. Extending this mechanism to provide fair sharing of CPU, memory, and network resources is a subject of future work.

Netbed provides flexibility in specifying interconnections between distributed nodes. By default, the nodes retain full, unmediated access to the Internet. However, if links are specified between the nodes, Netbed sets up IP tunnels so that distributed nodes can use "private" IP addresses. In conjunction with Netbed's automated routing setup, this creates an overlay network configured to the experimenter's specifications. These tunnels also allow transparent communication between distributed nodes and experimental interfaces on local nodes, so that networks can contain both Internet and emulated links. Thus, distributed nodes may be treated the same as local nodes with respect to traffic generation, routes, and IP addresses.

Simulated Resources: Netbed integrates simulation through *ns*'s emulation facility, *nse* [10], allowing simulated nodes, links, and traffic to interact with application traffic. Though simulation abstracts detail [15, 11], it can provide scalability beyond the limits of physical resources; many virtual simulated nodes can be multiplexed on one physical node.

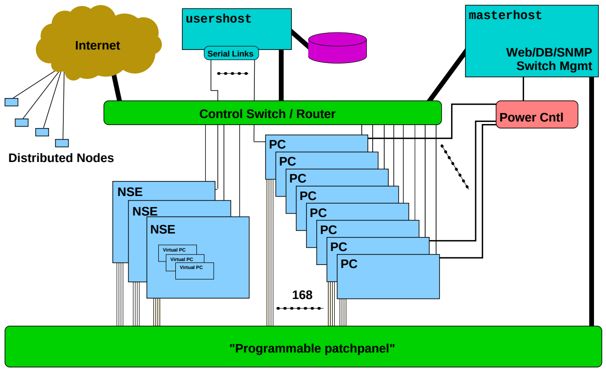

Figure 1: Netbed Architecture

Netbed's deployment of *ns* brings a wealth of simulation infrastructure to emulated and distributed experiments, including *ns*'s rich and diverse protocol suite, varied statistical models, and support for wireless devices. *nse* can also be used to simulate a large-scale network within emulation. The close interaction between simulation and live protocols presents an opportunity to validate *ns*'s abstractions.

Planned Extensions: Plans are underway to integrate additional virtual resource types: we are constructing a WAN emulator based on the Intel IXP1200 network processor [17] that provides improved features and performance over Dummynet. Second, we plan to control and configure ModelNet [42] through Netbed's existing interfaces.

3 Experiment Life Cycle

An experiment is Netbed's central operational entity. It represents a network configuration, including links and VLANs; node state, including operating system images; and database entries, including event sequences. The intended duration of an experiment ranges from a few minutes, to many days, to months or years on distributed nodes. This section follows the life cycle of an experiment to illustrate Netbed's operation and further develop its role as a virtual machine for network experimentation.

The Netbed virtual machine is architected around interacting state machines, monitored by a state management daemon. A primary state machine represents the experiment, while subsidiary state machines handle node allocation, configuration, and disk reloading. The state daemon catches illegal or tardy state transitions. For example, if a node hangs while rebooting, the state daemon times out and attempts an alternate reboot mechanism. This approach copes reasonably well with the reliability challenges of large-scale distributed systems which are composed of often unstable commodity hardware, but further work on reliability is needed.

3.1 Accessing Netbed

To minimize administrative overhead, Netbed employs a hierarchical structure for authorization: To begin a new *project*, a "leader," e.g., a faculty member or senior student, submits a simple web form. Once the project has been approved by Netbed staff, accountability and ability to authorize other project members are delegated to the project leader. The web interface then serves as a universally-accessible portal to Netbed, through which an experimenter may create or terminate an experiment, view the corresponding virtual topology, or configure node properties.

After experiment creation, experimenters may log directly into their allocated nodes, or in to `usershost`, depicted in Figure 1, which serves as a centralized point of control. This node is also `fileserver`, which stores operating system images, exports home and project directories to local nodes via NFS and to distributed nodes via SFS, the Secure File System [20]. `masterhost` is a secure server for many of our critical systems, including the web server, database, and switch management.

3.2 Specification

Just as program text is the concrete specification of a run-time process, an *ns* script written in Tcl configures a Netbed experiment. This choice facilitates validation and comparison since *ns*-specified topologies, traffic generation, and events can be reproduced in an emulated or wide-area environment. For the large community of researchers familiar with *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.

Emulated nodes and links enjoy full implementation transparency. By default, links specified in the *ns* experiment file are realized as interposed Dummynet nodes. To instead incorporate distributed nodes, an experimenter need only specify an appropriate node type. For example, Figure 2 requests an Internet-connected node by specifying a `pc-inet` hardware type. A simulated topology can be embedded within an emulated topology by wrapping standard *ns* syntax in a `make-simulated` block, a Netbed-specific construct.

Any constant bit rate traffic flow identified via standard *ns* syntax automatically instantiates traffic sources and sinks using the TG Tool Set [21]. Simulated FTP and Telnet flows are rendered using *ns*'s emulation facility, *nse*. This mechanism injects traffic generated by models, such as the tcplib telnet distribution, into a live network. Such cross traffic is important for studying protocol behavior in the face of congestion.

Netbed defines a small set of *ns* extensions, including procedures to configure a node's operating system and to specify its hardware type. These procedures are not required; Netbed supplies default behavior in their absence. A stub library defines null procedures so that the same script may be executed on Netbed and within *ns*.

```
set ns [new Simulator] # Create the simulator
source tb_compat.tcl   # Add Netbed commands
$ns rtproto Static     # Netbed computes routes

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 on local node
tb-set-hardware $dest pc-inet  # Request distributed node

$ns run                        # "run" on Netbed
```

Figure 2: An *ns* file showing a linear topology with routing and a distributed node

Program objects are a Netbed-specific *ns* extension that provides a rudimentary remote execution facility. A program object is associated with an *ns* node in the script and attaches arbitrary applications to the corresponding local node. It may be independently controlled during an experiment's execution. Program objects are currently not available on distributed nodes, until we finish securing the distributed event system.

Experimenters unfamiliar with *ns* syntax may create topologies graphically via a Java GUI, which generates an *ns* configuration file. Alternatively, a standard topology generator such as GT-ITM or BRITE may be used to generate an *ns* script. This highlights one of the primary benefits of integration: application of tools intended for one experimental domain, in this case simulation, to another.

3.3 Parsing

A traditional compiler is separated into front and back ends whose interactions are mediated by an intermediate representation. This aids portability since the same front end can be reused with back ends supporting different hardware architectures. Since Netbed targets multiple, heterogeneous physical resources simultaneously, it uses an analogous split-phase style of compilation. A database serves as the shared repository between the front-end Tcl/*ns* parser and resource-specific back-end mechanisms. Thus, a single experiment may incorporate simulated, emulated, and wide-area links without requiring excessive resource-specific knowledge in the specification language or front-end parser.

Netbed's 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 and Tcl primitives. Netbed executes the experiment configuration script in the context of these new definitions. Unrecognized *ns* commands output a warning, while *ns* syntax configuring links and traffic endpoints triggers the overloaded procedures. *ns*-specified event generation is performed at this time, storing the events in the database. Therefore, *ns*-specified events are static and have a (large) limit on their number.

Both overloaded and Netbed-specific procedures populate the database, which also stores information about hardware, users, and experiments. The database presents a consistent abstraction of heterogeneous resources to higher layers of Netbed and to experimenters. For example, the front-end database representations 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 across the Internet between distributed nodes.

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 linker-directed name binding. For overall simplicity, resources are currently allocated on demand rather than reserved by experimenters in advance.

Netbed uses general combinatorial optimization techniques to perform resource allocation. The algorithms map a target configuration, stored in the database, onto available physical resources. Such a mapping respects the interconnections of the virtual topology, including their latency, bandwidth, and loss rates. As further explained in Sections 5.2 and 5.3, we use separate algorithms for local and distributed nodes due to their differing constraints. The mapping program for local nodes, `assign`, uses simulated annealing, while the `wanassign` program uses a genetic algorithm for distributed resources. Based on the output of `assign` and `wanassign`, Netbed reserves nodes and links and updates the database with resource mappings and user-supplied parameters.

Although within an experiment we follow our principle of conservative resource allocation, we've found it impractical to do so between experiments on local nodes. We currently have only 2 Gbps inter-switch bandwidth, much of which is theoretically consumed by single experiments, preventing other experiments from mapping successfully. However, our traffic monitoring has shown that, in practice, experiments rarely use their allocated inter-switch bandwidth. Therefore we have adopted a policy of over-reserving these bottleneck links while continuously monitoring them for high bandwidth use. Thus far, that has never occurred.

Occasionally, there is a need to dynamically change node membership in an experiment. This can happen, for example, if a node fails and must be replaced, or if nodes are no longer needed because of a change in application demands. Netbed supports the dynamic addition or removal of nodes in any active experiment, and can graft added nodes into LAN-connected topologies.

To ensure consistent naming across instantiations of an *ns* configuration, Netbed virtualizes IP addresses 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 Netbed. Netbed deterministically names nodes and links for consistency across experiment creations.

3.5 Node Self-Configuration

Node configuration is driven by the nodes themselves, but entirely controlled by state stored centrally in the database. This is accomplished in a manner reminiscent of Unix dynamic linking and loading. A traditional dynamic linker is responsible for establishing the proper context for a process, loading it, and then invoking it. Netbed applies this strategy at the node level to achieve distributed self-configuration, which includes obtaining a host name, loading a disk image, and executing experiment startup scripts.

Intelligent *node state* management is crucial in realizing our robustness and security goals. Nodes are kept free of persistent configuration state; their memory and local disks are considered volatile soft state. This allows an experiment to be "swapped out" and its resources reclaimed. If experimenters wish to retain local disk modifications, such as kernel revisions, they can easily save an image of their disk on persistent store. A reference to the image is stored in the database and becomes hard state. While an experiment is swapped out, Netbed stores its virtual topology, host name, and general setup in the database. "Swap in" reconstitutes this hard state on an equivalent set of physical resources and brings the node to a fully-known state.

For local nodes, Netbed ensures that a clean disk image is installed on every node before experiment swap-in or creation. Then, in parallel, Netbed attempts to reboot all the nodes using increasingly aggressive techniques. First, it issues a `reboot` command via `ssh`; any nodes that fail to boot in a timely manner are sent a secure authenticated "ping of death"; should that fail, they are power-cycled. Nodes boot using Intel's PXE [34] network bootstrap protocol. Each node's PXE BIOS contacts `masterhost`, which loads a first level kernel as directed by the database. This first level kernel might be a fast disk image loader, a memory file system-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 [12] kernel. This multi-phase approach permits flexible configuration and customization of the OS that runs on each node. The system then waits for the nodes to come back up. If a node does not come up in a timely manner, one more attempt is made; if it still fails, the entire experiment swap-in fails. To improve resilience, over-allocation of nodes is an obvious avenue for future work. It is not entirely straightforward, due to topological constraints and heterogeneous node types.

Distributed nodes use an analogous disk loading mechanism. Each time a distributed node reboots, it does so from a CD-ROM which then negotiates with `masterhost` to, if necessary, securely apply software updates or reload the disk over the network. On each distributed node, Netbed instantiates a new Jail in a known initial state, analogous to the known initial state of a local node after disk loading and booting. In addition, a Jail can be "powered off" by terminating it or "rebooted" by restarting it.

Once a node or Jail has booted, our initialization sequence invokes a node configuration script that uses a program called the Testbed Master Control Client, TMCC, to securely communicate with a daemon on `masterhost` that fronts the database. Using this script and TMCC, a node obtains and initializes its hostname, experimental network IP addresses, routes, software packages, user accounts, and other configuration information. Local nodes NFS-mount the appropriate project tree and users' home directories from `fileserver`; in the wide-area, SFS is used instead.

3.6 Experiment Control

Traditional operating systems provide signals as a rudimentary form of control over local processes. Whereas users often start, stop, and resume processes, experimenters want to start, stop, and resume traffic generators and network monitors. To support dynamic experiment control, Netbed uses an event system to extend the notion of signals across sets of nodes and links. This facility closely mirrors the style of event schedulers found in network simulators. Just as simulation allows experimenters to manipulate link characteristics at prescribed times, so too can experimenters dynamically change latencies, bandwidths, and loss rates on emulated links. For example, to bring down a link named `link0` 10.5 seconds after experiment creation, a script would specify: `$ns at 10.5 "$link0 down"`.

Our event system is built on top of Elvin [38], a publish/subscribe system that supports federation. Static events are extracted from the database and fed into Elvin at experiment creation time. Dynamic events may be created through library interfaces and a command-line tool. Current clients of the event system include traffic generators, a WAN emulator control agent, a general remote execution facility, and Netbed's own management programs.

The event system is used extensively on local nodes but sparingly on distributed nodes, due to its current insecure deployment. Well-known solutions exist to secure the system; we are exploring a number of them, including using Elvin's "security keys," which limit the exchange of subscriptions and events to specific producers and consumers.

The event system controls high-level abstractions as defined in the *ns* configuration file, including links, nodes, and program objects. If experimenters were restricted to such high-level interfaces and tools, Netbed would limit the granularity of their control. Therefore, to the extent allowed by local policy, Netbed provides low-level and open access to resources, including root privileges on local nodes and Jail-restricted root privileges on distributed nodes. Of course, with such privileges experimenters can unwittingly corrupt their resources. Netbed's ability to quickly restore an experiment's hard state from the database and reload disk images makes it easy to recover from such accidents.

Root access on local nodes has proven to be an especially valued aspect of control, since it enables experiments requiring kernel modifications or access to raw sockets. To maintain security and isolation in the face of root access, Netbed prevents MAC and IP spoofing on local nodes through switch mechanisms. Since privileged access is mediated by Jail on shared, distributed nodes, these issues are not a concern there: though a process "in jail" can access raw sockets, it can only bind to its assigned IP address. This gives experimenters access to tools such as `tcpdump` and `traceroute`, without exposing insecurities.

Since the local nodes currently in use have serial console lines, power controllers, multiple network interfaces, and are dedicated to an experiment, they provide additional control mechanisms. Each local node is connected to a separate control network, isolated from the networks that are used for experimental traffic. This separate network provides three important features: more reliable control, cleaner experimental data, and greater security. Unless a program requires the use of a display or mouse attached directly to the node, Netbed does not penalize remote experimenters—with only minor exceptions, remote users have as much control over these nodes as they do over desk-side machines. For example, node consoles are virtualized so that an experimenter need not be logged into the server that physically hosts the serial console lines. Instead, all consoles can be securely accessed from any Unix or Windows machine via a local telnet session, connected through a transparent application-level SSL tunnel. We find that most kernel developers, once they have tried it, prefer remote use of Netbed machines to using desk-side test boxes.

3.7 Preemption and Scheduling

Traditional operating systems preempt and schedule processes for better system throughput and CPU utilization. Because Netbed manages shared community resources, efficient utilization is also a priority. Local nodes currently use a conservative allocation policy: each virtual node is mapped to a separate physical node. Therefore, Netbed can preempt idle experiments on local nodes to reacquire physical resources and to satisfy "runnable" experiments. Distributed nodes typically run each virtual node within a Jail, and are not currently subject to preemption. This policy is used because an idle distributed virtual node consumes only a single Jail rather than an entire physical node, and additional OS resource accounting mechanisms would be needed to accurately detect idle virtual nodes.

Local nodes are often idle despite being assigned to experiments. Determining idleness in Netbed is difficult; the indicators used in standard clusters are not sufficiently sensitive, since activity may constitute something as simple as infrequent network probes. Netbed's idle detection system currently monitors three metrics: traffic on the experimental networks, use of pseudo-terminal devices, and CPU load averages.

To avoid inconveniencing users, we manually confirm idle indications with them before swapping out their experiments. With recent tuning of the idle detection heuristics, Netbed has not experienced false positives and appears to find all truly idle experiments. Since our current swapping mechanism preserves only hard state, users with experiments dependent on soft state may manually disable preemption. With planned future work in disk state saving, Netbed should be able to safely preempt such experiments.

When experimenter interaction is not required, Netbed can fully automate the experimentation process by scheduling batch experiments, which execute whenever resources become available. Batch processing allows an experimenter to iterate over a large problem space without manual interaction. It also helps accommodate large experiments that may only find sufficient resources at low-usage, inconvenient times. Such off-peak scheduling further improves Netbed utilization.

4 Improving Network Experimentation

While Netbed provides most of the benefits of emulation, simulation, and wide-area experimentation, it is more than a simple sum of services. Netbed's common set of tools and abstractions have important practical benefits for experimentation, including: automated and efficient realization of virtual topologies, efficient use of resources through time- and space-sharing, and increased fault-tolerance through resource virtualization.

The savings afforded by automated mapping of a virtual topology to physical devices removes a significant experimentation barrier. Our user experiments show that after learning and rehearsing the task of manually configuring a 6-node "dumbbell" network, a student with significant Linux system administration experience took 3.25 hours to accomplish what Netbed accomplished in less than 3 minutes. This factor of 70 improvement and the subsequent programmatic control over links and nodes encourage "what if" experiments that were previously too time- and labor-intensive even to consider.

Efficient use of scarce and expensive infrastructure is also important and a sophisticated testbed system can markedly improve utilization. For example, analysis of 12 months of Netbed's historical logs gave quantitative estimates of the value of time-sharing (i.e., swapping out idle experiments) and space-sharing (i.e., isolating multiple active experiments). Although the behavior of both users and facility management would change without such features, the estimate is still revealing. Without Netbed's ability to time-share its 168 local Utah nodes, a testbed of 1064 nodes would have been required to provide equivalent service. Similarly, without space-sharing, 19.1 years, instead of one, would be required. These are order-of-magnitude improvements.

Netbed virtualizes node names and IP addresses such that equivalent nodes can be used interchangeably. For example, when an experiment is swapped in, it need not execute on the same set of physical nodes. Any nodes exhibiting the same properties and interconnection characteristics are suitable candidates. The flexibility to allocate from an equivalence class provides fault tolerance. If a node or link fails, an experimenter need not wait until the node or link partition is available again, but may instead re-map the experiment to an equivalent set of machines. This feature is valuable wherever node or link failures are anticipated, such as within large-scale clusters or wide-area networks.

5 Key Services and Evaluation

Much of *ns*'s popularity and power result from the flexibility it gives experimenters to efficiently change parameters and network scenarios. Netbed aims to bring a similar level of control and ease of use to emulated and

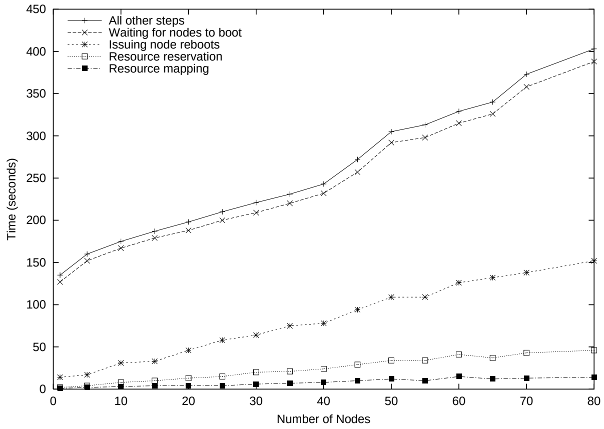

Figure 3: Time to create an experiment without disk loading. Times shown are cumulative, i.e., the difference between adjacent lines represents the time for that step.

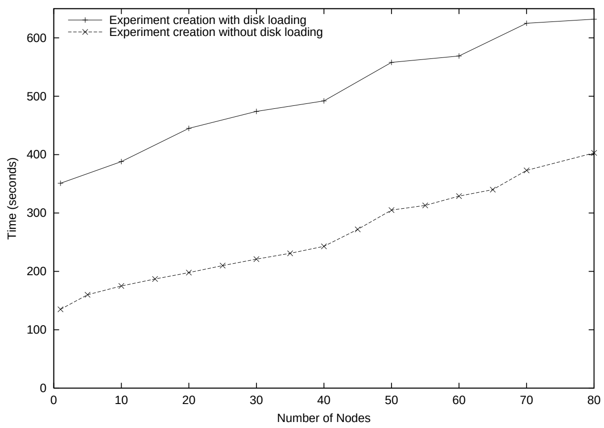

Figure 4: Time to create an experiment with disk loading. Time without disk loading from Figure 3 is also shown for comparison; note that the y-axis scale is different here.

wide-area experimentation, through automation and efficiency. In this section we describe the main challenges to Netbed's efficiency, and evaluate how well Netbed meets those performance challenges. These challenges include experiment creation and swapping, disk loading, mapping of virtual resources to local and distributed physical resources, and multiplexing simulated nodes.

5.1 Experiment Creation and Swapping

This subsection quantifies the time spent in experiment creation, which is comprised of parsing, global resource allocation, and local self-configuration, as described in Section 3. These results apply only to local resources; since distributed nodes are typically shared resources, Netbed does not routinely reboot them or re-install disk images on experiment creation. As shown in Figures 3 and 4, disk loading and node rebooting dominate experiment creation time. Therefore, configuration of distributed nodes is lightweight and not examined here.

The top line in Figure 3 shows the total time to create typical experiments. The duration of experiment creation is essentially equal to the swap-in duration, since the one-time expenses unique to experiment creation are insignificant compared to the cost of mechanisms shared by both, such as node rebooting. A single-node experiment takes 135 seconds. The majority of this time is spent rebooting the node and waiting for it to finish booting. As experiment sizes grow, creation time remains linear, with a marginal cost per node of approximately 3.4 seconds. Throughout the process, Netbed exploits parallelism as much as possible. For example, although it takes non-negligible time, VLAN setup does not contribute to creation time because it occurs in parallel with the longer node reboot stage.

Figure 3 also breaks out the costs of the most time-consuming stages of experiment creation, in the order those steps occur. The bottom line represents the time taken by `assign` to map physical resources. The next line is for reservation of those resources, which turns out to be dominated by reassigning serial console lines and logs. The next line is for issuing reboots to the nodes. They are rebooted in parallel, with a ten second pause every eight nodes so as not to over-stress network resources and lose too many control-related UDP packets, typically manifested by nodes failing to boot.[1] Finally, in the slowest step, Netbed waits for all nodes to come back up. The PC's BIOS is the biggest culprit; average time spent in the BIOS was 55 seconds for the nodes used in this experiment. Netbed also has 40 nodes that spend only 20 seconds in the BIOS, but in order to achieve consistency up to large scales, we limited these experiments to the more numerous nodes.

Figure 4 shows the additional expense of automatic disk loading, performed when an experimenter requests a custom disk image. Since our default dual-boot FreeBSD/Linux disk images prove sufficient for most experimenters, the majority of experiments do not incur this cost. Though much of the added time comes from transferring and writing the new disk image, a significant amount comes from rebooting each node twice (once to enter the disk loader, and again into the newly-loaded operating system). Although the absolute time for experiment creation is higher when loading disks, it is similarly scalable; the marginal cost per node is comparable.

5.2 Mapping Local Resources

Netbed's local assignment phase must not only realize user-specified node types, features, link characteristics, and topologies, but must also respect the limitations of available bandwidth. That is, Netbed ensures that the physical hardware will support the emulated traffic flows

[1] The PXE ROMs use UDP and a fixed timeout that we cannot change; hence we are forced to work around the problem.

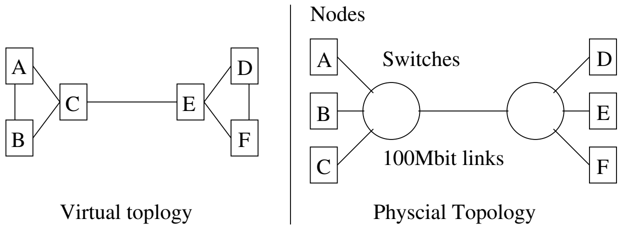

Figure 5: A trivial six-node partitioning problem

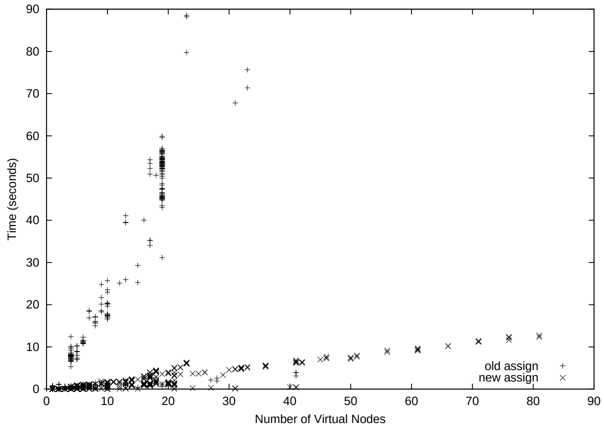

Figure 6: Performance and Scaling of `assign`

without introducing any bottlenecks, with their accompanying experimental artifacts. To map the desired virtual topology of Figure 5 onto the physical topology shown to its right, Netbed should 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 traffic across the inter-switch link.

This *testbed mapping problem* problem is trivial in this six-node example, but in the general case, is NP-hard (by reduction to the multiway separator problem or the minimum-degree graph partitioning problem [13]). In conjunction with aggressive abstraction techniques to reduce the search space, `assign` uses simulated annealing [16], a randomized heuristic algorithm, to map virtual nodes and links to local nodes and VLANs. In addition to satisfying the individual experiment's requirements, the algorithm also attempts to minimize the required inter-switch bandwidth and the number of involved switches, in order to promote efficient utilization of the cluster.

Netbed has kept detailed logs of every experiment submitted since June 2001. We analyzed the following 12 months' data, covering over 2000 experiments. Figure 6 shows that a reliable indicator of the difficulty of a mapping problem, as measured by the runtime of `assign`, is the number of virtual nodes the user requests. We added a general notion of resource equivalence classes to `assign` in December 2001; the strikingly bimodal distribution in the figure demonstrates the resulting improvements. Grouping nodes into equivalence classes greatly reduces the search space since `assign` need only search the small number of equivalence classes rather than the large number of nodes. The new version takes less than 13 seconds on even the largest topologies and less than 5 seconds for most experiments.

5.3 Mapping Distributed Resources

The distributed case has different constraints. First, the underlying physical nodes are treated as fully connected, via the Internet. Second, distributed nodes are fairly well characterized by the nature of their "last-mile" link, e.g., cable modem, commodity Internet, or Internet2. Therefore, Netbed assigns corresponding intuitive subtypes to distributed nodes, e.g., `pc-cable`, `pc-inet`, `pc-inet2`. This typing lets experimenters request virtual nodes by their type or subtype, rather than specify a particular topology connecting them. Netbed's generic resource assignment code, identical for both local and distributed resources, handles this common situation.

However, some experimenters may want more precisely-matched resources or a particular virtual topology. Netbed allows them to request a virtual topology with wide-area links of specific latency, loss, and bandwidth characteristics. They may assign weights to each of the three attributes, based on their perceived importance. Unlike the highly configurable local links in a Netbed cluster, connections between distributed nodes traverse the Internet through uncontrollable links. Therefore, our challenge is to map virtual nodes to physical resources such that the requested links best match the actual characteristics of the corresponding inter-node Internet paths. (Netbed's database is updated frequently with the measured latency and loss on the NxN paths, and occasionally updated with bandwidth measurements.)

This mapping is a variation of the NP-hard Quadratic Assignment Problem. To provide an efficient, best-effort solution, Netbed's `wanassign` is implemented as a genetic algorithm [39]. Possible solutions are scored based on how closely they match desired link characteristics. For each solution, a normalized sum of errors-squared is found for latency, loss rate, and bandwidth. A geometric mean of the three errors results in an overall score. `Wanassign` evolves its answer by propagating solutions with the least error.

We conducted two experiments to test `wanassign`'s performance. The first mapped a wide variety of virtual topologies onto a set of 16 physical, distributed nodes. We varied the number of requested nodes from 4 to 16 and the number of requested links from 4 to 120, examining 48 pairs from this set to present a cross section of experiment complexities. For each of these pairs, we ran hundreds of tests on automatically-generated topolo-

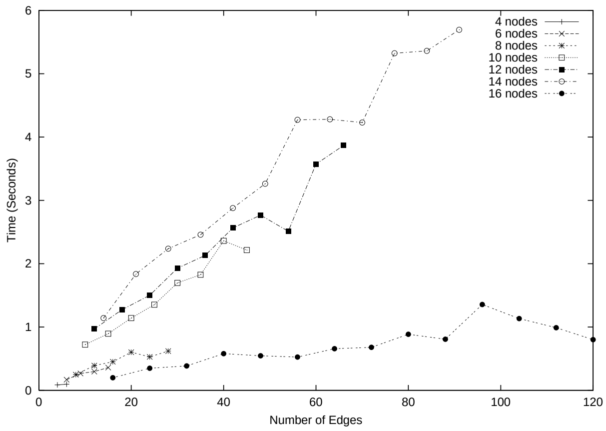

Figure 7: Average time for `wanassign` to find a solution for a variety of experimental topology complexities (node and edge counts).

gies. Figure 7 shows the average time to find a 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 algorithm does not contribute noticeably to the total experiment setup time, nor is it prohibitively slow for experiments involving most of the available nodes.

The second experiment explored further scalability, mapping a range of virtual topologies onto a synthetic set of 256 distributed nodes. All experiments requesting 32 virtual nodes, as well as all sparse topologies, mapped in a few minutes. For larger and denser topologies, up to 256 nodes and approximately 40 edges/node, mapping time ranged from 10 minutes to 2 hours. We expect to improve these results by an order of magnitude using the following three techniques: less stringent and more clever termination conditions; standard optimization techniques, in particular memoizing; and parallelizing the algorithm, which is practical in either a shared memory multiprocessor or on a cluster [39]. Finally, we expect major additional improvement to come from "binning" the nodes and links into groups with similar characteristics, dramatically reducing the search space.

5.4 Disk Reloading

An important feature of testbed control is the ability to reload the contents of node local disks automatically. This not only ensures node integrity, but also allows custom OS configurations. The two common approaches for achieving this goal are to load complete disk images [14, 32, 35] or to work through the file system to incrementally synchronize a target hierarchy with a reference copy (`rsync` [37], `Unison` [41]). There are five reasons for preferring disk imaging. (1) While sometimes more efficient in terms of network bandwidth, on our images, at least, the synchronization approach is slower. `rsync` takes over 50% longer to compare file timestamps on our typical image (80K inodes, 500MB data) than Netbed's disk loader takes to copy all the allocated blocks. Comparing hashes of file contents takes much longer. (2) Approaches that rely solely on file timestamps cannot be used for security reasons, as falsified timestamps allow modified files to corrupt the next experiment. (3) Approaches working through the file system cannot be used on corrupt target file systems, nor (4) to install custom OS's with unknown file systems. (5) Bulk disk imaging is scalable through multicast-based approaches. A third approach based on content hashes of blocks, as in LBFS [23], may be worth investigating.

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

Each node in a new experiment requires a clean disk. However, disk reloading remains the most time-consuming aspect of experiment creation and swap-in, even though we have reduced it to less than 100 seconds. Netbed's current policy reloads each node's disk with the default image containing both FreeBSD and Linux. This works well since most users request one of these OSes, and if there are sufficient free nodes, the disks are reloaded in the background and are immediately available for the next swap-in.

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, during which time the user requests a similar number of nodes for their next experiment. To avoid this anomaly we currently pace the reloading of freed nodes, instead of reloading them all at once. For security reasons, we allow an un-reloaded node to be assigned only to an experiment in the same project as the node's previous experiment. This approach, however, has robustness vulnerabilities, since the disk's soft state will not be reinitialized, and may have been changed by the previous experiment—though that is rare.

Users can also specify an alternate disk image or partition. In this case, the background disk reloading is wasted, as the default image is overwritten by the user's custom one. Automated analysis of historical and ongoing experiment creation and swap patterns is one promising 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 file system-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; we will soon add Windows XP. 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.

The Netbed disk loader, termed "Frisbee" (the flying disk) uses three main techniques to improve performance from Netbed's first loader, which took 29 minutes per image. First, it carefully overlaps block decompression and device I/O. Second, it uses a domain-specific compression algorithm that uses file system information to identify which parts of the disk need to be saved; it compresses these portions with standard `zlib`-based compression. Third, it uses a custom reliable multicast protocol to deliver compressed images to clients, dramatically reducing the required server bandwidth and improving scalability. The result is that a standard FreeBSD image requires 88 seconds to load onto a single node. It also scales well; 80 nodes can be loaded simultaneously with an average of only 97 seconds per node, and with all nodes completing in 117 seconds. Frisbee's performance also compares favorably to commercial tools; in our initial tests, it was able to load our standard Linux image on a single node in 77% of the time taken by Norton Ghost.

The compression algorithm exploits the fact that many disks contain large swap partitions and mostly-empty file systems, and looks at partition types and file system free-block lists to find these. 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 file system-specific compression enables the Frisbee decompression program to optionally skip, rather than zero, the free file system blocks when writing the disk image. This turned out to be very important: once we had done standard compression and implemented a multicast mechanism, writing to the disk became the bottleneck. For the aforementioned FreeBSD disk image, Frisbee wrote 550MB of actual decompressed data rather than the full 3GB.

5.5 Scaling of Simulated Resources

Experiments can leverage simulation to multiplex simulated nodes onto a single physical node and to obtain greater scalability. Since the simulator interacts with the physical world through *nse*, it must keep pace with real time. Its ability to do so is dependent on the rate of events that need to be processed, rather than the number of nodes or links per se. Towards achieving greater scale, we have made several improvements and contributed fixes to *nse*. We describe here a simple study that achieves greater scale through simulation.

An instance of *nse* simulated 2Mb constant bit rate UDP flows between pairs of nodes on 2Mb 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 Netbed nodes, again at a rate of 2Mb. 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.

More complex hybrid topologies exposed unanticipated routing behavior. Incorrect routing arises when an *nse* simulation, running on a multihomed host, relies on its kernel's routing tables. The solution required Netbed's global system perspective; it computes the overall routes, using Unix policy routing mechanisms (`ipfw` and `ipchains`) to control the packet routes.

6 Validation and Testing

This section validates Netbed's emulation capabilities through micro- and macro-benchmarks. Since Netbed is itself a complex and evolving distributed system, it requires continual testing and validation. This section therefore outlines a testing methodology intended to ensure Netbed's continued accuracy.

6.1 WAN Emulator Validation

There are two concerns with using off-the-shelf PCs and a general purpose operating system for emulation: first, machines must be able to keep pace when emulated links are operating at full speed; second, delays, bandwidths, and packet loss rates should be emulated accurately.

Emulation nodes in Netbed run a FreeBSD 4.6 kernel with Dummynet and polling device drivers. We run these kernels with a clock frequency of 10000HZ to allow sub-millisecond delay granularity, while the polling drivers reduce interrupt load and provide improved precision.

As a capacity test, we generated streams of UDP round-trip traffic between two nodes, with and without an interposed emulator node. The emulator node showed no adverse effects on 1518-byte packets; either configuration easily saturated a 100Mb link. With 64-byte packets, the two nodes exchanged 55000 packets (3.5MB) per second when connected directly versus 37000 packets

delay (ms)	packet size	observed Dummynet RTT	observed Dummynet stdev	observed Dummynet % err	adjusted Dummynet RTT	adjusted Dummynet % err	observed *nse* RTT	observed *nse* stdev	observed *nse* % err	adjusted *nse* RTT	adjusted *nse* % err
0	64	0.177	0.003	N/A	N/A	N/A	0.238	0.004	N/A	N/A	N/A
	1518	1.225	0.004	N/A	N/A	N/A	1.554	0.025	N/A	N/A	N/A
5	64	10.183	0.041	1.83	10.006	0.06	10.251	0.295	2.51	10.013	0.13
	1518	11.187	0.008	11.87	9.962	0.38	11.586	0.067	15.86	10.032	0.32
10	64	20.190	0.063	0.95	20.013	0.06	20.255	0.014	1.28	20.017	0.09
	1518	21.185	0.008	5.92	19.960	0.20	21.675	0.093	8.38	20.121	0.61
50	64	100.185	0.086	0.18	100.008	0.00	100.474	0.029	0.47	100.236	0.24
	1518	101.169	0.013	1.16	99.943	0.05	102.394	3.440	2.39	100.840	0.84
300	64	600.126	0.133	0.02	599.949	0.0	601.690	0.546	0.28	601.452	0.24
	1518	600.953	0.014	0.15	599.728	0.04	602.999	0.093	0.49	601.445	0.24

Table 1: Accuracy of Dummynet and *nse* delay at maximum packet rate as a function of packet size and link delay. The 0ms measurement represents the base overhead of the link. Adjusted RTT is the observed value minus the base overhead.

bandwidth (Kbps)	packet size	observed Dummynet bw (Kbps)	observed Dummynet % err	observed *nse* bw (Kbps)	observed *nse* % err
56	64	56.06	0.11	55.60	0.71
	1518	56.67	1.89	56.63	1.12
384	64	384.2	0.05	376.3	2.00
	1518	385.2	0.34	382.1	0.49
1544	64	1544.7	0.04	1444.5	6.44
	1518	1545.8	0.11	1531.0	0.84
10000	64	10004	0.04	N/A	N/A
	1518	10005	0.05	9659.6	3.40
45000	1518	45019	0.04	39857	11.43

Table 2: Accuracy of Dummynet and *nse* bandwidth as a function of link bandwidth and packet size.

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	0.819	2.37
	1518	0.803	0.3	0.820	2.50
2.5	64	2.51	0.4	2.477	0.92
	1518	2.47	1.1	2.477	0.92
12	64	12.05	0.4	11.88	1.00
	1518	12.09	0.7	11.89	0.91

Table 3: Accuracy of Dummynet and *nse* packet loss rate as a function of link loss rate and packet size.

(2.4MB) when joined by an emulator node. Since these are round trip measurements, the packet rates are actually twice the numbers reported.

To bound the accuracy and precision of emulation nodes, we performed a series of experiments using a representative range of delay, bandwidth, and packet loss rate values coupled with high packet rates for both large and small packets.

After establishing maximum emulation rates for large and small packets, we ran a series of tests using those packet rates with various delay, bandwidth, and loss rate values, measuring both accuracy and precision. The delay results are presented in Table 1. The 0ms rows represent the base overhead associated with interposition of an emulation node. These results seem to indicate, and further experimentation confirmed, that emulation node overhead is proportional to the packet size. As indicated in the "observed" column, small packets show noticeable error with delays less than 10ms and large packets suffer with delays less than 50ms. While both are tolerable for wide-area emulation, we can improve accuracy by adjusting delays to compensate for emulation overhead. As a first approximation, we scaled delays by the base overhead shown in the 0ms case. The adjusted results, shown in the "adjusted" column, are both accurate and precise.

To measure the bandwidth limiting capabilities of an emulation node, we used one-way traffic. A sender node sent packets through an emulation node to a consumer node, which calculated bandwidth. Results are summarized in Table 2.

Finally, using the same setup, we instead measured packet loss rates as observed by the consumer. Results are summarized in Table 3.

6.2 *nse* Validation

This section uses the methodology of Section 6.1 to validate the observed latencies, bandwidths, and loss rates induced by *ns*'s emulation facility, *nse*, against their expected values. *nse* runs on a FreeBSD 4.5 kernel at 1000HZ. The simulation is configured with two nodes and a duplex link connecting them. The physical node running *nse* interposes two other traffic-generating physical nodes. This setup mimics Section 6.1, differing only in packet rate. 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. Note that the actual capacity is twice this value due to the duplex link. With this capacity, we performed experiments to measure the delay, bandwidth and loss rates for representative values. The results are summarized in Tables 1, 2 and 3. Netbed's integration of *nse* is much less mature than its support for Dummynet. This is reflected in the larger relative error rates of *nse* bandwidth and loss rates with respect to Dummynet. Integrating *nse* has already uncovered a number of problems that have since been solved; as we continue to gain experience with *nse*, we expect the situation to improve.

	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 4: 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 intermixed slower links.

6.3 Validation Against a Wide-Area Network

This section validates Netbed's emulation mechanisms against a wide-area network: it compares two macro-benchmarks run on a set of live Internet nodes and then within a corresponding emulation. The first example also demonstrates the transparency of Netbed's heterogeneous resource specification and its ability to provide a best-fit mapping between requested wide-area links and live Internet links.

Distributed Multiplayer Game: This benchmark evaluates a derivative of DOOM on four network configurations, making at least four repeated runs on each. In these scenarios, five synthetic clients communicate using a simple protocol. At a target rate of 30 times per second, each client sends unicast packets to all other clients, doing so only after receiving all packets from the prior period. We specified the desired latency and bandwidth of the ten links comprising a fully-connected graph between the five clients.

The first configuration specified a node type of `pcvremote` to obtain wide-area "virtual" nodes. In this sense, virtual means the nodes may be multiplexed onto a single physical distributed node. Netbed's distributed mapping service, the genetic algorithm described in Section 5.3, found the best-matching fit from among the distributed nodes with available virtual node "slots."

The second configuration used the same link specification, but instead of mapping to the live Internet, requested emulation on local nodes and links. Making that switch to an entirely different experimental environment required changing only one line within a Tcl loop that set the node type. The third and fourth configurations were analogous to the first two configurations, but requested a few substantially slower links.

The results were similar between emulation and the live Internet, as presented in Table 4. 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 [7] to those obtained emulating the observed CAIRN delay and bandwidth characteristics with Netbed. Their application-level measurements of communication characteristics matched well [3]. Netbed offered two advantages over CAIRN: First, with Netbed's control, they were able to study the system-wide effects caused by varying network characteristics. Second, they were able to obtain a set of nodes of a consistent type.

6.4 Testing

Netbed presents unusual testing challenges: First, it is inherently coupled to physical artifacts which, unlike software state, can not be cloned. This makes full test and regression runs impossible. Second, 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 Netbed must run continuously, even as its software radically evolves.

We have countered with the following procedures. First, we have created a separate 8-node Netbed, Minibed. As an independent Netbed instance, Minibed is also important to our future work on federation.

Second, we have integrated support for testing throughout the Netbed 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 Netbed 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 Netbed database and use that hardware in conjunction with the duplicate database and software. This enables the test environment to affect this hardware, which is ignored by the "main" Netbed system. This feature is made possible by database-driven, node-specific redirection to alternate daemons and databases.

Third, we have developed a comprehensive regression test suite that is run nightly and optionally at compile time. However, we currently only systematically test for software bugs. To monitor Netbed accuracy, we are adding additional point tests as well as end-to-end tests.

7 New Experimental Techniques

This section showcases the novel experimental opportunities made possible by Netbed. The first case study capitalizes on Netbed's *ns* compatibility to automate comparison of emulated and simulated results. Other systems have leveraged a similar synergy between simulation and live experimentation [6], but required adoption of a non-standard programming interface. The second case study shows the importance of automation.

7.1 TCP Dynamics

Network simulators, such as *ns*, have proven invaluable in studying TCP behavioral dynamics [11]. Nevertheless, with its abstractions such as 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. Fortunately, the notion of "deviant behavior" [9] allows an experimenter simultaneously to gain confidence in the validity of simulation and the correctness of implementation. This case study leverages existing simulation experiments to drive emulated scenarios. This approach makes an existing corpus of test scenarios amenable to live experimentation. Thus, corner cases with known results can be applied as regression tests to real network stacks to evaluate their conformance.

The *ns* maintainers run nightly regression tests [24]. Netbed's 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 Netbed's flexibility in modulating a virtual network at various scales.

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 Netbed's event system and passes a list of target drop packets to the correct Dummynet node (we have extended Dummynet to drop packets by ordinal packet number). Again via the event system, the framework starts a program object to record packet traces and finally invokes the traffic generators.

Figure 8 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. They behave similarly; over 10 experiments FreeBSD 4.5 achieves a mean throughput of 50232Bps (standard deviation 4.09) and *ns* achieved 48090Bps. By contrast, we discovered that FreeBSD 4.3 does not retransmit until triggered by a timer expiration, which greatly degrades throughput. 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 subtle bugs that would be exceedingly difficult to detect and reproduce without Netbed's fine-grained control.

7.2 The Armada I/O Framework

Simulation allows an experimenter to effortlessly explore a large parameter space. Using Netbed's programmatic *ns* interface to loop over a configuration space and exercising its distributed event system to affect link characteristics, an experimenter has similar power over emulation. Oldfield and Kotz [30] used these techniques in evaluating Armada [29], a file system for computational grids. Armada's performance is highly dependent on link bandwidth, latency, and packet loss rate. The authors used Netbed's batch system to evaluate every possible combination of 7 bandwidths, 5 latencies, and 3 application parameter settings on four different configurations on a set of 20 nodes, performing a total of 420 different tests in 30 hours, averaging 4.3 minutes each.

8 Related Efforts

Network Emulation: ModelNet [42] is a new network emulation system focused on scalability. It uses a small gigabit cluster, running a much extended version of Dummynet, which is able accurately to emulate an impressively large number of moderate speed links. This core routes packets between applications running on additional "edge nodes." Applications can be multiplexed on edge nodes, without resource isolation. ModelNet shares some of Netbed's automatic configuration of physical resources by including tools to take a target topology specified in a high-level format and map it into ModelNet mechanisms; it provides the added capability of optionally distilling the topology to trade accuracy for scalability.

ModelNet emphasizes scalability through a high-performance implementation of emulated links. This contrasts with our emphasis on complete accuracy through conservative resource allocation, exposure of all resources (including link emulation mechanisms) to manipulation by experimenters, and integration of disparate techniques into a common framework.

ModelNet's core contributions are complementary to Netbed's; indeed, we intend to work together to integrate ModelNet into Netbed. This combination should bring Netbed's rich user interface and ease of use to ModelNet, while adding a scalable new mechanism to those available through Netbed's common abstractions.

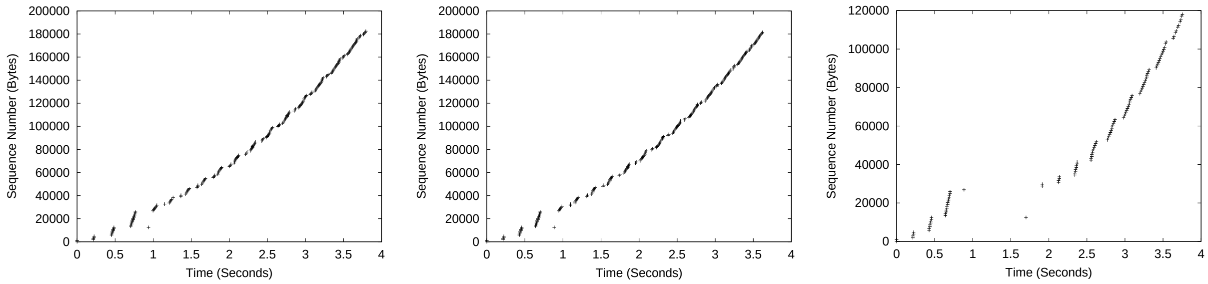

Figure 8: New Reno One Drop Test: (a) *ns* (b) FreeBSD 4.5 (c) FreeBSD 4.3 (different y-axis scale)

Yet another link emulation technique is trace modulation [27], which recreates observed end-to-end characteristics of a wireless network. Interposing trace modulation instead of Dummynet would bring wireless emulation to Netbed.

There have been a large number of single-node network emulation efforts. These include hitbox [1], ONE [2], NIST Net [26], and Rice's support for evaluating their OS optimizations [31]. Another category is represented by the "Orchestra" fault-injection system [8]. With a few exceptions, these single node emulators were tailored for a specific research application. A few multi-node network emulators have been planned or built, but only for specific projects. One of the earliest and largest was a particular configuration of 12 workstations at USC in 1994, used to study TCP Vegas [1]. They cite an emulator effort at Bell Labs [19], which apparently started to build a more general emulator.

Distributed Network Testbeds: The "Access" vision [5] originated the idea of a set of small testbeds, distributed over dozens of sites. The Access vision overlapped with Netbed in our shared emphasis on completely replaceable node software and our operational model of a Web-accessible master control host. However, Access did not intend to provide an emulation facility nor did it intend to offer integration. They did recognize a need we identified only later, for real wide-area links for some experimenters.

PlanetLab [33] is a new effort that plans to provide to researchers a large number (1000) of centrally administered, geographically distributed PCs, along with a modest number of clusters. This testbed, currently in its initial phase, would be used for arbitrary research, yet provide a transition avenue to production deployment of overlay network services. Unlike Netbed, PlanetLab plans to emphasize the design of APIs and services that can be shared by higher-level services.

Netbed's distributed node support is similar to what is planned for PlanetLab's next phase. Although with a different primary goal, PlanetLab's notion of a "service" across a "slice" of PlanetLab nodes is similar to Netbed's "experiment," since Netbed experiments can be of arbitrary duration. An experiment is richer in that it contains flexible notions of topology, swapping, hard state, soft state, and optional shared persistent storage. Like Netbed, PlanetLab's current testbed management is centralized. Their future plans emphasize unbundled management in order to facilitate research into management; our plans emphasize federation, in order to achieve greater scalability and another route to overlay service deployment. In fact, we are jointly exploring providing access to PlanetLab through Netbed's interface.

Network Simulators: Network simulators successfully isolate protocol dynamics but may do so at the expense of accuracy. Therefore, results from simulators may not be valid indicators of deployed performance [11]. Brakmo and Peterson [6] highlight differences between simulated and implemented TCP protocols. Their *x*-kernel-based simulator avoids inaccuracies by using actual protocol code, as does recent work integrating Click elements into *ns* [25]. However, both systems rely on non-standard protocol implementations.

Cluster Management: Through its virtualization of cluster hardware and software, "Emulab Classic"—Netbed's cluster-based emulation portion that has been in public production use since October 2000—is relevant far beyond network experimentation. In its flexible and efficient allocation of all hardware and software resources (except shared persistent storage) and ability to isolate virtual sub-clusters, Emulab overlaps many or most of the low level facilities in "computing utility" efforts such as IBM's Océano [28], HP's Utility Data Centers, and Duke's Cluster-on-Demand [22]. Netbed has the flexible interfaces and all the needed mechanisms—including dynamically adding or removing nodes in an experiment—to support reconfiguration by Service Level Agreements or by sub-cluster management systems.

9 Conclusion

Acting as a virtual machine for network experimentation, Netbed virtualizes and integrates simulated, emulated, and distributed nodes and links. Through a rich user interface, efficiency, and automation, Netbed enables qualitatively new kinds of experimentation across these mechanisms.

References

[1] J. S. Ahn et al. Evaluation of TCP Vegas: Emulation and Experiment. In *Proc. of SIGCOMM '95*, pages 185–195, Aug. 1995.

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

[3] Y. Amir, C. Danilov, M. Miskin-Amir, J. Stanton, and C. Tutu. Practical Wide-Area Database Replication. Technical report, Johns Hopkins University, 2002.

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

[5] 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/.

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

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

[8] S. Dawson et al. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In *Proc. FTCS '96*.

[9] 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.

[10] K. Fall. Network Emulation in the Vint/NS Simulator. In *Proc. IEEE ISCC '99*, 1999.

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

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

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

[14] Symantec Ghost. http://www.symantec.com/sabu/ghost/.

[15] J. Heidemann et al. Effects of Detail in Wireless Network Simulation. http://www.isi.edu/˜johnh/PAPERS/Heidemann00d.html.

[16] L. Ingber. Very Fast Simulated Re-Annealing. *Journal of Mathematical Computer Modelling*, 12:967–973, 1989. http://www.-ingber.com/asa89_vfsr.ps.gz.

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

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

Acknowledgments: Many thanks to Chris Alfeld, Dave Andersen, and Kirk Webb for discussion, design, and code, to Dave for MIT's RON testbed nodes, to Russ Christensen, Alastair Reid, Tim Stack, and Parveen Patel for running experiments, to Eric Eide for editing, to ex-Fluxers who helped build the Emulab cluster, to John Regehr, Robert Morris, and the anonymous reviewers for comments, to Jim Griffioen for bravely being the first to bring up another cluster, to Ron Oldfield for the Armada results, to Nicolas Christin for suggesting validating TCP dynamics, to Mark Tinguely for clarifications on FreeBSD TCP, and to our many users. Finally, we are grateful to our many sponsors, especially NSF under grants ANI-0082493 and ANI-0205702, Cisco Systems, and DARPA/Air Force under grant F30602-99-1-0503.

[19] 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.

[20] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of SOSP '99*, December 1999.

[21] 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. http://www.postel.org/tg/.

[22] J. Moore and J. Chase. Cluster On Demand. Technical Report CS-2002-07, Duke University, Dept. of Computer Science, May 2002.

[23] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-bandwidth Network File System. In *Proc. 18th SOSP*, Oct. 2001.

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

[25] M. Neufeld, A. Jain, and D. Grunwald. Nsclick: Bridging Network Simulation and Deployment. In *Proc. MSWiM 2002*.

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

[27] B. D. Noble et al. Trace-Based Mobile Network Emulation. In *Proc. of SIGCOMM '97*, Sept. 1997.

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

[29] 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*, May 2001.

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

[31] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A Unified I/O Buffering and Caching System. In *Proc. 3rd OSDI*, Feb. 1999.

[32] Partition Image. http://www.partimage.org/.

[33] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proc. HotNets-I*, Princeton, NJ, Oct. 2002.

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

[35] Rembo Technology. BpBatch. http://www.bpbatch.org/.

[36] L. Rizzo. Dummynet and Forward Error Correction. In *Proc. of the 1998 USENIX Annual Technical Conf.*, June 1998.

[37] rsync. http://rsync.samba.org/.

[38] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *Proc. AUUG '00*, June 2000.

[39] R. Tanese. The Distributed Genetic Algorithm. In *Proc. ICGA '89*. Morgan Kaufmann, 1989.

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

[41] Unison. http://www.cis.upenn.edu/˜bcpierce/unison/.

[42] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. 5th OSDI*, Dec. 2002.