Large-scale Virtualization in the Emulab Network Testbed

Mike Hibler Robert Ricci Leigh Stoller Jonathon Duerig
Shashi Guruprasad† Tim Stack† Kirk Webb† Jay Lepreau

University of Utah, School of Computing
www.emulab.net www.flux.utah.edu

Abstract

Network emulation is valuable largely because of its ability to study applications running on real hosts and "somewhat real" networks. However, conservatively allocating a physical host or network link for each corresponding virtual entity is costly and limits scale. We present a system that can faithfully emulate, on low-end PCs, virtual topologies over an order of magnitude larger than the physical hardware, when running typical classes of distributed applications that have modest resource requirements. This version of Emulab virtualizes hosts, routers, and networks, while retaining near-total application transparency, good performance fidelity, responsiveness suitable for interactive use, high system throughput, and efficient use of resources. Our key design techniques are to use the minimum degree of virtualization that provides transparency to applications, to exploit the hierarchy found in real computer networks, to perform optimistic automated resource allocation, and to use feedback to adaptively allocate resources. The entire system is highly automated, making it easy to use even when scaling to more than a thousand virtual nodes. This paper identifies the many problems posed in building a practical system, and describes the system's motivation, design, and preliminary evaluation.

1 Introduction

Network experimentation environments that emulate some aspects of the environment—network testbeds—play an important role in the design and validation of distributed systems and networking protocols. In contrast to simulated environments, testbeds like Emulab [28] and PlanetLab [20] provide more realistic testing grounds for developing and experimenting with software. Emulated environments implement virtual network configurations atop real hardware: this means that experimenters can use real operating systems and other software, run their applications unmodified, and obtain actual (not simulated) performance measures.

A primary challenge for future emulation environments is scale. Because emulated environments are supported by actual hardware, an emulated system that is "larger" than the underlying physical system requires the careful allocation and multiplexing of a testbed's physical resources. To avoid experimental artifacts, the original Emulab used strictly conservative resource allocation. It mapped virtual network nodes and links one-to-one onto dedicated PCs and switched Ethernet links. We have four motivations for relaxing this constraint, allowing controlled multiplexing of virtual onto physical resources. First, some applications such as peer-to-peer systems or dynamic IP routing algorithms require large topologies or nodes of high degree for evaluation, yet are not resource-hungry. Second, much research and educational use simply does not need perfect performance fidelity, or does not need it on every run. Third, such multiplexing provides more efficient use of shared hardware resources; for example, virtual links rarely use their maximum bandwidth and so waste the underlying physical link. Fourth, it makes small-scale emulation clusters much more useful.

In this paper we present a collection of techniques that allow network emulation environments to virtualize resources such as hosts, routers, and networks in ways that preserve high performance, high fidelity, and near-total transparency to applications. Our primary motivation is scale: i.e., to support larger and more complex emulated environments, and to allow a single testbed to emulate more such environments at the same time. Our techniques allow a testbed to better utilize its physical resources, and allow a testbed to emulate kinds of resources that it may not have (e.g., hosts with very large numbers of network interfaces). One goal of our techniques is to preserve the performance fidelity of emulated resources, but our approach can also be used in cases where users do not require high fidelity, e.g., during early software development, in education, or in many

†Currently at Cisco Systems, VMware, and Morgan Stanley, respectively. Work done at the University of Utah.

kinds of reliability studies. Our techniques provide benefits to both testbed operators, who can provide better services with fewer hardware resources, and users, who have improved access to testbeds and their expanded services.

Our goal is that the overall system *scales well* with increasing size of virtual topologies. Scalability is not only about speed and size, but also concerns reliability and ease of use. We are concerned with (i) the time to reliably instantiate an experiment, which affects system throughput and Emulab's interactive usage model; (ii) the number of physical machines and links required for a particular virtual topology; (iii) monitoring, control, and visualization of testbed experiments, both by the user and the system; and (iv) the user's time spent in customizing instrumentation and management infrastructure. In achieving good scalability, however, we require two constraints to be met. One is that the emulation system be, as much as possible, *transparent to applications*. Even if they deal with the network or OS environment in an idiosyncratic manner, we should not require them to be modified, recompiled, relinked, or even run with magic environment variables. Second, we must provide good (not perfect) *performance fidelity*, so that experimenters can trust their results.

To meet our goals, our enhanced testbed multiplexes virtual entities onto the physical infrastructure using four key design techniques. First, it uses the minimum degree of virtualization that will provide sufficient transparency to applications. Second, it exploits the hierarchy found in the logical design of computer networks and in the physical realization of those networks. Our resource allocator relies on implicit hierarchy in the virtual topology to reduce its search space; our IP address assigner infers hierarchy to provide realistic addresses; and our testbed control system exploits the hierarchy between virtual nodes and their physical hosts. Third, our system employs optimistic automated resource allocation. The system or the user makes a "best guess" at the resources required, which are fed into a resource assigner that uses combinatorial optimization. Fourth, our testbed uses feedback to allocate resources adaptively. In training runs and in normal use, system-level and optional application-specific metrics are monitored. The metrics are used to detect overload or underload conditions, and to guide resource re-allocation. Emulab can automatically execute this adaptive process.

This paper makes the following contributions:

- It describes levels of virtualization that are appropriate for this domain, and discusses the design tradeoffs.
- It shows how to solve the NP-hard resource assignment problem for networks of thousands of entities, and describes how to support flexible specification of arbitrary resources.
- It presents a new feedback-directed technique to support virtualization and scaling.
- It outlines a new algorithm for efficiently assigning realistic IP addresses.
- It provides a preliminary experimental evaluation of various aspects of the system.
- The system it describes provides a useful new facility and is proven in production use.

One of the lessons of our work is that, in practice, achieving such a scalable system requires a collection of techniques covering a wide range of issues. The challenge is more broad and difficult than simply virtualizing an OS or virtualizing network links. It includes solving difficult problems in IP address assignment, in allocating node and network resources, in performance feedback, and in scalable internal control systems. We believe that this fact—that a host of problems must be solved to achieve scalable network experimentation in practice—is not widely recognized.

2 Testbed Context

The Emulab software is the management system for a network-rich PC cluster that provides a space- and time-shared public facility for studying networked and distributed systems. One of Emulab's goals is to transparently integrate a variety of different experimental environments. Historically, Emulab has supported three such environments: emulation, simulation, and live-Internet experimentation. This paper focuses on our work to expand it into a fourth environment, virtualized emulation.

An "experiment" is Emulab's central operational entity. An experimenter first submits a network topology specified in an extended *ns* [6] syntax. This virtual topology can include links and LANs, with associated characteristics such as bandwidth, latency, and packet loss. Limiting and shaping the traffic on a link, if requested, is done by interposing "delay nodes" between the endpoints of the link, or by performing traffic shaping on the nodes themselves. Specifications for hardware and software resources can also be included for nodes in the virtual topology.

Once the testbed software parses the specification and stores it in the database, it starts the process of "swapin" to physical resources. Resource allocation is the first step, in which Emulab attempts to map the virtual topology onto the PCs and switches with the three-way goal of meeting all resource requirements, minimizing use of physical resources, and running quickly. In our case the physical resources have a complex physical topology: multiple types of PCs, with each PC connected via four 100 Mbps or 1000 Mbps Ethernet interfaces to switches that are themselves connected with multi-gigabit links.

The testbed software then instantiates the experiment on the selected machines and switches. This can mean configuring nodes and their operating systems, setting up VLANs to emulate links, and creating virtual resources on top of physical ones. Emulab includes a synchronization service as well as a distributed event system through which both the testbed software and users can control and monitor experiments.

We have seven years of statistics on 2000 users, doing 69,000 swapins, allocating 806,000 nodes. An important observation is that people typically use Emulab *interactively*. They swap in an experiment, log in to one or more of their nodes, and spend hours running evaluations, debugging their system, and tweaking parameters, or sometime spend just a few minutes making a single run. When done for the morning, day, or run, they swap out their experiment, releasing the physical resources.

This leads to two points: speed of swapin matters, and people "reuse" experimental configurations many times. These points are important drivers of our goals and design.

3 Minimal Effective Virtualization

Multiplexing logical nodes and networks onto the physical infrastructure is our approach to scaling. *Virtualization* is the technique we use to make the multiplexing transparent. Our fundamental goal for virtual entities is that they behave as much like their real-life counterparts as possible. In the testbed context, there are three important dimensions to that realism: functional equivalence, performance equivalence, and "control equivalence." By the last, we mean similarity with respect to control by the testbed management system (enabling code reuse) and by the experimenter (enabling knowledge reuse and scripting code reuse). This paper concentrates on the first two dimensions, functional realism, which we call *transparency*, and performance realism.

Our design approach is to find the minimum level of virtualization that provides transparency to *applications* while maintaining high performance. If a virtualization mechanism is transparent to applications, it will also be transparent to experimenters' control scripts and to their preconceived concepts. We achieve both high performance and transparency by virtualizing using native mechanisms: mechanisms that are close to identical to the base mechanisms.

For virtual nodes, we implement virtualization within the operating system, extending FreeBSD's jail abstraction, so that unmodified applications see a system call interface that is identical to the base operating system. For virtual links and LANs, we virtualize the network interface and the routing tables. That allows us to provide key aspects of emulated networks using native switch-supported mechanisms such as broadcast and multicast. These mechanisms give us high—indeed native—performance, while providing near functional equivalence to applications. In our current virtual node implementation we give up resource isolation, but we are saved by our higher-level adaptive approach to resource allocation and detection of overload.

3.1 Virtual Nodes

There are many possible ways to implement some notion of a "virtual node." The available technologies and the trade-offs are well documented in the literature (e.g., [19]). When choosing the technology for Emulab virtual nodes, we evaluated each against four criteria, two from an application perspective, two from a system-wide perspective:

Application transparency. The extent to which virtual node name spaces (e.g., process, network, filesystem) are isolated from each other. (Can the application run unchanged?)

Application fidelity. The extent to which virtual node resources (e.g., CPU, memory, IO bandwidth) are isolated from each other. (Does the application get the resources it needs to function correctly?)

System capacity. The amount of virtualization overhead. (How many virtual nodes can we host per physical node?)

System flexibility. The level at which virtualization takes place (can we run multiple OSes?) and the degree of portability (can we run on a wide range of hardware?)

3.1.1 Emulab Virtual Nodes

Application transparency is important in the Emulab environment, requiring at least namespace isolation to be present. On the other hand, we anticipated that the initial network applications run inside virtual nodes would have modest CPU and memory requirements, making resource isolation—except for the network, which we already handle—less important. We do at least provide inter-experiment resource isolation since physical nodes are dedicated to experiments, hosting virtual nodes only for that experiment. Finally, we hoped to achieve at least a ten fold multiplexing factor on low-end PCs (850 MHz, 512 MB memory), necessitating a lightweight virtualization mechanism. Considering these requirements, a process-level virtualization seemed the best match. Given our BSD heritage and expertise, we opted to design and implement our virtual nodes by extending FreeBSD jails. In the following discussion, we refer to an instance of our virtual node implementation as a *vnode*.

Jails. Jails provide filesystem and network namespace isolation and some degree of superuser privilege restriction. A jailed process and all its descendents are restricted to a unique slice of the filesystem namespace using *chroot*. This not only gives each jail a custom, virtual root filesystem but also insulates them from the filesystem activities of others. Jails also provide the mechanism for virtualizing and restricting access to the network. When a jail is created, it is given a virtual hostname and a set of IP addresses that it can bind to (the base jail implementation allowed a single IP address with a jail, we added the ability to specify multiple IP addresses). These IP addresses are associated with network interfaces outside of the jail context and cannot be changed from within the jail. Hence, jails are implicitly limited to a set of interfaces they may use. We further extended jails to correctly limit the binding of the INADDR_ANY wildcard address to only those interfaces visible to the jail and added restricted support for raw sockets. Finally, jails allow processes within them to run with diminished root privilege. With root inside a jail, applications can add, modify and remove whatever files they want (except for device special files), bind to privileged ports, and kill any other processes in the same jail. However, jail root cannot perform operations that affect the global state of the host machine (e.g., reboot).

Virtual disks. Our design of virtual disks made it easy not only to be efficient in disk use, but also to support inter-vnode disk space separation. Jails provide little help: even though each jail has its own subset of the filesystem name space, that space is likely to be part of a larger filesystem. Jails themselves do nothing to limit how much disk space can be used within the hosting filesystem.

Our design uses the BSD *vd* device to create a regular file with a fixed size and expose it via a disk interface. Filesystem space is only required for blocks that are allocated in a virtual disk, thus this method is space-efficient for the typical case where the virtual disk remains mostly empty. These fixed-size virtual disks contain a root filesystem for each jail, mounted at the root of each jail's name space. Since the virtual disks are contained in regular files, they are easy and efficient to move or clone.

Control of vnodes. While enhancing the Emulab system with node types other than physical cluster nodes, we worked to preserve uniformity and transparency between the different node types wherever possible. The result is that the system is almost always able to treat a node the same, regardless of its type, except at the layers that come in direct contact with unavoidable differences between node types, or when we aggregate expensive actions by operating through the parent physical node.

An example of the transparency is the state machines used to monitor and control nodes of all types. While non-physical nodes have significant differences from physical nodes, the state machines used to manage them are almost identical. In addition, the same machine is used for Emulab vnodes as well as PlanetLab virtual servers. Reusing—indeed, sharing—such complex and crucial code contributes to the overall system's reliability.

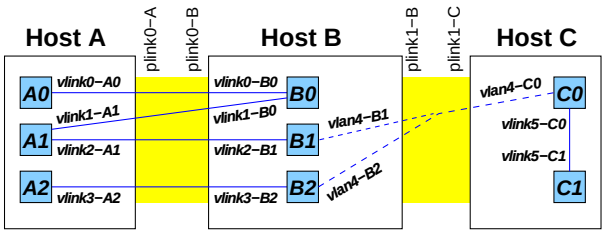

Figure 1: A network topology illustrating routing issues due to the multiplexing of virtual nodes and links. Large boxes represent physical nodes and links, while small boxes and lines (with *italic labels*) represent virtual nodes and links. Virtual network interfaces (*vlinks*), virtual LANs (*vlans*), and physical links (plinks) have names as shown.

3.2 Virtual Networks

Virtualizing a network includes not only multiplexing many logical links onto a smaller number of physical links, but also dealing with network namespace isolation issues and the subtleties of interconnecting virtual nodes within, and between, physical nodes. As with virtualizing nodes, there is a range of techniques available for virtualizing networks. When choosing the technology for Emulab, the criteria we evaluated against were:

Level of virtualization. A virtual network implementation can present either a virtual layer 2 (e.g., Ethernet) or a virtual layer 3 (e.g., IP).

Use of encapsulation. Virtual network links may (e.g., 802.1Q VLANs) or may not (e.g., "fake" MAC addresses) encapsulate their packets when traversing physical links.

Sharing of interfaces. The end point of a virtual network link as seen by a virtual node may be either a shared physical interface device or a private virtual one. This may affect whether interface-centric applications like tcpdump can be used in a virtual node.

Ability to co-locate virtual nodes. Can there be more than one virtual node from a given network topology on the same physical host? If so, there are additional requirements for correct virtualization.

3.2.1 Emulab Virtual Networks

The Emulab virtual node implementation uses unshared, virtual Ethernet devices in order to maintain transparency with the "bare machine" model which presents dedicated physical Ethernet devices to applications. By default, these virtual devices are configured to use a custom 16-byte encapsulation format, allowing use of the virtual devices with any switching infrastructure. There is also an option to allow rewriting the source MAC address in outgoing packets with the virtual MAC address. Since any physical host is dedicated to a single experiment, whether using virtual nodes or not, it is necessarily the case that virtual nodes for a topology will be co-located. This raises two issues related to forwarding packets that are addressed in the next sections.

Virtual network interfaces. While the FreeBSD jail mechanism does provide some degree of network virtualization by limiting network access to specific IP addresses, it falls short of what we need. In particular, though jails have their own distinct IP addresses, those IP addresses are associated directly with shared physical interfaces, and thus have problems with interface-oriented applications such as `tcpdump`. Moreover, because we allow co-location, it is possible that two virtual nodes in the same LAN could wind up on the same physical host, for example B1 and B2 in Figure 1. FreeBSD does not allow two addresses in the same subnet to be assigned to one interface.

To solve these problems, we developed a virtual Ethernet interface device ("veth"). The veth driver is an unusual hybrid of a virtual interface device, an encapsulating device and a bridging device. It allows us to create unbounded numbers of Ethernet interfaces (virtualization), multiplex them on physical interfaces or tie them together in a loopback fashion (bridging) and have them communicate transparently through our switch fabric (encapsulation). These devices also provide the handle to which we attach the IPFW/Dummynet rules necessary for doing traffic shaping. Veth devices can be bridged with each other and with physical interfaces to create intra- and inter-node topologies and ensure the correct routing of packets at the link level. For example, this bridging prevents "short-circuit" delivery of traffic between co-located nodes A0 and A2 in Figure 1, which might otherwise occur when FreeBSD recognized that both interfaces are local. Since multiple veth devices may be bridged to the same physical device, incoming packets on that device are demultiplexed based on the virtual device's MAC address, which is either contained in the packet (encapsulation) or is exposed directly as the packet's MAC address (no encapsulation).

Virtual routing tables. While virtual Ethernet devices are sufficient to enable construction of virtual Ethernet topologies, they are not sufficient to support arbitrary IP topologies. This is due to FreeBSD jails sharing the host's IP infrastructure, in particular, the routing table. In the routing table, it is only possible to have one entry per destination. But with a physical node hosting multiple jails representing different virtual nodes at different points in the topology, we need to be able to support multiple routes to (next hops for) a single destination. This is known as the "revisitation" problem [24]. For example, in Figure 1, packets sent from A0 to C0 will pass through host B twice. B0's next hop for C needs to be A (for A1) while B1's needs to be C (for C0). Thus there need to be separate routing tables for B0 and B1. Further, even with separate routing tables, incoming packets to B need context to determine which routing table to use.

For Emulab, we have adopted and extended the work of Scandariato and Risso [22] which implements multiple IP routing tables to support multiple VPN end points on a physical node. Routing tables are identified by a small integer routing table ID. An ID is the glue that binds together a jail, its virtual interfaces, and a routing table. Incoming packets for different jails on a physical node can thus be differentiated by the ID of the receiving interface and can be routed differently based on the content of the associated routing table.

3.2.2 IP Address Assignment

A subtle aspect of implementing virtual networks is assigning addresses, in particular IPv4 addresses, to the potentially thousands of links which make up a topology. The topologies submitted to Emulab typically do not come annotated with IP addresses; most topology generators do not provide them, and it is cumbersome and error-prone for experimenters to assign them manually. We thus require an automated method for producing "good" IP address assignments, and it must scale to the large networks enabled by virtualization. A desirable address assignment is one that is realistic—that is similar to how addresses would be allocated in a real network. In the real world, the primary (though not only) factor that influences address assignment is the underlying hierarchy of the network. Hierarchical address assignment also leads to smaller routing tables and thus better scaling. Since real topologies are not strictly hierarchical, the challenge becomes identifying a suitable hierarchical embedding of the topology. Our work on IP address assignment centers on inferring hierarchy in this practical setting.

We developed and evaluated three different classes of algorithms: bottom-up, top-down, and spectral methods, described in detail elsewhere [8]. The approach we ultimately deployed in production, called recursive parti-

tioning, creates the IP address tree in a top-down manner. At the top level, the root of the tree contains every node in the graph. Then we partition it into two pieces using a graph-partitioner [17], assigning each half of the graph to a child of the root. By applying this strategy recursively, we create a tree suitable for IP address assignment. The result is a fast algorithm that produces small routing tables: for example, it can assign addresses to networks of 5000 routers—comparable to today's largest single-owner networks—in less than 3 seconds.

4 Automated Resource Assignment

When an experimenter submits an experiment, Emulab automatically chooses a set of physical nodes on which to instantiate that experiment. This process of mapping the virtual topology to a physical topology is called the network testbed mapping problem [21], and virtual nodes add new challenges to an already NP-hard problem. The needs of this mapping are fundamentally similar to other virtualized networking environments, such as the planned GENI facility [10] (where such mapping will be done by a Slice Embedding Service), and ModelNet [26].

For an experiment with virtual nodes, a good mapping is one that "packs" virtual hosts, routers, and links on to a minimum number of physical nodes without overloading the physical nodes. This means, for example, placing, when possible, nodes that are adjacent in the virtual topology on the same physical node, so that the links between them need not use physical interfaces or switch capacity. This is particularly difficult because the virtual nodes may not have uniform resource needs, and physical nodes may not have identical capacities. Since Emulab is a space-shared testbed, it is also important that bottleneck resources, such as trunk links between switches, are conserved, since they may be needed by other concurrent experiments. Finally, experimenters may request nodes with special hardware or software, and the mapper must satisfy these requests.

Emulab finds an approximate solution to the network testbed mapping problem by taking a combinatorial optimization approach. It uses a complex solver called `assign` [21]. `assign` is built around a simulated annealing core: it uses a randomized heuristic to explore the solution space, scoring potential mappings based on how well they match the experimenter's request, avoid overloading nodes and links, and conserve bottleneck resources. We found, however, that Emulab's existing `assign` was not sufficient for mapping virtual node experiments, and enhanced it accordingly.

First, we needed new flexibility in specifying how virtual nodes are to be multiplexed ("packed") onto physical nodes. To get efficient use of resources, we found it necessary to add fine-grained resource descriptions, and to relax `assign`'s conservative resource allocation policies.

Second, because virtualization allows for topologies that are an order of magnitude larger than one-to-one emulation, we ran into scaling problems with `assign`. Since it must be run every time an experiment is swapped in or re-mapped as part of auto-adaptation, runtimes in the tens of minutes were interfering with the usability of the system and making auto-adaptation too cumbersome. To combat this, we made enhancements to `assign` that exploit the natural structure of the virtual topologies it is given to map.

4.1 Flexible Resource Specification

`assign` must use some criteria to determine how densely it can pack virtual nodes onto physical nodes. `assign` already had the ability to use a coarse-grained packing, in which each physical node has a specified number of "slots," and each virtual node is assumed to occupy a single slot. Thus, it can be specified that `assign` may pack up to, for example, 20 virtual nodes on each physical node. It became clear that this would not be sufficiently fine-grained for many applications, including our auto-adaptation scheme, because different virtual nodes will have different roles in the experiment, and thus consume different amounts of resources.

To address this, we added more packing schemes to `assign`. In the first, virtual nodes can fill more than one slot; experimenters can use this when they have an intuitive knowledge, for example, that servers in their topology will require more resources than clients by an integer ratio: 2:1, 10:1, etc.

The second packing scheme models multiple independent resources such as CPU cycles and memory, and can be used when the experimenter has estimated or measured values for the resource needs of the virtual nodes. Each virtual node is tagged with the amount of each resource that it is estimated to consume, and `assign` ensures that the sum of resource needs for all virtual nodes assigned to a particular physical node does not exceed the capacity of the physical node. This scheme builds on `assign`'s system of "features and desires": virtual nodes can be identified as having "desires" which must be matched by "features" on the physical nodes they are mapped to. Features and desires are simply opaque strings, making this system flexible and extensible. We have enhanced `assign` to allow features and desires to also express capacities, which are then enforced as described above. While we use this scheme for relatively low-level resources (CPU and memory), it could also be used for higher-level metrics such as sustainable event rate for discrete event simulators such as *ns*.

The resource-modeling scheme is particularly useful

for feedback-based auto-adaptation. The values used for CPU and memory consumption of a virtual node can simply be obtained by taking measurements of an earlier run of the application. The maximum or steady-state usage can then be used as input to the mapping process. The coarse-grained and resource-based packing criteria can be used in any combination.

In addition to packing nodes, virtual links must be packed onto physical links. Though the two types of packing are conceptually similar, a different set of issues applies to link packing. Some of these issues exist for one-to-one emulation, but there are also some new challenges that come with virtual emulation.

Link mapping issues that one-to-one and virtual emulation have in common. First, physical nodes in a Emulab-based testbed have multiple interfaces onto which the virtual links must be packed. Second, the topology of the experimental network is typically large enough that it is comprised of multiple switches. These switches are connected with links that become a bottleneck, so the mapping must be careful to avoid over-using them.

Link mapping challenges that arise with virtual emulation. When mapping virtual-node experiments, links between two virtual nodes that are mapped to the same physical node become "intra-node" links that are carried over the node's "loopback" interface. It is advantageous to use intra-node links, as they do not consume the limited physical interfaces of the physical node. Although the bandwidth on a loopback interface is high, there are practical limits on it, and for some experiments that use little CPU time but large amounts of bandwidth, loopback bandwidth can become the limiting factor for packing virtual nodes onto physical ones. We have extended `assign` to take this finite resource into account.

One of the guiding principles of `assign` has historically been conservative resource allocation; when assigning links, it ensures that the full bandwidth specified for the link will always be available. While this makes sense for artifact-free emulation, is at odds with our goal of using virtualization to provide best-effort, large-scale emulation. For example, an experimenter may have a topology containing a cluster of nodes connected in a LAN. Though the native speed of this LAN is 100 Mbps, the nodes in this LAN may never transmit data at the full line rate. Thus, if `assign` were to allocate the full 100 Mbps for the LAN, much of that bandwidth would be wasted. To make more efficient resource utilization possible, we have added a mechanism so that estimated or measured bandwidths can be passed to `assign`. As with node resources, this bandwidth can be measured as part of auto-adaptation.

4.2 Improving `assign`'s Scaling

`assign` has been designed and tuned to run well on Emulab's typical one-to-one workload, consisting of topologies with at most a few hundred nodes. In order to make `assign` scale to topologies of the scale enabled by virtual nodes, we developed several new techniques.

4.2.1 Searching the Solution Space

Our first techniques for tackling scaling issues are aimed at improving the way in which `assign` searches through the solution space of possible mappings. `assign` finds sets of homogeneous physical nodes and combines them into equivalence classes; this allows it to avoid large portions of the solution space which are equivalent, and thus do not need to be searched. However, this strategy breaks down with the high degree of multiplexing that comes with virtual-node experiments, because a physical node that has been partially filled is no longer equivalent to an empty node. We have addressed this problem by making these equivalence classes adapt dynamically at run time, with physical nodes entering and leaving classes as virtual nodes are assigned or unassigned to them.

Another improvement to the search strategy came from the observation that, in a good solution, nodes that are adjacent in the virtual topology will tend to be placed on the same physical node. So, we made an enhancement to the way `assign` selects new virtual-to-physical mappings to try, as it moves through the search space. To conduct this search, `assign` takes a potential solution, selects a virtual node, selects a new physical node to map it to, and determines whether or not the resulting mapping is better than the original. This process is repeated, typically hundreds of thousands or millions of times, until a no better solutions are found. In our modified version, rather than selecting a random physical node, with some probability, `assign` selects a physical node that one of the virtual node's neighbors has already been mapped to. This improvement made a dramatic difference in solution quality, leading to much tighter packing and exhibiting much better behavior in clustering connected nodes together.

4.2.2 Coarsening the Virtual Graph

Though these changes to the search strategy improved `assign`'s runtime and solution quality, running `assign` on very large topologies could still take more than an hour, much too long for our purposes. To make the problem more tractable, we exploit topological features of the virtual topology.

We expect that most large virtual topologies will be based on the structure of the Internet; these may come

from actual Internet "maps" from tools like Rocketfuel [23] or from topology generators designed to create Internet-like networks, such as GT-ITM [33], inet [29], and Orbis [15]. The key realization is that such networks tend to have subgraphs of well-connected nodes, such as ISPs, ASes, and enterprises. In addition, we expect that many topologies will have edge-LANs that represent clusters, groups of workstations, etc.

We exploit the structure of the input topology by applying a heuristic coarsening pre-pass to the virtual graph before running `assign`. By giving `assign` a smaller virtual topology, we reduce the solution space that it must search, in turn reducing the time required to find a good solution. The goal of this pre-pass is to find sets of virtual nodes that, in a good mapping, will likely be placed on a single physical node. A new virtual graph is then generated, with each of these sets combined into a single node. These "conglomerates" retain all properties of their constituent nodes; for example, the CPU needs of each constituent are summed together to produce the CPU required for the conglomerate.

We have implemented two coarsening algorithms. The first stems from the realization that many topologies contain LANs representing groups of clients or farms of servers. An optimal mapping will almost always place as many members of these LANs onto a single physical node as possible. So, we find leaf nodes in LANs (that is, nodes whose only network interface is in that LAN), and combine all leaf nodes from the same LAN into a conglomerate.

The second algorithm uses a graph partitioner, METIS [17], to partition the nodes in the virtual graph. We choose a number of partitions such that the average partition will fit on the "smallest" available physical node. We then combine the virtual nodes in each partition into a single conglomerate node. The quality of the partitions returned by the partitioner is dependent on the extent to which separable clusters of nodes are present in the graph. Since we are focusing on Internet-like topologies with some inherent hierarchy, we expect good results from this method.

The coarsening algorithms (particularly METIS) do not know the intricacies of the network testbed mapping problem, such as constraints on node types, resource usage, and link bandwidths; this is one reason they are able to run much faster than `assign` itself. This leaves us with the problem that they may return sets of nodes to cluster that cannot be mapped onto any physical resources; for example, they may require too much CPU power or have more bandwidth than a single node can handle. Once the coarsening algorithm has returned sets of nodes, we use a multidimensional bin-packing approximation algorithm to pack these into the minimum number of mappable conglomerates.

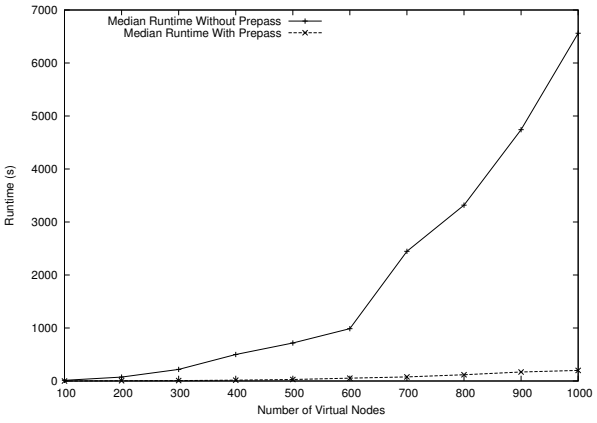

Figure 2: Median runtime of `assign` with and without a coarsening pre-pass.

Both coarsening algorithms help `assign` to run faster by making heuristic decisions that limit `assign`'s search space, but could, in turn, make clustering decisions that result in sub-optimal mapping. However, in our domain obtaining a solution in reasonable time is more important than obtaining a near-optimal solution. The mappings obtained by `assign` will always be valid, but it is possible that some topologies are coarsened in such a way the mapping does not make the most efficient use of resources. The biggest potential problem is fragmentation, in which the coarsening pass makes conglomerates whose sizes do not pack well into the physical nodes. We take measures to try to avoid this circumstance, by carefully choosing our target conglomerate size. In practice, the worst fragmentation we have seen caused only a 13% increase in physical resources used.

To evaluate our new resource mapper as well as to understand the effects of the coarsening pre-pass, we compared runs of `assign` with and without the pre-pass. These runs mapped transit-stub topologies generated by GT-ITM [33] onto Emulab's physical topology. Each test was run ten times. In all cases, the runtime of the pre-pass itself was negligible compared to the runtime of `assign`.

Figure 2 presents the median runtimes for these tests on a 1.5 GHz Pentium IV, showing the greatly significant time savings from the pre-pass. As we scale up the number of virtual nodes the improvement goes from a factor of 15 at 100 nodes (12.0 to 0.78 seconds), to a factor of 32 at 1000 nodes (6560 to 200 seconds). The absolute result is also good: it takes just 200 seconds to map 1000 nodes.

This speedup, of course, does not come without a cost. Figure 3 shows the decrease in solution quality, in terms of the quality of link mappings. Intra-node links connect two virtual nodes mapped to the same physical node;

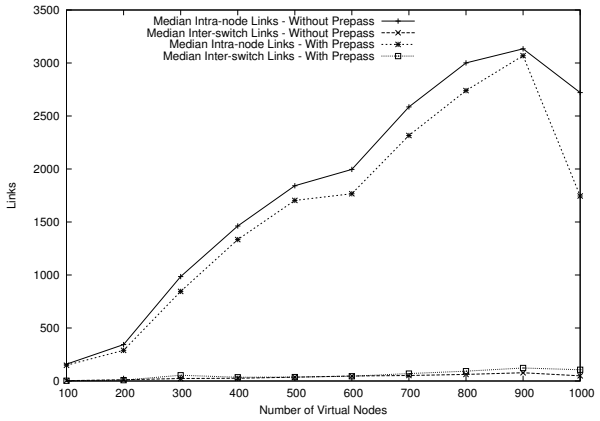

Figure 3: Number of intra-node and inter-switch links found by `assign`. Larger numbers of intra-node links are better, and smaller numbers of inter-switch links are better.

they do not use up shared switch resources, so having a large number of them is an indicator of a good mapping. Inter-switch links, on the other hand, are an indicator of a poor mapping, because they consume the shared resource of inter-switch links. Though the pre-pass does cause `assign` to find somewhat worse mappings, the differences are tolerable, and the speedup is a clear win. In over 70% of the test cases, the number of intra-node links found when using the pre-pass was within 10% of the number found by `assign` by itself. The worst run was within 16%.

5 Exploiting Physical Hierarchy

In addition to the previously described IP assignment and mapping problems, a number of more general but severe "system" scaling issues arose, which prevented us from reaching large size until we addressed them. Some are system-wide issues that are the byproducts of the order of magnitude increase in the potential size of an experiment. Others are per-node issues that are the result of increasing the resource consumption on a node. In both cases, we devise solutions that exploit the physical structure and realities of the physical testbed infrastructure.

Most system-wide problems have to do with accessing centralized services and the use of unreliable protocols, primarily during initial experiment setup. The system-wide scaling problems encountered here are essentially the same issues faced when increasing the number of physical machines in the testbed. For example, sharing a single NFS filesystem does not scale well. We are constantly addressing these types of issues as we expand into larger virtual node experiments. Ultimately, virtual node growth will continue to outpace physical resource growth by 1–2 orders of magnitude. However, by leveraging the close relationship between virtual nodes and their host we significantly reduce the burden on the central infrastructure as highlighted by the following examples.

There are a number of situations in which we use the physical host as a caching proxy for its hosted virtual nodes. Nodes in Emulab "self configure" when they boot, after obtaining the necessary configuration information from a central server. Since the physical host necessarily boots before its virtual nodes, it downloads configuration information for all virtual nodes in a single operation, pre-loading a cache for each, and in some cases, performing configuration operations itself in a more efficient manner. Similarly, the physical host acts as an Emulab event system proxy, using a single connection to the master event server to collect and distribute control events for all its virtual nodes.

One of the most compute-intensive parts of instantiating an experiment is calculating routing tables for all of the nodes. Though Emulab supports dynamic routing through the use of a routing daemon such as `gated` or `zebra`, most experimenters prefer the consistency and stability offered by computing routing tables off-line before the experiment begins. Typical algorithms for doing this, however, have runtimes ranging from $O(V^2 \cdot lg(V) + V \cdot E)$ (Dijkstra's algorithm with a Fibonacci heap) to $O(V^3)$ (Dijkstra's algorithm with a linear-array priority queue), with respect to the number of vertices (nodes) and edges (links) in the topology graph. To solve this problem, we parallelize route computation across all of the physical nodes in the experiment, with each physical node responsible for the routing tables of the virtual nodes it hosts. We distribute one copy of the topology to each physical host, and run Dijkstra's algorithm sourced from each virtual node hosted on that physical node. Thus the route calculation time becomes $O(V^2 \cdot n)$, where n is the number of virtual nodes hosted on each physical node. In practice, with the size of virtual topologies that are feasible to run on Emulab and the level of virtual-to-physical multiplexing possible, this time never exceeds a few seconds.

The original Emulab system could not reliably instantiate an experiment larger than about 100 nodes. Our improvements in Emulab allow experiments of up to at least two thousand nodes to be reliably instantiated. A fundamental limitation on speed of instantiation is that vnode construction is not parallelizable within a uniprocessor host. However, virtual nodes on distinct physical hosts can be setup in parallel. To demonstrate the degree to which this parallelism can be successfully exploited, we performed a simple test in which an experiment consisting of a single LAN was repeatedly instantiated, each time adding to the LAN one physical node hosting 10 virtual nodes. In the base case of one physical node with 10 virtual nodes in the LAN, setup, including topology mapping, node configuration and startup, required 194

seconds. At 80 virtual nodes on 8 physical nodes, it took 290 seconds, a 50% increase in time for an 800% increase in size.

6 Feedback-Directed Resource Allocation

Maximum scalability is achieved when Emulab's physical nodes and networks can be divided as finely as possible, each physical resource providing support to as many emulated and/or simulated entities as possible. However, for these emulated and simulated environments to be worthwhile to most Emulab users, they must be accurate recreations of devices in the real world. Meeting our scalability goal and our realism constraint at the same time means making virtual nodes that are "just real enough" from the point of view of the software systems under test.

Finding the proper balance between scalability and fidelity is not easy: the ideal tradeoff that is "just real enough" is inherently specific to the software being tested. Therefore, to find the appropriate resource mappings for a user's experiment, our technique is to automatically search for a mapping that minimizes physical resource use while preserving fidelity according to application-independent (provided by the system) and/or application-dependent (provided by the user) feedback.

Testbed users have two options for adapting their experiments: executing a single-stage "training" run that requires little effort, or running a multi-stage *automatic experiment adapter* that requires additional effort.

The first option does not require the experiment to be fully automated, so is suitable for an interactive style of experimentation. In this model, the user creates an experiment and swaps it in on virtual nodes mapped one-to-one on physical nodes to ensure adequate resources. Users can then login to the nodes, run their programs, and, when they have determined that the experiment is in a representative state, click a button to record a profile. This profile is then used in subsequent runs to drive the resource mapping. Of course, because of the one-to-one initial mapping, this simplistic manual approach will not work for large topologies. For those topologies we will necessarily need to start out with some virtual nodes multiplexed many to one on physical nodes and thus we cannot gather an accurate resource requirements profile with a single run. For these situations we offer the second option.

In the multi-stage approach, an experiment is automatically run multiple times, each time adjusting the mapping to account for any resource overloads noted in the previous run. To do this, the user must automate the execution of the experiment. Each run of the experiment starts up a representative workload, monitors resource usage once that workload reaches a steady state, invokes a script to gather the monitor output and create a profile, and then remaps and reinstantiates the experiment based on that profile. This process continues until there is a run in which no resource overload is detected.

Our feedback-driven adaptation technique automatically finds virtual-to-physical mappings that provide the user's required level of emulation fidelity while allowing Emulab to make maximally efficient use of its resources. There is a risk, however, that the mappings set up by the adapter will fail to provide sufficient fidelity to the user's software during "production" testbed runs, e.g., because the user modifies the software or is driving it in a different way. Emulab relies on run-time feedback to detect such cases and signal the user about possible problems with his or her experiment.

6.1 Implementation

Ensuring application fidelity when multiplexing virtual nodes can be achieved quickly and accurately through monitoring the application's steady state resource usage and feeding this data back into `assign`. Utilizing application-independent metrics, like CPU and memory usage, we can automatically adapt the packing of virtual resources on to physical hosts. This is done in a way that minimizes physical resource use while leaving sufficient headroom for the vnode's steady state resource consumption. Any available application-specific metrics can then be used to refine the mapping to account for lack of precision in the low level data.

On each physical node Emulab gathers a number of application-independent resource usage statistics to feed back to the adaptation mechanism. These include CPU use, interrupt load, disk activity, network traffic rates, and memory consumption. CPU and memory information are also gathered at vnode granularity, which is how we determine the resource demand of individual vnodes. The other global statistics allow us to ensure that the physical node as a whole is not overloaded.

The multi-stage adaptation technique requires that an experiment be automated, running a particular sequence of actions. This is easily done using Emulab's event system, which allows for executing operations in parallel, in sequence, or at specific times relative to the start of the experiment. The user only needs to create an event sequence to perform the steps described earlier. There are built-in events for two feedback specific activities. One allows for running the application-independent resource monitor on all nodes for a fixed length of time. The other performs the remap itself, a process which includes gathering the log files from all nodes, analyzing the data to detect overloads and produce new per-node resource usage estimates, and finally invoking `assign` and reconfiguring the experiment to reflect the new virtual node

layout.

The mechanism for supporting application-dependent metrics and providing user-directed feedback based on those metrics is a prototype and likely to change. In the current implementation, the user must provide three components. First, he or she provides one or more resource monitors that gather and log appropriate resource usage data. These could be separate programs, or they could just be the applications themselves, logging relevant information. Second, the user provides a "baseline summary" file describing the expected behavior of the system when not constrained by node resources. This summary file can be in any format, as it is interpreted by a user-supplied script. That script is the third component. The script is automatically invoked at the end of each run of the experiment, with the log files from the monitors and the baseline summary as inputs. Its job is to aggregate the log file information into a new summary and compare that summary with the baseline to determine if there is a resource overload. If the answer is "yes," then the experiment will be remapped.

6.2 Usage Scenarios

We see three common ways in which the auto-adaptation mechanism can be used. In the first, the user starts with a one-to-one mapping of virtual nodes to physical nodes and "packs" the experiment into fewer physical nodes. Starting with the one-to-one mapping we gather bootstrap resource data in the first pass. The system then runs successive passes, increasing or decreasing the packing until it arrives at a maximally dense packing factor with virtual node resource use that is consistent with the one-to-one mapping. At this point, the user will probably want to increase the size of her or her topology. The simplest approach is to use the bootstrap data for nodes that will remain in the experiment and perform a bootstrap on the newly added nodes. Alternatively, the user can divide nodes into resource classes (e.g., client/server), which are initialized using data derived from previous runs.

A second style of adaptation, using the same mechanism, is to start with a dense mapping of a topology and then expand it. A dense mapping is achieved by providing no initial feedback data, allowing `assign` to map strictly on the basis of available physical node and link characteristics. In this configuration, there can be no training run to gather clean resource usage data. Instead, feedback data are provided by the application-independent metrics (pushing the experiment away from obvious overload conditions) or with interactive guidance from the user. This form of adaptation is used with large topologies where there are not enough physical resources to map it one-to-one.

Finally, in the third scenario, an Emulab experiment can incorporate purely simulated nodes and networks, using a modified version of *nse* [9]. As described in detail elsewhere [12, 13], these simulated entities can be transparently spread across physical nodes, just as vnodes are dispersed. Since these simulated nodes interact with real traffic, the simulator must keep up with real time. Detecting when virtual time has significantly fallen behind real time gives us a way to detect overload that is more straightforward than with vnodes, although there are subtleties with synchronizing the two that must be taken into consideration, as described in the above references. Our infrastructure can adaptively remap simulated networks similarly to the way it handles virtualized nodes and links.

7 Results

In the following sections we present a preliminary evaluation of the Emulab virtual node implementation in three areas: application fidelity, application transparency, and performance and fidelity of the adaptation mechanism. All results were gathered on Emulab's low-end "pc850" machines, 850 MHz PCs with 512 MB of RAM and four 100 Mb Ethernet interfaces.

7.1 Application Fidelity

Microbenchmarks. To get a lower-level view of fidelity with increasing co-location, we performed an experiment in which we ran the `pathrate` [7] bandwidth-measurement tool between pairs of nodes co-located on the same physical host. Each pair of nodes was connected with a T1-speed (1.5 Mbps) link. We measured the bandwidth found by `pathrate` as we increased the number of node pairs from one to ten. Across all runs, `pathrate` measured the correct bandwidth to within 1 Kbps, with a standard deviation across runs of `pathrate` of 0.004.

Applications. We ran a synthetic peer-to-peer file sharing application called *Kindex*, which is modeled after a peer music file sharing network such as KaZaa. Kindex maintains a distributed peer-to-peer index of file contents among a collection of peer servers. It also keeps track of replicas of a file among peers and their proximity, to expedite subsequent downloads of the same file. In our simplified experiment, we start a series of 60 clients sequentially. Each of 60 clients uploads a single file's information to the global index, and starts randomly searching for other files, fetching those not previously fetched into its local disk. Each client generates between 20 to 40 requests per minute for files, whose popularity follows a Zipf distribution. Each client has sufficient space to hold all 60 files. Hence after the experiment has run for

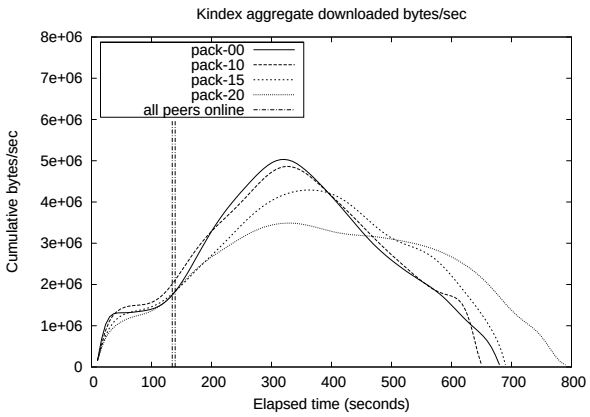

Figure 4: Cumulative system bandwidth for co-location factors of 0, 10, 15 and 20. "All peers online" is the point in time where all 60 peers are running and downloading files.

a while, all clients end up caching all files, at which time we stop the experiment.

The network topology consists of six 10 Mbps campus LANs connected to a core 40 Mbps LAN of routers with 100 ms roundtrip between themselves. Each campus LAN is connected to a router via a 3 Mbps, 20 ms RTT link.

We plotted the aggregate bandwidth delivered by the system to all its users as a time line. For this, we measured the total size of files downloaded by all users in every 10-second interval. We expect that initially downloads are slow, but as popular files are cached widely, subsequent downloads are more likely to be satisfied from a peer within the same campus, driving up the aggregate bandwidth due to the higher speed links. However, due to the fetch-once behavior of clients, as more files are downloaded by all users, downloads become less frequent, driving down the aggregate bandwidth.

We ran the experiment in four configurations. First, we emulated the topology on just physical nodes to establish a base line. We then repeated the experiment using virtual nodes with co-location factors of 10, 15 and 20 virtual nodes per physical node. Figure 4 shows the results. The base line (pack-00) shows the expected behavior, aggregate bandwidth increasing to a peak and then tapering off. At a co-location factor of 10, one campus LAN mapped per physical node, the behavior is indistinguishable for the base line. However, as we increase the co-location to 15 and 20, since peers have to supply files over the faster LAN links, the load on the local disk rises. This is the reason for the reduced peak bandwidth and its shift to the right, causing the curve to be flattened.

While this example shows that we can achieve an order of magnitude scaling improvement with an IO intensive application on low-end PCs, it also illustrates the utility of feedback data for driving virtual node multiplexing. In this example, some node disks hit 100% busy in both the pack-15 and pack-20 cases, an event easily detected by application-independent metrics. However, the user might also decide that the results from pack-15 were acceptable, but those in pack-20 were not. In this case they might construct a custom metric saying that remapping is only necessary if the disk were saturated for three consecutive measurements.

7.2 Application Transparency

Correctly achieved transparency is difficult to rigorously demonstrate; only failures of transparency are obvious. Our most compelling evidence is that experimenters have run thousands of diverse virtual node experiments, yet generated only a handful of requests for "missing features" such as support for multicast routing and IPFW firewall rules. We did perform one empirical stress test, running a routing daemon in a complex virtual network topology. By causing a series of link failures within the topology, we verified that the routing daemons were functioning as expected. In that test, we ran unmodified `gated` routing daemons on all nodes in a 416 vnode hierarchical topology on 22 PCs and automatically generated OSPF configuration scripts. Once we verified the connectivity between some leaf nodes across the diameter of the topology, we caused a link failure in the interior to see how OSPF would route around the failure. Before the failure, a route between two leaf nodes was symmetric with 11 hops. We found a 5 second downtime in one direction and 9 seconds in the reverse direction, after which alternate 12 hop paths were established. The forward and reverse paths were different in one hop. When we removed the link failure, it took 22 and 28 seconds respectively for the route paths to be restored. Finally, we rebooted two interior nodes in the topology. `gated` restored all the routes in a little over a minute.

7.3 Adaptation Results

We evaluated our feedback system in three scenarios: a Java-based web server and clients, the BitTorrent peer-to-peer file distribution system, and the Darwin Streaming Server [2].

We first ran a Java-based web server on one host with 69 clients continually downloading a 64 KB file. The clients were separated into three different types based on their link characteristics. Nine clients were evenly spread across three links on a single router using 2 Mb LANs to emulate cable modem clients. Forty clients were directly connected to a single router using 2 Mb multiplexed links to emulate DSL modems. Finally, 20 clients

Metric	2 Mb LAN	2 Mb Link	56 Kb Link
74 vnodes on 74 physical nodes			
Avg. Transaction Rate	1.19	2.29	0.09
Avg. Response Time (s)	0.84	0.43	10.67
Packed onto 7 phys. nodes after first iteration			
Avg. Transaction Rate	1.10	1.85	0.09
Avg. Response Time (s)	0.91	0.53	10.77
Packed onto 7 phys. nodes after second iterations			
Avg. Transaction Rate	1.19	2.29	0.09
Avg. Response Time (s)	0.84	0.43	10.70

Table 1: Performance of clients continually downloading a 64 KB file in different vnode mappings.

were directly connected to a single router using 56 Kb multiplexed links, to emulate phone modem clients. The feedback loop required three iterations to reach acceptable application fidelity; the results are shown in Table 1. The first iteration is a one-to-one mapping that allows the system to get a clean set of feedback data. The second iteration packed the 74 vnodes onto 7 physical nodes and resulted in a drop in performance because the CPU intensive server node was co-located with several client nodes. The final iteration amplifies the feedback data (i.e., increases the CPU and memory requirements) by 20%, which is enough to isolate the server and return the application metrics to their original one-to-one values, without allocating any more physical nodes. It should be noted that the bad mapping found in the second iteration could have been avoided with higher precision monitoring. However, in our context a bad initial remapping is a benefit because it denotes the lower bound on the number of required nodes and we always wish to minimize the number of physical nodes required for a topology.

To demonstrate scaling a real application to large topologies that cannot fit in a one-to-one mapping, we ran the BitTorrent p2p file distribution program on a 310-node network packed onto 74 physical nodes. The topology consisted of 300 clients communicating over 2 Mb LANs or links, a single "seed" node with a 100 Mb link, and nine routers that formed the core. To bootstrap the mapping we used feedback data from a smaller topology for the clients, since their resource usage was dependent on the link constraints and not the number of clients in the system. However, the resource use of the seed node and routers is tied to the size of the network, so they were left one-to-one. In total, it took 19 minutes to instantiate the topology: seven minutes for `assign` to map the virtual topology onto the physical topology and twelve minutes to load disks onto the machines, reboot, and setup the individual virtual nodes. This should be comparable to the length of time it would take to setup the same

Mapping	Video gap (ms)		Audio gap (ms)	
	Min	Max	Min	Max
One to one	0.93	90.99	48.23	210.96
Phys. Link Shared	0.04	470.3	0.07	531.27
Phys. Link Unshared	0.54	91.99	30.88	232.10

Table 2: Interpacket gap of clients receiving a 100 Kbps video and audio stream in different configurations where the physical link is shared and not shared. The values are the median of five runs.

topology on physical machines (if Emulab had sufficient nodes). On physical nodes, the `assign` time would be less, but the time to setup switch VLANs would exceed the time required to setup virtual links.

The adaptation mechanism can also accommodate applications that have throughput constraints as well as timing sensitivity. We tested the Darwin Streaming Server sending a 100 Kbps video and audio feed to 20 clients. When packed densely to 2 physical nodes, the interpacket gap variance is high, but if we set the estimated bandwidth for the client links to 100 Mb, sparser virtual to physical link mapping results. This in turn forces virtual nodes to relocate onto other physical nodes, raising the total number physical nodes to 6 (see Table 2). The oversubscription of network bandwidth thus clears a path for time sensitive packets.

8 Related Work

The ModelNet network emulator [26] achieves extremely large scale by foregoing flexibility and optionally abstracting away detail in the interior of a network topology. Edge hosts run the user's applications on generic OSes, using IP aliasing and a socket interposition library to give a weak notion of virtual machine, called a VN. The VNs route their traffic through one or more physical "core" machines that emulate the link characteristics of the interior topology. ModelNet has emulated topologies in excess of 10,000 links. However, it cannot emulate arbitrary computation in the core of a topology, which excludes simple applications like traceroute as well as more complex services like user-configurable dynamic routing, unless support for each feature is hardwired in (as has been done for DSR) [25].

Compared to Emulab, ModelNet is less transparent to applications and it is harder to provide performance monitoring, because it currently uses only a very weak notion of virtual machine. For example, it does not virtualize filesystem namespace, VNs cannot be multihomed, and it provides no network bandwidth isolation between VNs on the same physical host. ModelNet and the new Emulab are clearly complementary—ModelNet is perfect for generic network interiors, while the new Emulab

is strong in other ways.

Building on ModelNet and the Xen [3] virtual machine monitor, DieCast [11] uses time dilation to run large virtual experiments. In DieCast, time is "slowed down" inside of the virtual machines by an amount equal to the multiplexing factor, resulting in an experiment that takes much longer to execute, but which provides the illusion that the full capacity of the host CPU, network bandwidth, and other "time-scalable" resources are available to each virtual node. As a result of this time dilation, each DieCast virtual node has more of these resources available to it than ours do, but the overall efficiency of the testing facility is not improved. Thus, our virtual nodes are more appropriate for a shared facility. DieCast represents an alternative approach to scale up experimentation resources, bringing with it a different set of challenges to solve.

The Virtual Internet architecture [24] is a partially implemented model targeted to deploying virtual IP networks as overlay networks on the live Internet. The VI work identified most of the issues with link virtualization at the IP layer that we encountered at the Ethernet level. It focuses on correct implementation of virtual links when nodes can simultaneously participate in multiple topologies (concurrence), as multiple nodes in a single topology (revisitation) and when nodes in a virtual topology can themselves act as base nodes for other topologies (recursion). It does not virtualize other node resources.

Virtual machines have a long history, but we discuss only a few recent examples that have been used specifically to implement network emulation environments. This related work generally concentrates on node and/or network virtualization, but we provide a complete system including experimenter control, automated resource assignment and feedback directed virtualization.

IMUNES [32] is an integrated network emulation environment using FreeBSD jail-based virtual nodes and the "vimage" virtual network infrastructure work [30, 31] (which is now part of FreeBSD-CURRENT, but was not available when we started). Rather than virtualize pieces of the network stack, the authors virtualize the entire stack and associate an instance with each jail. While conceptually cleaner, the complete duplication of all network resources raises issues of kernel memory fragmentation. Their implementation provides some basic control over CPU usage that ours currently does not. Although IMUNES topologies can span multiple physical machines, they do not have the automation support to layout and control such topologies.

The node virtualization facility added to the Network Emulation Testbed (NET) [16] provides a lightweight virtual node mechanism in Linux based on virtual routing tables and custom Linux modifications. Their environment provides wireless as well as wired network emulation. The NET virtual networking implementation is analogous to ours, with their "vnmux" virtual interface and bridge taking the place of our "veth" device and the "NETshaper" replacing our Dummynet usage. Some degree of application transparency is achieved by using chvrf, a Linux chroot-like utility, to separate process and network name spaces. The NET work is highly complementary to ours in that it provides a Linux virtual node implementation as well as wireless network emulation that could be integrated with Emulab.

PlanetLab [20] is a geographically distributed network testbed, with machines time-shared among mutually untrusting users. PlanetLab uses Linux vservers [14] enhanced with a custom kernel module that provides enhanced resource isolation, including CPU and network bandwidth. Node virtualization is constrained by the fact that the nodes are subject to the restrictions of the site at which they reside. For example, since they cannot assume more than a single routable IP address is available per node, IP name space is not virtualized.

VINI [4] is a virtual network infrastructure designed to allow multiple, simultaneous experiments with arbitrary network topologies to run on a "real" shared physical network infrastructure. Specifically, PL-VINI is an implementation of VINI on PlanetLab nodes. It builds on top of PlanetLab vservers, adding virtual routers connected by virtual point-to-point links along with the ability to direct real Internet traffic through the resulting virtual network. The absolute performance of PL-VINI was poor due to the need to implement forwarding infrastructure in user mode on the PlanetLab Linux kernel. It also offers only rudimentary traffic shaping and topology setup mechanisms.

A new implementation of VINI called Trellis [5] improves the performance and capabilities of PL-VINI by moving the virtual networking into the Linux kernel, enabling faster packet forwarding and traffic shaping via standard Linux tools. We are currently collaborating with the VINI developers to bring VINI nodes under Emulab control, enabling the full power of Emulab's experiment creation and control infrastructure.

Auto adaptation, using an automated iterative process to best match a workload to available resources, is also not a new idea. One example is Hippodrome [1], a tool for optimizing storage system configurations. Hippodrome uses storage-relevant metrics (e.g., IOs/sec) to analyze a target workload. It feeds that information into a "solver" which uses modeling to find a good candidate storage architecture, then reconfigures the underlying storage subsystem accordingly. This process is repeated until a configuration is found that satisfies the workload's IO requirements.

Compared with our work, Hippodrome is focused on

a much narrower set of resources. They are concentrated on IO bandwidth where we must consider a workload's CPU, memory and network resource requirements as well as storage requirements. As a result, they can use more sophisticated and specialized analysis and design tools (e.g., storage system models), allowing quicker convergence on a suitable resource configuration.

9 Discussion and Conclusion

Our resource allocation and monitoring techniques do not assure the *timeliness* of events. In general, assured timeliness is expensive to provide, requiring real-time scheduling of CPU and links. However, we do provide two ways to address the issue, with another planned. First, the user's application-specific metrics, if they can be gathered on unmultiplexed nodes, serve as a safety mechanism to catch arbitrary performance infidelities. Second, the user can specify a shorter time period (the default is 1 second) over which the monitoring daemon will average, as it looks for overload. Finally, we may add a kernel mechanism that will detect if any resource use over very fine time scales, e.g., 1–10 msecs, has exceeded a user-settable threshold. Given this mechanism and typical Internet latencies, users can be quite confident that timing effects regarding network I/O have not affected their experiments.

Evaluation of packet timeliness and CPU scheduling effects remain to be done, but by offering the user application-level metrics directing adaptation, that is not essential. Exhaustive validation of the link emulation fidelity should be done, similar to the inter-packet arrival and time-variance analysis we do for mixed simulated/emulated resources [13]. Another issue is that our default mode of encapsulation decreases the MTU by a few bytes, which could affect some applications. In this case we support two other techniques that require no loss of MTU size: the virtual network devices can be configured to use fake MAC addresses in place of encapsulation, or to use 802.1Q VLAN tagging. We may add well-known OS resource isolation mechanisms such as proportional-share scheduling and resource containers. In a completely different but important area, some aspects of Emulab's Web-based user interface, such as its Graphviz-based topology visualization, are inconvenient to use on thousands of nodes. In response we have built on Munzner's hyperbolic three-dimensional graph explorer library[18] to provide an interactive "fish-eye" visualizer for Emulab, though have not yet put it into production use. Finally, our node support is limited to FreeBSD, yet many want Linux or Windows. When the Trellis work is mature we plan to adopt that to obtain equivalent support for Linux. We currently have Xen partially supported in Emulab, and are exploring VMware [27].

In conclusion, we have identified, designed, and implemented the many features necessary to support practical scalable network experimentation, and deployed them in a production system. We have shown that, by relaxing the constraints of conservative resource allocation, we can significantly increase the scale of topologies that we can support, or lower the required physical resources, with minimal loss of fidelity. In the future we will gather experience on how experimenters use the feedback and adaptation system, and evolve our system accordingly.

Acknowledgments

Many members of the Flux Research Group assisted with this work. We owe special thanks to Eric Eide for significant writing and LaTeX assistance, to Russ Fish for the Hypview visualizer, and to Sai Susarla for helping with evaluation.

We also thank the anonymous reviewers and our shepherd, Zheng Zhang, whose comments helped us to improve this paper. This material is based upon work largely supported by NSF grants 0082493, 0205702, and 0335296, and by hardware grants from Cisco Systems.

References

[1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running Circles Around Storage Administration. In *Proc. of the Conf. on File and Storage Technologies (FAST)*, pages 175–188, Monterey, CA, Jan. 2002.

[2] Apple Inc. Open Source Streaming Server. http://developer.apple.com/opensource/server/streaming.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, Oct. 2003.

[4] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. of SIGCOMM*, pages 3–14, Pisa, Italy, Sept. 2006.

[5] S. Bhatia, M. Motiwala, W. Mühlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford. Hosting Virtual Networks on Commodity Hardware. Technical Report GT–CS–07–10, Department of Computer Science, Georgia Tech, Jan. 2008.

[6] 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. 33(5):59–67, May 2000. (An expanded version is available as USC CSD TR 99-702b.).

[7] C. Dovrolis, P. Ramanathanm, and D. Moore. What Do Packet Dispersion Techniques Measure? In *Proc. of IEEE INFOCOM*, pages 905–914, Anchorage, AK, Apr. 2001.

[8] J. Duerig, R. Ricci, J. Byers, and J. Lepreau. Automatic IP Address Assignment on Network Topologies. Flux Technical Note FTN–2006–02, University of Utah, Feb. 2006. http://www.cs.utah.edu/flux/papers/ipassign-ftn2006-02.pdf.

[9] K. Fall. Network Emulation in the Vint/NS Simulator. In *Proc. of the 4th IEEE Symp. on Computers and Communications (ISCC)*, pages 244–250, Sharm El Sheik, Red Sea, Egypt, July 1999.

[10] GENI Planning Group. GENI Facility Design. GENI Design Document GDD–07–44, GENI, Mar. 2007. http://geni.net/GDD/GDD-07-44.pdf.

[11] D. Gupta, K. V. Vishwanath, and A. Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proc. of NSDI*, pages 407–421, San Francisco, CA, Apr. 2008.

[12] S. Guruprasad. Issues in Integrated Network Experimentation using Simulation and Emulation. Master's thesis, University of Utah, Aug. 2005. http://www.cs.utah.edu/flux/papers/guruprasad-thesis-base.html.

[13] S. Guruprasad, R. Ricci, and J. Lepreau. Integrated Network Experimentation using Simulation and Emulation. In *Proc. of the First Intl. Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, pages 204–212, Trento, Italy, Feb. 2005.

[14] Linux-VServer Project. http://www.linux-vserver.org/.

[15] P. Mahadevan, C. Hubble, D. Krioukov, B. Huffaker, and A. Vahdat. Orbis: Rescaling Degree Correlations to Generate Annotated Internet Topologies. In *Proc. of SIGCOMM*, pages 325–336, Kyoto, Japan, Aug. 2007.

[16] S. Maier, D. Herrscher, and K. Rothermel. On Node Virtualization for Scalable Network Emulation. In *Proc. of the 2005 Intl. Symp. on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 917–928, Philadelphia, PA, July 2005.

[17] METIS Familiy of Multilevel Partitioning Algorithms Web Page. http://www-users.cs.umn.edu/~karypis/metis/.

[18] T. Munzner. Exploring Large Graphs in 3D Hyperbolic Space. *IEEE Computer Graphics and Applications*, 18(4):18–23, 1998.

[19] S. Nanda and T. Chiueh. A Survey on Virtualization Technologies. Technical Report ECSL–TR–179, SUNY at Stony Brook, Feb. 2005.

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

[21] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *ACM SIGCOMM Computer Communication Review (CCR)*, 33(2):65–81, Apr. 2003.

[22] R. Scandariato and F. Risso. Advanced VPN support on FreeBSD systems. In *Proc. of the 2nd European BSD Conf.*, Amsterdam, The Netherlands, Nov. 2002.

[23] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of SIGCOMM*, pages 133–145, Pittsburgh, PA, Aug. 2002.

[24] J. D. Touch, Y.-S. Wang, L. Eggert, and G. G. Finn. A Virtual Internet Architecture. Technical Report ISI–TR–2003–570, Information Sciences Institute, Mar. 2003.

[25] A. Vahdat. Personal communication, May 2004.

[26] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. of the 5th Symp. on Operating Systems Design and Impl. (OSDI)*, pages 271–284, Boston, MA, Dec. 2002.

[27] VMware, Inc. VMware: A Virtual Computing Environment. http://www.vmware.com/, 2001.

[28] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the 5th Symp. on Operating Systems Design and Impl. (OSDI)*, pages 255–270, Boston, MA, Dec. 2002.

[29] J. Winick and S. Jamin. Inet-3.0: Internet Topology Generator. Tech Report CSE–TR–456–02, University of Michigan, 2002.

[30] M. Zec. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *Proc. of the FREENIX Track: 2003 USENIX Annual Tech. Conf.*, pages 137–150, San Antonio, TX, June 2003.

[31] M. Zec and M. Mikuc. Real-Time IP Network Simulation at Gigabit Data Rates. In *Proc. of the 7th Intl. Conf. on Telecommunications*, Zagreb, Croatia, June 2003.

[32] M. Zec and M. Mikuc. Operating System Support for Integrated Network Emulation in IMUNES. In *Proc. of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, Boston, MA, 2004.

[33] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proc. of IEEE INFOCOM*, pages 594–602, San Francisco, CA, Mar. 1996.