Feedback-directed Virtualization Techniques for Scalable Network Experimentation

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

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

University of Utah Flux Group Technical Note FTN–2004–02
May 2004

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 relatively 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. The new 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 thousands of virtual nodes. This paper 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 [27] and PlanetLab [18] 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 new 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. One of our primary motivation is scale: i.e., to support larger and more complex emulated environments, and also 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 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.

Largely sponsored by NSF grants ANI-0082493, ANI-0205702, and CNS-0335296, and hardware grants from Cisco Systems.
Contact: testbed@flux.utah.edu.
URL: http://www.cs.utah.edu/flux/papers/virt-ftn2004-02-base.html

Our motivating 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. Our primary dimensions are: i) "swapin" performance: the time to reliably instantiate an experiment, which affects system throughput and Emulab's interactive usage model; ii) monitoring, control, and visualization of testbed experiments, both by the user and the system; iii) resource use—the number of physical machines and links required for a particular virtual topology; iv) the user's time spent in customizing instrumentation and management infrastructure.

However, no matter how scalable the system, 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 these goals, we multiplex virtual entities onto the physical infrastructure, using four key design techniques:

- Using the minimum degree of virtualization that will provide sufficient transparency to applications. In our case, this means that the majority of the OS and network mechanisms used by virtual entities are identical to the native mechanisms.
- Exploiting hierarchy, both in real computer networks (which the user's virtual topology represents) and in the physical realization of those networks. Our resource allocator relies on implicit hierarchy in the virtual topology to reduce its search space, and our IP address assigner infers hierarchy in order to provide realistic IP addresses. Our testbed control system exploits the hierarchy between virtual nodes and their physical hosts; for example, by extensive proxying, caching, and acting on many virtual entities simultaneously.
- Optimistic automated resource allocation. The system or the user makes a "best guess" at the resources required, which are fed into a powerful resource assigner that uses combinatorial optimization.
- Use of feedback to adaptively allocate resources. Both in training runs and during normal use, system-level and optional application-specific metrics are monitored. The metrics are used to detect overload or (sometimes) underload conditions, and to guide resource re-allocation, if required. Emulab can automatically execute this adaptive process by leveraging its high degree of automation.

This paper makes the following contributions: (1) It describes levels of virtualization that are appropriate for this domain, and discusses some of the design tradeoffs. (2) 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. (3) It presents a new feedback-directed technique to support virtualization and scaling. (4) It describes a new algorithm for assigning realistic IP addresses. (5) It provides a preliminary experimental evaluation of various aspects of the system. (6) The system it describes provides a useful new facility, and, with the exception of auto-adaptation, is proven in public production use.

The rest of this paper is organized as follows. Section 2 provides background on Emulab and its use. Section 3 describes our node and network virtualization mechanisms, as well as our algorithm to assign IP addresses hierarchically. Section 4 covers automated resource assignment, and Section 5 outlines how we exploit hierarchy after resources are allocated. Sections 6 describes our feedback-directed adaptation and presents experimental results. Section 7 describes related work, and we then discuss limitations of our system, future work, and conclude.

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* 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 Ethernet interfaces to switches that are themselves connected with multigigabit 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 3.5 years of statistics on 700 users, doing 10,000 swapins, allocating 155,000 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

Design Alternatives

There are many possible ways to implement some notion of a "virtual node." Which strategy is chosen depends upon the requirements of the environment relative to the following attributes:

Application transparency: the extent to which name spaces are isolated. (Can the application run unchanged?)

Application fidelity: the extent to which resources are isolated. (Does the application get the resources it needs to function correctly?)

System capacity: the amount of virtualization overhead. (How many vnodes can we host?)

System flexibility: the level of virtualization (can we run multiple OSs?) and the degree of portability (can we run on a wide range of hardware?)

Here, we briefly summarize some of the alternatives for virtual node implementations.

The big hammer in the virtualization toolbox is the classic virtual machine monitor (VMM). Classic VMMs like VMware [25] provide complete virtualization of an architecture, typically presenting an instance of the underlying physical hardware. Full virtualization provides the ultimate in flexibility, allowing arbitrary unmodified operating systems and their applications to run on the same host concurrently. But full virtualization comes at a cost both in performance and host resources.

A recent trend, represented by Xen [1], is so-called *paravirtualization*, in which the VMM presents an architecture that is largely the same as the underlying hardware, but in some cases provides abstractions that are more closely aligned with OS expectations. Since the architecture is not a complete virtualization, OSes must be "ported" to run on the VMM architecture. Once ported, an OS and its applications typically perform much better than under a classic VMM. However, there is still considerable cost to running an application inside its own instance of an OS compared to running it in a process on a traditional OS.

A third alternative is to make modifications to an existing operating system to provide limited virtualization features for processes. Techniques used here include overriding library interfaces (ModelNet [24]), intercepting system calls (UML [2]), or adding features to the OS kernel (BSD jails [11], Linux vservers [13]). The emphasis is typically on providing namespace (e.g., filesystem or network) isolation rather than resource isolation since the latter is much harder and often not needed. Some resource isolation may be available courtesy of pre-existing mechanisms in the kernel. Here we achieve very low virtualization overhead and transparency for applications, but require OS modifications and most likely give up resource isolation.

Finally, in many environments, it is sufficient to just run multiple copies of an application as multiple processes on a single machine. This strategy is not in general transparent to applications. Applications will need to be modified, or configured at run time, to reflect that they are sharing resources with other instances; e.g., a config file that tells it how much memory to use, where to get or store its data in the filesystem, what network ports to use, and how to identify itself to other programs. There is no virtualization overhead because the "virtualization" is static, embodied in the configuration file or program itself. Thus you get the highest performance, since applications are running "directly on the hardware," at the expense of almost any level of isolation.

Emulab Virtual Nodes: the Abstraction

A virtual node is a first class object in Emulab. Abstract virtual nodes have five key attributes:

1. They are multiplexed onto physical hosts.

2. They have independent namespaces: each has its own hostname, interfaces, IP addresses, routing table, filesystem, and process space. Two virtual nodes running on the same host cannot see each other.

3. Each virtual node can be controlled independently of the physical node hosting it. Virtual nodes can be individually booted, rebooted, and halted. Users can login directly to the virtual nodes using `ssh` and Emulab's event system can directly control a virtual node. Emulab state-machine-driven "node lifecycle" control and monitoring system treats each virtual node independently.

4. There is strong communication isolation between virtual nodes on a physical host: they can communicate only through the network. There is weaker isolation from the host, so it can bootstrap and proxy services for its virtual nodes.

5. Virtual nodes support controllable, shaped links and LANs to other nodes, both virtual and physical.

Emulab Virtual Nodes: the Reality

Application transparency is important in the Emulab environment, requiring at least namespace isolation 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. Moreover, since physical nodes are dedicated to experiments, hosting only vnodes for that experiment, we do provide inter-experiment resource isolation. Finally, we hoped to achieve at least a ten fold multiplexing factor on relatively low-end PCs (600 MHz, 256 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.

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 jailed processes. 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 to support inter-vnode disk space isolation. 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. Disk quotas aren't useful either: within the jail's name space, files are not restricted to a single uid or even subset of uids; they can be owned by anyone.

Our design uses BSD *vnode disks* to create a regular file with a fixed size and expose it via a disk interface. We create empty virtual disks by seeking to the end; that allocates no actual blocks in the underlying filesystem, so is space-efficient in the typical case that 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 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.

3.2 Virtual Links and LANs

3.2.1 Design Issues

In a general context, virtual links provide a way of multiplexing many logical links onto a smaller number of phys-

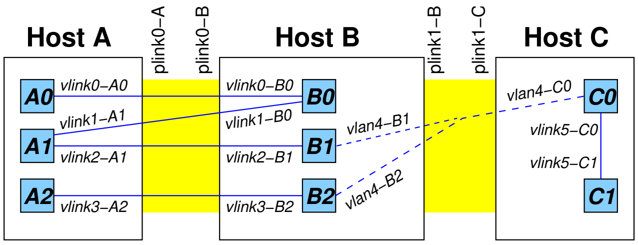

Figure 1: A network topology illustrating routing issues due to the multiplexing of virtual nodes and links. Large boxes represent physical nodes and links, while small boxes and lines (with *italic labels*) represent virtual nodes and links. Virtual network interfaces (*vlinks*), virtual LANs (*vlans*),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

ical links. In this light, virtual links can be used to provide a higher-degree of connectivity to nodes, whether those nodes are virtual or physical. In the context of virtual nodes, our discussion of virtual links includes not only this multiplexing capability but also the network namespace isolation issues and the subtleties of interconnecting virtual nodes within, and between, physical nodes.

The interesting characteristics of virtual links are:

Level of virtualization. Virtual link implementations can present either a virtual link layer by providing a virtual Ethernet device or a virtual IP layer by using multiple IP addresses per physical interface. The former is more flexible, allowing traffic other than IP, but such flexibility is often not needed.

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

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

Ability to co-locate virtual nodes. Three factors related to the implementation of virtual links influence which, if any, virtual nodes in the same topology or virtual nodes in different topologies can be co-located on a physical node.
First, if virtual links are implemented using IP aliases on shared physical interfaces, then there are restrictions on what addresses can be assigned to the interface. For example two nodes in different topologies could not have the same IP address. Virtual LANs provide another example. As shown in Figure 1, virtual nodes B1, B2, and C0 are part of a virtual LAN spanning two physical nodes. As such, they must have IP addresses in the same subnet. However, most OSes prohibit assigning addresses in the same subnet to the same interface. Hence, B1 and B2 could not be co-located as shown.
Second, even with per-vnode interfaces, it is possible that two co-located nodes in the same topology might have routing "short-circuited" by the OS if it recognized that both interfaces refer to the local host. For example, traffic between A0 and A2 might be delivered directly rather than following the topology.
Finally, if virtual links use a shared routing table, then two co-located nodes cannot have different next hop addresses for the same destination. For example, in the figure, packets sent from A0 to C0 will pass through host B twice. B0's next hop for C needs to be A while B1's needs to be C. This is known as the "revisitation" problem [22]. Further, even with separate routing tables, incoming packets to B need context to determine which routing table to use. This information needs to be conveyed in the packet, either through encapsulation or "fake" link addresses.

3.2.2 Emulab Virtual Links

Virtual network interfaces. While the BSD 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`. Further, when packets leave a physical host they lose the identity of the virtual node that was the most-recent hop of the packet, meaning that jails cannot implement node revisitation. Fake MAC addresses, used by some network emulators, are also inadequate; space constraints preclude more detail here.

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). Virtualization gives us per-jail interfaces above the physical interface to which we can apply jail-specific ipfw/dummynet rules or on which the jail processes can operate. Bridging allows the correct routing of packets at the link level so that virtual interfaces only receive the packets that they should. Encapsulation preserves the virtual link information necessary to implement revisitation when crossing physical links, without making any assumptions about the switching fabric.

Although there exist virtual ethernet drivers, bridging code, and encapsulation devices, to our knowledge we are the first to integrate the three concepts into one.

Virtual routing table. 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 shared IP infrastructure, in particular, the routing table. In BSD routing tables, 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. We have adopted and extended the work of Scandariato and Risso [20] 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. We use these IDs as the glue that bind together jails, virtual interfaces and routing tables.

3.2.3 Realistic IP Address Assignment

In an Emulab experiment, users have the option of either assigning IP addresses themselves or letting Emulab choose IP addresses automatically. Most experimenters choose the latter, as manual assignment can be tedious and error-prone. Some topologies derived from the real Internet include IP addresses. However, most topology generators are intended for use with simulation and therefore do not include them. Additionally, requiring users to always designate IP addresses would violate our goal of generality and would preclude the use of new experimental generators.

Our objective in assigning IP addresses is to lay them out in a realistic fashion. In the Internet, ISPs are typically assigned blocks of IP addresses. These ISPs then delegate subnets within these blocks to smaller ISPs and customers. IP addresses assigned in a manner approximating this have three key strengths: (1) The IP addresses are intuitive to experimenters when they need to find out how packets move through their topology. (2) The size of routing tables can be greatly reduced using standard CIDR routing. (3) Realistic IP addresses cause realistic behavior from dynamic routing protocols such as OSPF and BGP.

To assign addresses in blocks like the internet, we must to take into account whatever hierarchical properties the input topology has. To infer this hierarchy, we search for groups of nodes in the topology graph that are strongly-connected, with a relatively small number of connections to the rest of the graph. Hierarchy inference is a problem in many domains. Our overall strategy for solving this problem is to recursively divide the graph into sub-partitions. Others [10, 7] use a similar strategy to perform database map queries.

IP address assignment is done using the following steps:

1. *Invert the graph.* Take each LAN as a vertex and each host as a hyper-edge in the new graph. For the purposes of this algorithm, a link is considered a special case of a LAN, with only two members. This is necessary because while a host can be multi-homed, a LAN can belong to only one subnet. The graph inversion process is described more formally in [29].

2. *Partition the graph.* Number each partition, allocating the minimum number of bits required to represent the number of partitions.

3. *Recursively partition* by repeating step two on each partition, continuing until we reach a partition of size one, or exhaust the available bit space.

4. *Associate a subnet with each partition.* Number each LAN within each partition and each node within each LAN.

5. *Combine* the numbers from our repeated application of step two with the two numbers from step four to give each interface a unique IP address.

Currently, the algorithm used to divide the graph is a heuristic for the NP-complete graph-partition problem. This graph partitioning algorithm, which is used in step two, must be given the number of partitions to create. If there are more or less than the ideal number of partitions, the quality of the partitioning is poor. A poor partitioning leads to unrealistic IP address assignment. On the other hand, the partitioner we use, METIS [15], is fast, running in sub-second times for most partitionings. We search for a partition size that minimizes the average number of border routers in each partition.

The keystone of the search is the scoring algorithm. A border in this context is a cut edge. Therefore the cost of the borders is the aggregate weight of the cut edges. However, the number of cut edges is not enough. In most circumstances, there will be more cut edges as the number of partitions increases. This means that it is not very useful to simply count up the aggregate weight cut, because this will bias the scoring towards small numbers of partitions. Instead, it is useful to find the average border size (cut weight) per partition. c is the aggregate cut-edge weight, p is the number of partitions, and the score $S = \frac{c}{p}$.

We are in the process of evaluating an alternative to the above search; the use of ratio-cuts. The use of ratio cuts is useful in the field of VLSI design [26] and we are adapting it for networking. A ratio cut is a cut which minimizes the ratio score on a particular graph. The ratio score of a cut depends upon the weight which is cut and the sizes of the partitions. If c is the aggregate cut-edge weight of a bipartitioning, and $|A|$ and $|B|$ are the number of vertices in the associated two sub-partitions, the ratio score $R = \frac{c}{|A| \cdot |B|}$.

Though the ratio cut problem is in NP, several methods exist to give approximations [26, 17] in linear time. These heuristics are a promising approach to better partitioning.

It is a nontrivial task to quantitatively evaluate how realistic an IP address assignment is. However, the reason that IP addresses are assigned by block is to facilitate CIDR routing. This means that assignments can be evaluated by the improvement they gain when using CIDR routing.

We found real world topologies mapped by Rocketfuel [31] and since Rocketfuel retains the IP addresses in the topologies it maps, we used the actual assignment as our benchmark. We also compare against a naive IP address assigner which chooses the addresses arbitrarily.

The mechanism for comparison is a route optimizer. The optimizer searches the graph and aggregates routes into a subnet whenever those routes are in the same subnet and have the same first hop.

Table 1 shows that the current algorithm is much better than arbitrary assignment, but there is still room for improvement. Finding the ratio-cut is where that improvement will likely be found.

Topology	Real	Emulab	Random
VSNL (India)	219	304	403
EBONE (Europe)	10841	12532	20096
Exodus (US)	21113	19722	26140
Tiscali (Europe)	8810	15450	19863

Table 1: Aggregate number of routes after optimization

4 Automated Resource Assignment

Emulab automatically maps an experimenter's requested virtual topology onto the available physical resources. It decides which virtual nodes to place onto which physical nodes in such a way as to avoid overloading hosts and links. This problem has been shown to be NP-hard [19]. The virtual and physical resources to be mapped include hosts, routers, switches, and the links that connect them. Experimenter requests, such as nodes with special hardware or software, must be satisfied, and bottleneck links and other scarce resources in the physical topology should be conserved when physical resources are shared, as they are in Emulab; in contrast, related systems such as ModelNet [24], do not space-share testbeds.

Emulab finds an approximate solution to the network testbed mapping problem by taking a combinatorial optimization approach. It uses a complex solver called `assign` [19] that is built around a simulated annealing core. We found, however, that Emulab's existing `assign` was not sufficient for mapping virtual node experiments.

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. Also, `assign` traditionally does conservative resource allocation; that is, it assigns nodes and links with the assumption that they will always fully utilize resources. While this makes sense for artifact-free emulation, it is contrary to our goal of using feedback to provide large-scale emulation.

Second, `assign` had scaling problems due to the fact that virtualization allows for topologies an order of magnitude larger than one-to-one emulation. 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 exploit the natural structure of the virtual topologies given to `assign`.

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. 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 different resource consumption.

So, we added more packing schemes to `assign`. In one, 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. Another 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. This scheme is extensible. It can be applied to any other resource that can be represented numerically, such as as interrupt load or disk bandwidth. It can even be used for higher-level metrics, such as sustainable event rate for discreet event simulators such as *ns*.

The resource-modeling scheme is particularly useful for feedback-based auto-adaptation. The values fed in for CPU and memory consumption of a virtual node, for example, can simply be obtained by taking measurements of a running 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 are conceptually similar, there are a different set of issues to address for 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. First, 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. Although the bandwidth on a loopback interface is high, there are practical limits on it, so `assign` must take this finite resource into account.

Second, 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. This is at odds with our goal of providing best-efforts, 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 100Mpbs, the nodes in this LAN may never transmit data at the full line rate. Thus, if `assign` were to allocate the full 100Mbps 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.

Finally, `assign` currently makes the assumption that links are symmetric; that is, that they use the same bandwidth in both directions. Clearly, there are large classes of applications for which this assumption is false. In most client-server applications, for example, servers transmit much more data than they receive. Requiring the bandwidth to be specified as a single number causes some of the bandwidth into the server to be wasted, since it will never be used. Since this clearly results in less efficient resource use, we are extending `assign` to support asymmetric bandwidth specifications.

4.2 Improving `assign`'s Scaling

4.2.1 Searching the Solution Space

Our first attempts at tackling scaling issues were aimed at improving the way in which `assign` searches through the solution space. `assign` reduces its search space by finding groups of homogeneous physical nodes and combining them into equivalence classes. This strategy breaks down with a high degree of multiplexing, however, because a physical node that has been partially filled is no longer equivalent to an empty node. We have addressed this by making these equivalence classes dynamic as `assign` runs. The result is that `assign` can avoid large portions of the solution space which are equivalent do not need to be searched.

Another improvement to the search strategy came from the observation that, in a good solution, two nodes that are adjacent in the virtual topology will have a high probability of being placed on the same physical node. So, we modified the function `assign` uses to select a new mapping to try for a virtual node. In our modified version, rather than selecting a random physical node, we, with some probability, select a physical node that one of the virtual node's neighbors has already been assigned 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, the runtime was still much too long to be acceptable for our purposes. Our strategy for making this problem more tractable is to 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 [21] or from topology generators designed to create Internet-like networks, such as GT-ITM [31] and inet [28]. 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 LANs that represent clusters, groups of workstations, etc.

We exploit the structure of the input topology by applying a coarsening pre-pass to the virtual graph before running `assign`. This reduces the solution space that `assign` must search, reducing its runtime. 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 leaf LANs onto a single physical node as possible. So, we find leaf LANs, and combine all members who are *only* members of that LAN single virtual node.

The second algorithm uses a graph partitioner, METIS [15], to partition up the virtual graph. We choose a number of partitions such that the average partition will fit on the "smallest" available physical node. We then turn each partition returned by the partitioner into a virtual node. The quality of the partitions returned by the partitioner are 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 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 virtual nodes.

There are many ways in which these coarsening algorithms can 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

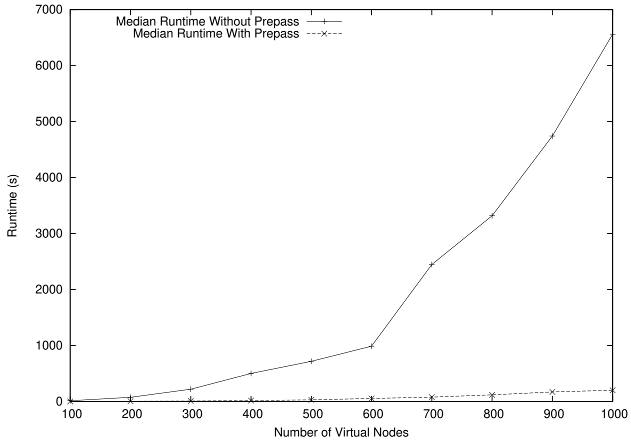

Figure 2: Median runtime of `assign` with and without a coarsening pre-pass.

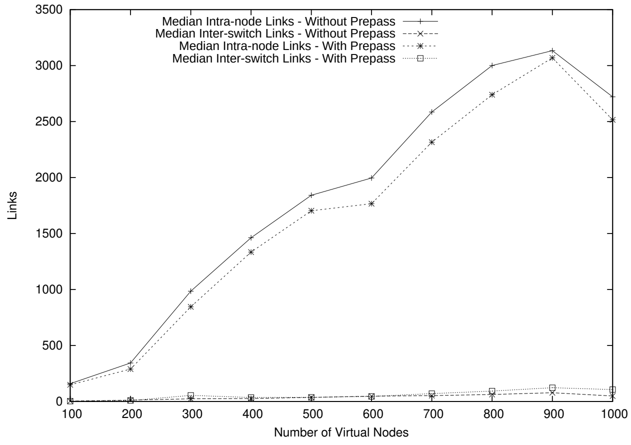

Figure 3: Number of intra-node and inter-switch links found by `assign`. Larger numbers of intra-node links are better, and smaller numbers of inter-switch links are better.

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 [31] 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 shows the median runtimes for these tests on a 1.5 GHz Pentium IV. We can see that the time savings are significant as we scale up the number of virtual nodes, going from a factor of 14 at 100 nodes to a factor of 28 at 1000 nodes. 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; 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 Exploit Physical Hierarchy

In addition to the previously described routing 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.

Nodes in Emulab retrieve their configuration information at boot time from a central server. The most straightforward way to boot virtual nodes is to have them individually perform this self-configuration. However, some information about the virtual nodes is required by the host in order to bootstrap them. So, we are able to save substantial server load by having the physical host download the complete configuration for each of its virtual nodes, and then populate a cache of this information inside the virtual nodes. Additionally, setup information that is the same for all virtual hosts can often be replicated more efficiently, possibly even as the side-effect of another operation. For example, customization of password and group files is actually done on the physical host and is propagated to virtual nodes as a consequence of cloning a root filesystem.

Another type of proxying is used in the Emulab event system. This system is used to schedule and then distribute events to agents on nodes, enabling dynamic control over aspects of an experiment such as modifying link parameters and starting and stopping traffic generators and other programs. Traditionally, each event agent on a node holds an open TCP connection to the event server. To reduce the connection load, virtual nodes instead connect to a proxy running on the physical host. This proxy holds a single connection to the central event server, listen for and forwarding events for all of its virtual nodes.

To reduce the load on the NFS server, instead of having each virtual node mount it directly, the physical host performs the mount, and then re-exports the shared filesystem to each virtual node.

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 being 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 single uniprocessor host. However, vnode on distinct physical nodes 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 one physical node hosting 10 vnodes to the LAN. In the base case of one physical node with 10 vnodes in the LAN, setup, including topology mapping, node configuration and startup, required 194 seconds. At 80 vnodes 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 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.

A user of the testbed has two options for adapting their experiment: a manual, single-stage "training" run that requires little effort by the user; and a multi-stage *automatic experiment adapter* that requires additional effort. The first option does not require the experiment to be fully automated, and is thus suitable for an interactive style of experimentation. Users can simply log in to their 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, the simplistic manual approach will not work for large topologies, so we offer the second option and require the user to follow these steps:

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 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 Auto-Adaptation

Ensuring application fidelity when multiplexing virtual nodes can be achieved quickly and accurately through monitoring of 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 vhost'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.

We gather a number of resource use 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 are able to determine the resource demand for individual vnodes. The other global statistics allow us to ensure that the system as a whole is not overloaded.

Initially, there is no feedback data to work from, so a bootstrap phase is done to get some clean resource usage data. Bootstrapping is simply a matter of forcing a one-to-one mapping by having each vnode reserve an entire pnode. Once the bootstrap resource data has been collected, the system can increase or decrease the reservations until it arrives at a maximally dense packing factor with vnode resource use that is consistent with the one-to-one mapping. At this point, the user will probably want to increase the size of the topology. The simplest approach would be 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 clear 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.

Using a similar mechanism and a modified version of *nse* [5], an Emulab experiment can incorporate purely simulated nodes and networks. As described in a thesis [6], these simulated entities can now 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 "falling behind" does not turn out to be black and white. Our infrastructure can adaptively remap simulated networks similarly to the way it handles virtualized nodes and links.

6.2 Fidelity Results

In this section we make a preliminary evaluation of the effect on emulation fidelity of increasing co-location of virtual nodes on physical nodes, using both fine grain measurements and real applications.

Microbenchmarks

To get a lower-level view of fidelity with increasing co-location, we performed an experiment in which we ran the `pathrate` [3] bandwidth-measurement tool between pairs of nodes co-located on the same physical host. Each pair of nodes was connected with a T1-speed (1.5Mbps) 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 1Kbps, with a standard deviation across runs of `pathrate` of 0.004.

Applications

We ran a synthetic peer-to-peer file sharing application called *Kindex*, that 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 a while, all clients end up caching all files, at which time we stop the experiment.

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

In order to demonstrate application transparency, 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

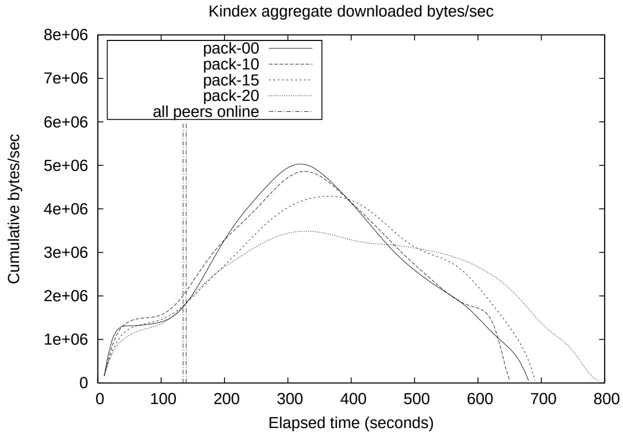

Figure 4: Cumulative system bandwidth for co-location factors of 0, 10, 15 and 20. "All peers online" is the point in time where all 60 peers are running and downloading files.

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.

6.3 Adaptation Results

We evaluated our feedback system in two scenarios: a Java-based web server and clients and the Bittorrent peer-to-peer file distribution system.

We first ran a Java-based web server on one host with 69 clients continually downloading a 64KB 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 2Mb LANs in order to simulate conventional cable modem clients. Forty clients were directly connected to a single router using 2Mb multiplexed links to simulate conventional DSL modems. Finally, 20 clients were directly connected to a single router using 56Kb multiplexed links, to simulate phone modem clients. The feedback loop required three iterations to reach an acceptable application fidelity, the results are shown in Table 2. 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 pnodes 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 by 20%, which is enough to isolate the server and return the application metrics to their original one-to-one values, without allocating anymore pnodes. 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.

Metric	**2Mb LAN**	**2Mb Link**	**56Kb Link**
74 vnodes on 74 pnodes			
Avg. Transaction Rate	1.19	2.29	0.09
Avg. Response Time (s)	0.84	0.43	10.67
Packed onto 7 pnodes 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 pnodes after second iterations			
Avg. Transaction Rate	1.19	2.29	0.09
Avg. Response Time (s)	0.84	0.43	10.70

Table 2: Performance of clients continually downloading a 64KB file in different vnode mappings.

To demonstrate scaling a real application to large topologies that cannot fit in a one-to-one mapping on our existing infrastructure, we ran the BitTorrent peer-to-peer file distribution program on a 310 node network packed onto 74 physical nodes. The topology consisted of 300 clients communicating over 2Mb LANs or links, a single "seed" node with a 100Mb 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.

The adaptation mechanism can also accomodate applications that have throughput constraints as well as timing sensitivity. We tested the Darwin Streaming Server sending a 100Kbps 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 100Mb, 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 3). The oversubscription of network bandwidth thus clears a path for time sensitive packets.

7 Related Work

The ModelNet network emulator [24] 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 operating systems, using IP aliasing and a socket interposition library to

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

Table 3: Interpacket gap of clients receiving a 100Kbps video and audio stream in different configurations where the physical link is shared and not shared. The values are the median of five runs.

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 recently been done for DSR) [23].

Compated 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, VN's 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. Therefore, we and the ModelNet team plan to integrate ModelNet into Emulab, dynamically allocating and configuring Modelnet "cores" for parts of the topology that use only generic router functions.

The Virtual Internet architecture [22] 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.

The "vimage" virtual network infrastructure work [14, 30] is similar to our FreeBSD jail-based implementation. 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 some issues with kernel memory fragmentation. Their implementation provides some basic control over CPU usage that ours currently does not. Although their topologies can span multiple physical machines, and new work is creating a GUI-based configuration tool [8], they do not have the automation support to control large topologies.

The vBET emulation environment [9] is built around an enhanced version of UML [2], a Linux virtual machine that runs as a process on unmodified Linux. UML's base performance is poor, obtaining 20–50% of base Linux performance on benchmarks involving significant I/O [1]. In contrast to our approach which uses native network mechanisms, vBET simulates hub and router devices, which, for example, cannot broadcast. vBET does not support topologies spanning multiple physical machines, so cannot emulate large topologies.

PlanetLab [18] is a geographically distributed network testbed, with machines time-shared among mutually untrusting users. PlanetLab uses Linux vservers [13] 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. Currently, it is impossible to reliably automate configuration of even modest numbers of PlanetLab vservers if overall setup speed is a requirement [12], apparently due to defects in the central PlanetLab service.

In the longer term the new Xen VMM offers some compelling features to network emulators. Xen provides good isolation and control of CPU, memory and disk resources, though network controls are not fully realized. It purports to host up to 100 simultaneous active virtual machines on "modern servers," and supports several popular operating systems.

Other. ACME, the "Application Control and Monitoring Environment" [16], provides scalable control and monitoring infrastructure, including distributed sensors and actuators. ACME or its ideas would be an alternate way for Emulab to provide online monitoring of overload conditions, should greater scalability be required.

Our CPU Broker [4] work has partially inspired our feedback-based approach, and several principles are similar. The broker mediates between multiple real-time tasks and an RTOS; using feedback and policies, it adjusts CPU allocations accordingly. It connects to its monitored tasks non-invasively, with user-provided proxies transforming resource use data into predictions. The CPU Broker reallocates resources on a fine time scale, while Emulab reallocates at a coarse time scale.

8 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 an a shorter time period (the default is 1 second) over which the monitoring daemon will average, as it looks for overload. Of course, the daemon may then consume excess resources itself, but since it runs on the user's own nodes, the testbed infrastructure is not threatened. Finally, we are adding a kernel mechanism that will report if any resource use over very fine time scales, e.g., 1–10msecs, has exeeded a user-settable threshold. Given this mechanism and typical Internet latencies, a user can quite confident that timing effects regarding network I/O have not affected his experiment.

In general, 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 needs to be done, similar to the inter-packet arrival and time-variance analysis we do for mixed simulated/emulated resources [6]. Another issues is that our encapsulation decreases the MTU by a few bytes, which could affect some applications. Many switches and NICs support larger MTU sizes, including ours; we are implementing that. In general, we could and will add well-known OS resource isolation mechanisms such as proportional-share scheduling and resource containers. Finally, our support is limited to FreeBSD, yet many want Linux or Windows. Clearly, we could port our work to Linux vservers, but we prefer to explore the Xen alternative.

In conclusion, 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 gathering experience on how experimenters use the feedback and adaptation system, and evolve our system accordingly.

References

[1] 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 Symposium on Operating Systems Principles*, pages 164–177. ACM Press, 2003.

[2] J. Dike. A User-Mode Port of the Linux Kernel. In *5th Annual Linux Showcase & Conference*, Oakland, California, 2001.

[3] C. Dovrolis, P. Ramanathanm, and D. Moore. What Do Packet Dispersion Techniques Measure? In *Proc. of IEEE INFOCOM'01*, May 2001.

[4] E. Eide, T. Stack, J. Regehr, and J. Lepreau. Dynamic CPU Management for Real-Time, Middleware-Based Systems. In *Proc. Tenth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2004)*, Toronto, ON, May 2004.

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

[6] S. Guruprasad. Issues in Integrated Network Experimentation using Simulation and Emulation. Master's thesis, University of Utah, May 2004. Draft. www.cs.utah.edu/flux/papers/guruprasad-draftthesis-base.html.

[7] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Effective Graph Clustering for Path Queries in Digital Map Databases. In *Proc. of ACM Conference on Information and Knowledge Management*, pages 215–222, 1996.

[8] IMUNES – An Integrated Multiprotocol Network Emulator/Simulator (Web site), May 2004. http://www.tel.fer.hr/imunes/.

[9] X. Jiang and D. Xu. vBET: a VM-Based Emulation Testbed. In *Proc. of ACM Workshop on Models, Methods and Tools for Reproducible Network Research*, Karlsruhe, Germany, Aug. 2003.

[10] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):409–432, May/June 1998.

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

[12] J. Lepreau. Message to Planetlab-Arch Mailing List. http://-lists.planet-lab.org/pipermail/arch/2004-March/000051.html, Mar. 2004.

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

[14] M. M. Marko Zec. Real-Time IP Network Simulation at Gigabit Data Rates. In *Proc. of the 7th Intl. Conference on Telecommunications*, Zagreb, June 2003.

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

[16] D. Oppenheimer, V. Vatkovskiy, H. Weatherspoon, J. Lee, D. A. Patterson, and J. Kubiatowicz. Monitoring, Analyzing and Controlling Internet-Scale Systems with ACME. www.cs.berkeley.edu/∼davidopp/pubs/draft22d.pdf, Oct. 2003.

[17] S. B. Patkar and H. Narayanan. An Effcient Practical Heuristic For Good Ratio-Cut Partitioning. In *Proceedings of the 16th International Conference on VLSI Design*, 2003.

[18] 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.

[19] R. Ricci, C. Alfeld, and J. Lepreau. A Solver for the Network Testbed Mapping Problem. *SIGCOMM Computer Communications Review*, 33(2), Apr. 2003.

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

[21] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of SIGCOMM'02*, Aug. 2002.

[22] J. Touch, Y. shun Wang, L. Eggert, and G. G. Finn. A Virtual Internet Architecture. Technical Report ISI-TR-2003-570, Information Sciences Institute, 2003.

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

[24] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 271–284, Boston, MA, Dec. 2002.

[25] VMWare, Inc. VMWare: A Virtual Computing Environment. http://-www.vmware.org/, 2001.

[26] Y.-C. Wei and C.-K. Cheng. Ratio Cut Paritioning for Heirarchical Designs. *IEEE Transactions on Computer-Aided Design*, 10(7):911–921, July 1991.

[27] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

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

[29] K. Yocum, E. Eade, J. Degesys, D. Becker, J. Chase, and A. Vahdat. Toward Scaling Network Emulation using Topology Partitioning. In *Proc. of MASCOTS 2003*, Oct. 2003.

[30] M. Zec. Implementing a Clonable Network Stack in the FreeBSD Kernel. In *Proc. of the 2003 USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[31] E. W. Zegura, K. Calvert, and S. Bhattacharjee. How to Model an Internetwork. In *Proceedings of IEEE Infocom '96*, San Fancisco, CA, Mar. 1996.