Scalability and Accuracy in a Large-Scale Network Emulator*

Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan,
Dejan Kostić, Jeff Chase, and David Becker
Department of Computer Science
Duke University
{*vahdat,grant,walsh,priya,dkostic,chase,becker*}*@cs.duke.edu*

Abstract

This paper presents ModelNet, a scalable Internet emulation environment that enables researchers to deploy unmodified software prototypes in a configurable Internet-like environment and subject them to faults and varying network conditions. Edge nodes running user-specified OS and application software are configured to route their packets through a set of ModelNet core nodes, which cooperate to subject the traffic to the bandwidth, congestion constraints, latency, and loss profile of a target network topology.

This paper describes and evaluates the ModelNet architecture and its implementation, including novel techniques to balance emulation accuracy against scalability. The current ModelNet prototype is able to accurately subject thousands of instances of a distrbuted application to Internet-like conditions with gigabits of bisection bandwidth. Experiments with several large-scale distributed services demonstrate the generality and effectiveness of the infrastructure.

1 Introduction

Today, many research and development efforts investigate novel techniques for building scalable and reliable Internet services, including peer-to-peer networks [17, 19, 20], overlay networks [1, 8], and wide-area replication [10, 23]. These projects and many others run on large numbers of cooperating nodes spread across the Internet. To test and evaluate such systems, their developers must deploy them in realistic scenarios, i.e., large, structured, dynamic wide-area networks.

* This work is supported in part by the National Science Foundation (EIA-9972879), Hewlett Packard, IBM, Intel, and Microsoft. Vahdat is also supported by an NSF CAREER award (CCR-9984328).

Unfortunately, it is difficult to deploy and administer research software at more than a handful of Internet sites. Further, results obtained from such deployments "in the wild" are not reproducible or predictive of future behavior because wide-area network conditions change rapidly and are not subject to the researcher's control. Simulation tools such as *ns* [15] offer more control over the target platform, but they may miss important system interactions, and they do not support direct execution of software prototypes.

This paper advocates *network emulation* as a technique for evaluating Internet-scale distributed systems. Network emulators subject traffic to the end-to-end bandwidth constraints, latency, and loss rate of a user-specified target topology. However, emulation has typically been limited to systems that are relatively small and static.

We present the design, implementation, and evaluation of ModelNet, a scalable and comprehensive large-scale network emulation environment. In ModelNet, unmodified applications run on *edge nodes* configured to route all their packets through a *scalable core* cluster, physically interconnected by gigabit links. This core is responsible for emulating the characteristics of a specified target topology on a link-by-link basis, capturing the effects of bursty flows, congestion, etc. In this context, this paper makes the following contributions:

- We show that a single modern server-class PC can accurately emulate bandwidth, latency, and loss rate characteristics of thousands of flows whose aggregate bandwidth consumption in the target topology approaches 1 Gb/s. Depending on communication patterns and topology mapping, we are able to scale available bisection bandwidth linearly with additional core nodes in our switched gigabit network hosting environment.

- We demonstrate the utility of a number of techniques that allow users to trade increased emulation scalability for reduced accuracy. This is required because, in general, it is impossible to capture the full complexity of the Internet in any controlled environment. Sample approaches include: i) progressively reducing the complexity of the emulated topology, ii) multiplexing multiple virtual edge nodes onto a single physical machine, and iii) introducing synthetic background traffic to dynamically change network characteristics and to inject faults.
- We illustrate the generality of our approach through evaluation of a broad range of distributed services, including peer-to-peer systems, ad hoc wireless networking, replicated web services, and self-organizing overlay networks. For one of our experiments, we use ModelNet to independently reproduce published results of a wide-area evaluation of CFS [6].

Our intent is for the research community to adopt large-scale network emulation as a basic methodology for research in experimental Internet systems.

The rest of this paper is organized as follows. The next section describes the ModelNet architecture. Section 3 discusses our implementation and an evaluation of the system's baseline accuracy and scalability. Section 4 then discusses techniques to support accuracy versus scalability tradeoffs in large-scale system evaluation. In Section 5, we demonstrate the generality of our approach by using ModelNet to evaluate a broad range of networked services. Section 6 compares our work to related efforts and Section 7 presents our conclusions.

2 ModelNet Architecture

Figure 1 illustrates the physical network architecture supporting ModelNet. Users execute a configurable number of instances of target applications on *Edge Nodes* within a dedicated server cluster. Each instance is a *virtual edge node* (VN) with a unique IP address and corresponding location in the emulated topology. Edge nodes can run any OS and IP network stack and may execute unmodified application binaries. To the edge nodes, an accurate emulation is indistinguishable from direct execution on the target network. ModelNet emulation runs in *real time*, so packets traverse the emulated network with the same rates, delays, and losses as the real network. We

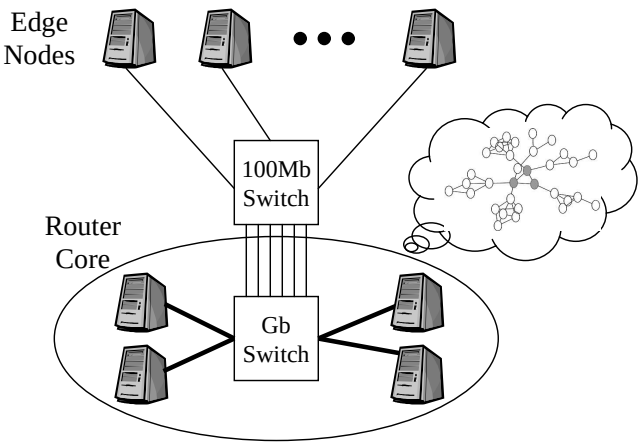

Figure 1: ModelNet.

use standard administrative tools to configure edge nodes to route their network traffic through a separate set of servers acting as *Core Routers*. The core nodes are equipped with large memories and modified FreeBSD kernels that enable them to emulate the behavior of a configured *target network* under the offered traffic load.

A key difference between ModelNet and earlier efforts is that it targets the emulation of entire large-scale topologies. Thus, ModelNet captures the effects of congestion and cross traffic on end-to-end application behavior. To achieve this, the core routes traffic through a network of emulated links or *pipes*, each with an associated packet queue and queueing discipline. Packets move through the pipes and queues by reference; a core node never copies packet data. Packets enter the queues as they arrive from VNs or exit upstream pipes, and drain through the pipes according to the pipe's specified bandwidth, latency, and loss rates. Each queue buffers a specified maximum number of packets; overflows result in packet drops. When a packet exits the link network, the core transmits the packet to the edge node hosting the destination VN.

2.1 ModelNet Phases

ModelNet runs in five phases as shown in Figure 2. The first phase, *Create*, generates a network topology, a graph whose edges represent network links and whose nodes represent clients, stubs, or transits (borrowing terminology from [3]). Sources of target topologies include Internet traces (e.g., from Caida), BGP dumps, and synthetic topology gener-

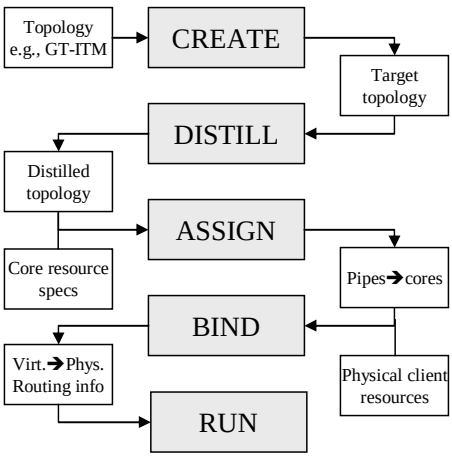

Figure 2: The phases of the ModelNet architecture.

ators [3, 4, 12]. ModelNet includes filters to convert all of these formats to GML (graph modeling language). Users may annotate the GML graph with attributes not provided by its source, such as packet loss rates, failure distributions, etc.

The *Distillation* phase transforms the GML graph to a pipe topology that models the target network. As an option, distillation may simplify the network, trading accuracy for reduced emulation cost. Section 4.1 discusses distillation and other techniques to balance accuracy and scalability in ModelNet.

The *Assignment* phase maps pieces of the distilled topology to ModelNet core nodes, partitioning the pipe graph to distribute emulation load across the core nodes. Note that the ideal assignment of pipes to cores depends upon routing, link properties, and traffic load offered by the application, an NP-complete problem. Currently, we use a simple greedy k-clusters assignment: for k nodes in the core set, randomly select k nodes in the distilled topology and greedily select links from the current connected component in a round-robin fashion. We are investigating approximations for dynamically reassigning pipes to cores to minimize bandwidth demands across the core based on evolving communication patterns.

The *Binding* phase assigns VNs to edge nodes and configures them to execute applications. ModelNet multiplexes multiple VNs onto each physical edge node, then binds each physical node to a single core. The binding phase automatically generates a set of configuration scripts for each node hosting the emulation. For core routers, the scripts install sets of pipes in the distilled topology and routing tables with shortest-path routes between all pairs of VNs. The scripts further configure edge nodes with appropriate IP addresses for each VN.

The final *Run* phase executes target application code on the edge nodes. We have developed simple scripts to automate this process, so that a single command is sufficient to execute thousands of instances of a particular application across a cluster. A key detail is that VNs running application instances must `bind` IP endpoints to their assigned IP addresses in the emulated network rather than the default IP address of the host edge node. This is necessary to mark outgoing packets with the correct source and destination addresses, so they are routed through the core and to their final destination correctly. ModelNet includes a dynamic library to interpose wrappers around the socket-related calls that require this binding step. Most modern operating systems provide a mechanism to preload libraries that intercept system calls while leaving the default variants accessible to the wrapper. In Linux, the affected system calls include `bind`, `connect`, `sendto`, `recvfrom`, `sendmsg`, `recvmsg`, and name resolution calls (e.g., `gethostbyname`, `gethostbyname2`, `gethostname`, `uname` and `gethostbyaddr`).

2.2 Inside the Core

During the *Binding* phase, ModelNet pre-computes shortest-path routes among all pairs of VNs in the distilled topology, and installs them in a routing matrix in each core node. Each route consists of an ordered list of pipes that need to be traversed to deliver a packet from source to destination. This straightforward design allows fast indexing and scales to 10,000 VNs, but the routing tables consume $O(n^2)$ space.

This scheme can be extended to scale to larger target networks. For common Internet-like topologies that cluster VNs on stub domains, we could spread lookups among hierarchical but smaller tables, trading less storage for a slight increase in lookup cost. Another alternative is to use a hash-based cache of routes for active flows (of size $O(n \lg n)$). In the rare case of a cache miss, the route may be fetched from an external cache or computed on the fly from an internal representation of the topology using Dijkstra's shortest path algorithm (an $O(n \lg n)$ operation).

Figure 3 illustrates the packet processing steps in ModelNet cores. First an IP firewall (ipfw) rule intercepts packets entering the emulated network based

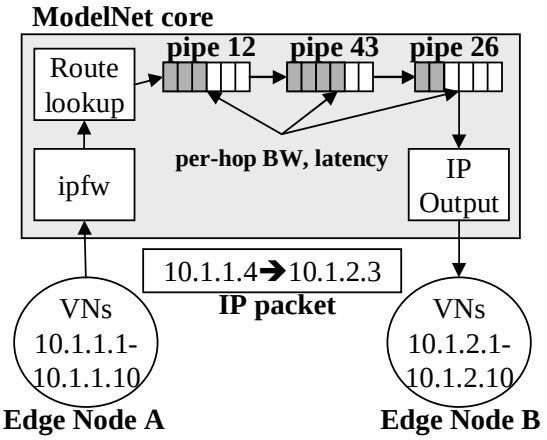

Figure 3: A packet traveling from one edge node to another through ModelNet.

on IP address: all VNs bind to IP addresses of the form *10.0.0.0/8*. On a match, control is transferred to the ModelNet kernel module, which first looks up the route for the given source and destination. This returns the set of pipes that are traversed in the emulated topology. ModelNet creates a descriptor referencing the buffered packet and schedules this descriptor on the appropriate pipes. Because these pipes are shared among all simultaneous flows and each has a fixed (specifiable) amount of queueing, ModelNet is able to emulate the effects of congestion and packet drops according to application-specific communication patterns.

Packet scheduling in ModelNet uses a heap of pipes sorted by earliest deadline, where each pipe's deadline is defined to be the exit time for the first packet in its queue. The ModelNet scheduler executes once every clock tick (currently configured at 10Khz) and runs at the kernel's highest priority level. The scheduler traverses the heap for all pipe deadlines that are later than the current time. The descriptor for the packets at the head of the queue for each of these pipes is removed and either: i) moved to the tail of the queue for the next pipe on the packet's path from source to destination, or ii) the packet itself is scheduled for delivery (using the kernels `ip_output` routine) in the case where the packet has reached its destination. The scheduler then calculates the new deadline for all pipes that had packets dequeued during the current clock tick and reinserts these pipes into the pipe heap according to this new deadline.

Achieving emulation accuracy requires careful coordination of kernel components. In steady state, a ModelNet core performs two principal tasks. First, it processes hardware interrupts to retrieve packets from the network interface. Second, the ModelNet scheduler wakes up periodically to move packets from pipe to pipe or from pipe to final destination. The second operation operates at a strictly higher kernel priority than the first. Thus, under load, ModelNet will preferentially emulate the delay for packets that have already entered the core rather than service packets waiting to enter the core through hardware interrupts. This means that core CPU saturation results in dropped packets (at the physical level) rather than inaccurate emulation. Hence, the relative accuracy of a ModelNet run is proportional to the number of physical packets dropped (note the distinction between physical drops and emulated "virtual" drops in the core).

We base our core implementation on dummynet [18], with extensions to improve accuracy and to support multi-hop and multi-core emulation. When a packet arrives at a pipe queue, the emulation computes the time to dequeue the packet and enter the pipe itself—if it is not dropped due to a congestion-related queue overflow, randomized loss, or a RED policy (each pipe is FIFO by default). We calculate this time based on the size of the packet, the size of all earlier packets queued waiting to enter the pipe, and the bandwidth of the pipe. As described above, the clock interrupt handler checks for dequeued packets once every system tick. A packet enters a pipe by transferring to the pipe's *delay line* queue, where it waits until it exits the pipe according to the pipe's specified latency; the link's delay-line queue holds its bandwidth-delay product if the link is fully utilized.

This process repeats for every pipe in the path from source to destination. When the packet exits its last pipe, `ip_output` forwards it to its destination edge node, where the host operating system delivers it to the VN process bound to the destination IP address.

For multi-core configurations, the next pipe in a route may be owned by a different core node. In this case, the core node tunnels the packet descriptor to the owning node, which is determined by a table lookup in a pipe ownership directory (POD) created during the Binding phase. Note that link emulation does not require access to the packet contents itself, so ModelNet can reduce the physical bandwidth requirements in the core by leaving the packet contents buffered on the entry core node and forwarding it directly to the destination edge node when it exits the emulated network [22].

2.3 Discussion

We now briefly discuss a number of outstanding issues with the current ModelNet architecture. Since we do not perform resource isolation among competing VN's running on the edges, it is possible for a single VN to transmit UDP packets as fast as allowed by the hardware configuration (100 Mbps in our setup). ModelNet will drop the appropriate portion of these packets according to the specified first-hop characteristics of emulated pipes in the topology. However, since UDP flows do not respond to congestion signals (dropped packets), they will continue sending at the same rate, preventing other well-behaving TCP flows originating from the same physical host to obtain their proper share of emulated bandwidth. A number of solutions exist to this problem, such as running individual VN's within a virtual machine or running traffic shapers (ModelNet itself could be applied recursively in this manner) on the edge hosts. For portability, simplicity, and our desire to focus on congestion-friendly distributed services, we choose to minimize required modifications to edge nodes.

There are a number of scalability issues with ModelNet. First, the traffic traversing the ModelNet core is limited by the cluster's physical internal bandwidth. For current commodity network switches, this means that application bisection bandwidth is limited to roughly 10 Gb/s—assuming 10 ModelNet cores on a switched gigabit network, and that individual pipes in the emulated topology are limited to the bandwidth of a single host, or 1 Gb/s. We believe this level of bandwidth to be sufficient for a wide variety of interesting distributed services. Next, Modelnet must buffer up to the full bandwidth-delay product of the target network. Fortunately, this memory requirement is manageable for our target environment. For example, a core node requires only 250 MB of packet buffer space to carry flows at an aggregate bandwidth of 10 Gb/s (currently beyond the capacity of a single core node) with 200 ms average round-trip latency.

Finally, we currently assume the presence of a "perfect" routing protocol that calculates the shortest path between all pairs of hosts. Upon an individual node or link failure, we similarly assume that the routing protocol is able to instantaneously discover the resulting shortest paths. We are currently in the process of implementing various routing protocols within the ModelNet core. The idea here is to emulate the propagation and processing of routing protocol packets within a ModelNet routing module without involving edge nodes. By leveraging our existing packet emulation environment, we are able to capture the latency and communication overhead associated with routing protocol code while leaving the edge hosts unmodified.

3 Implementation and Evaluation

We have completed an implementation of the entire ModelNet architecture described above. Here, we describe the novel aspects of the resulting implementation, focusing on ModelNet's performance and scalability. Unless otherwise noted, all experiments in this paper run on the following configuration, as depicted at a high level in Figure 1. The ModelNet core routers are 1.4Ghz Pentium IIIs with 1GB of main memory running FreeBSD-4.5-STABLE with a separate kernel module running our extensions. Each core router possesses a 3Com 3c966-T NIC based on the Broadcom 5700 chipset and is connected to a 12-port 3Com 4900 gigabit switch. Edge nodes are 1Ghz Pentium IIIs with 256MB of main memory running Linux version 2.4.17 (though we have also conducted experiments with Solaris 2.8 and FreeBSD-4.x). Unless otherwise noted, these machines are connected via 100Mb/s Ethernet interface to multiple 24-way 3Com 100 Mb/s switches with copper gigabit aggregating uplinks to the 3Com 4900 switch connecting the ModelNet core.

3.1 Baseline Accuracy

The ability to accurately emulate target packet characteristics on a hop-by-hop basis is critical to ModelNet's success. Thus, a major component of our undertaking was to ensure that, under load, packets are subject to appropriate end-to-end delays, as determined by network topology and prevailing communication patterns. We developed a kernel logging package to track the performance and accuracy of ModelNet. The advantage of this approach is that information can be efficiently buffered and stored offline for later analysis. Using such profiling, we were able to determine the expected and actual delay on a per-packet basis. We found that by running the ModelNet scheduler at the kernel's highest priority level, each packet-hop is accurately emulated to within the granularity of the hardware timer (currently 100 μs) up to and including 100% CPU utilization. Recall that packets are physical dropped on the core's network interface under overload. If a packet takes 10 hops from source to destination, this

corresponds to a worst-case error of 1 ms, which we consider acceptable for our target wide-area experiments. We are in the process of implementing packet debt handling within the ModelNet scheduler, where the scheduler maintains the total emulation error and attempts to correct for it in subsequent hops. With this optimization, we believe that per-packet emulation accuracy can be reduced to 100 μs in all cases.

ModelNet delivers packets to their destination within 1 ms of the target end-to-end value that accounts for propagation delay, queueing delay, and hop-by-hop link bandwidth. The system maintains this accuracy up to and including 100% CPU utilization. Given that the principal focus of our work is large-scale wide-area network services, we believe that this level of accuracy is satisfactory.

3.2 ModelNet Capacity

Our first experiment quantifies the capacity of ModelNet as a function of offered load, measured in packets per second (like most "routers", ModelNet overhead is largely independent of packet size save the relatively modest `memcpy` overhead) and as a function of emulated hops that the packet must traverse. We vary from 1-5 the number of edge nodes with 1 Gb/s Ethernet connections communicating through a single core with a 1 Gb/s link. Each edge node hosts up to 24 `netperf` senders (24 VNs) communicating over TCP. We spread the receivers across 5 edge nodes, each also hosting up to 24 `netperf` receivers. The topology directly connects each sender with a receiver over a configurable number of 10 Mb/s pipes with an end-to-end latency of 10 ms. By changing the number of pipes in the path, we vary the total amount of work ModelNet must perform to emulate end-to-end flow characteristics.

Figure 4 plots the capacity in packets/sec of the single ModelNet core as a function of the number of simultaneous flows competing for up to 10 Mb/s of bandwidth each. There are five curves, each corresponding to a different number of pipes that each packet must traverse from source to destination. For the baseline case of 1 hop, Figure 4 shows that the performance of the core scales linearly with offered load up to 96 simultaneous flows (4 edge nodes). At saturation with 1 hop, ModelNet accurately emulates approximately 120,000 packets/sec. At this point, the ModelNet CPU is only 50% utilized, with the bottleneck being in the network link. 120,000 packets/sec corresponds to an aggregate of 1 Gb/s (average packet size is 1 KB when accounting for 1 ACK for every two 1500-byte data packets).

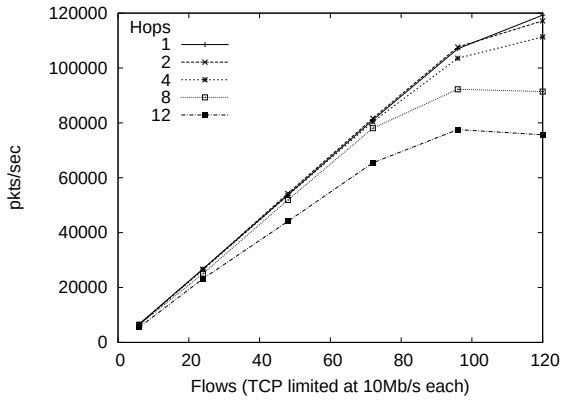

Figure 4: Capacity of a single ModelNet core.

Only once we attempt to emulate more than 4 hops per flow does the ModelNet CPU become the bottleneck. For 8 hops per flow, ModelNet accurately forward approximately 90,000 packets/sec. At this point, the machine enters a state where it consumes all available resources to i) pull 90,000 packets/sec from the NIC, ii) emulate the 8-hop characteristics of each packet, and finally iii) forward the packets to their destination. The NIC drops additional packets beyond this point (recall that ModelNet emulation runs at higher priority than NIC interrupt handling). These drops throttle the sending rate of the TCP flows to the maximum 90,000 packets/sec. Overall, we measure a 0.5 μs overhead for each hop in the emulated topology and a fixed per-packet overhead of 8.3 μs for each packet traversing the core (resulting from the standard IP protocol stack). Our hardware configuration can forward approximately 250,000 small packets/sec when not performing ModelNet emulation, indicating significant opportunity for improved system capacity with improving CPU performance.

3.3 Scaling with Additional Cores

The next experiment measures ModelNet's ability to deliver higher throughput with additional communicating core routers. Recall that ModelNet's ability to deliver additional bandwidth with additional cores depends upon the target topology and application communication patterns. As the topology is divided across an increasing number of cores, there is an increasing probability that a packet's path forces it to cross from one core to another before being deliv-

Cross-Core Traffic	Throughput (Kpkt/sec)
0%	462.5
25%	404.5
50%	276.3
75%	219.3
100%	155.8

Table 1: Scalability as a function of communication pattern for 4-core experiment.

ered to its final destination. Each such core crossing negatively impacts overall system scalability.

To quantify this effect, we configure 20 machines as edge nodes, each hosting 56 VNs, evenly split among 28 senders and 28 receivers (for a total of 1120 VNs). For this experiment, the edge machines were each configured with a gigabit link to the ModelNet core to allow us to load the 4-node configuration with fewer resources. We emulate a star topology with all VNs connected to a central point with 10 Mb/s, 5 ms latency pipes. Thus, all paths through the topology consist of 2 hops. We assign one quarter of the VNs to each core. During the experiment all 560 senders use `netperf` to simultaneously transmit TCP streams to 560 unique receivers; there is no contention for the last-hop pipes. We vary the communication pattern so that during any one experiment, x% of flows must cross from one core to the other, leaving $(1-x)$% of the traffic to travel within a single core.

Table 1 summarizes the results of this experiment, showing the maximum system throughput in packets/sec as a function of the percent of cross-core traffic. With 0% cross-core traffic, the system delivers four times the throughput of a single core (compare to the 2-hop results from Figure 4 above) degrading roughly linearly with additional cross-core traffic. For more complicated topologies where, for example, each packet must traverse four different cores, throughput would degrade further. Thus, the ability to scale with additional core nodes depends on application communication characteristics and properly partitioning the topology to minimize the number of inter-core packet crossings.

4 Accuracy versus Scalability Tradeoffs

This section outlines several techniques to configure emulations on a continuum balancing accuracy and cost. It is impractical to model every packet and link for a large portion of the Internet on a PC cluster. Rather, the goal of ModelNet is to create a controlled Internet-like execution context with a diversity of link capabilities, rich internal topology, network locality, dynamically varying link status, cross traffic, congestion, and packet loss. The key to emulation at scale is to identify and exploit opportunities to reduce overhead by making the emulation less faithful to the target network in ways that minimally impact behavior for the applications under test. Ideally, the emulation environment would automate these tradeoffs to yield the most accurate possible emulation on the available hardware, then report the nature and degree of the inaccuracy back to the researcher. We have taken some initial steps toward this vision in our work with ModelNet.

4.1 Distillation

ModelNet's *Distillation* phase modifies the topology with the goal of reducing the diameter of the emulated network. In a pure *hop-by-hop* emulation, the distilled topology is isomorphic to the target network; ModelNet faithfully emulates every link in the target network. This approach captures all congestion and contention effects in the topology but also incurs the highest per-packet overhead. At the other extreme, *end-to-end*, distillation removes all interior nodes in the network, leaving a full mesh with $O(n^2)$ links interconnecting the n VNs. The distiller generates the mesh by collapsing each path into a single pipe p: the bandwidth of p is the minimum bandwidth of any link in the corresponding path, its latency is the sum of the link latencies along the path, and its reliability $(1 - lossrate)$ is the product of the link reliabilities along the path. This approach yields the lowest per-packet emulation overhead—since each packet traverses a single hop—and it accurately reflects the raw network latency, bandwidth, and base loss rate between each pair of VNs. However, it does not emulate link contention among competing flows except between each pair of VNs.

ModelNet provides a *walk-in* "knob" to select an arbitrary balance along the continuum between these extremes by "preserving" the first *walk-in* links from the edges. A breadth-first traversal generates successive *frontier sets*. The first frontier set is the set of all VNs; the members of the $i+1$ frontier set are the nodes that are one hop from a member of the ith frontier set, but are not members of any preceding frontier set. The *interior* consists of all nodes that are not members of the first *walk-in* frontier sets.

The distiller replaces the interior links of the target topology with a full mesh interconnecting the interior nodes, as in the end-to-end approach. This approach reduces emulation cost relative to hop-by-hop because each packet traverses at most *(2*walk-in)+1* pipes, rather than the network diameter. A *last-mile* emulation (*walk-in = 1*) preserves the first and last hop of each path.

Distilled *walk-in* emulations do not model contention in the interior, but this is a useful tradeoff assuming that today's Internet core is well-provisioned and that bandwidth is constrained primarily near the edges of the network at peering points and over the last hop/mile. To model under-provisioned cores, ModelNet extends the *walk-in* algorithm above to preserve the inner core of breadth *walk-out*. This extends the *walk-in* algorithm above to generate successive frontier sets until a frontier of size one or zero is found, representing the topological "center" of the target topology. Suppose that this is frontier set *c*. The interior corresponding to breadth *walk-out* is the union of frontier sets *c - walk-out* through *c*. Distillation preserves links interconnecting the interior, while collapsing paths between the walk-in and walk-out frontiers as described above.

To illustrate distillation, we ran a simple experiment with a ring topology. The ring has 20 routers interconnected at 20 Mb/s; each ring router has 20 VNs directly connected to it by individual 2 Mb/s links. The VNs are evenly partitioned into *generator* and *receiver* sets of size 200. Each generator transmits a TCP stream to a random receiver. This topology then has 419 pipes shared among the 400 VNs. The end-to-end distillation contains 79,800 pipes, one for each VN pair, each with a bandwidth of 2 Mb/s. The last-mile distillation preserves the 400 edge links to the VNs, and maps the ring itself to a fully connected mesh of 190 links. The last-hop configuration can handle significantly more packets on a given emulation platform, since each packet traverses 3 hops rather than the average case of 7 in the original ring topology.

Figure 5 plots a CDF for the distribution of bandwidth achieved by the 200 flows in the various emulations, and in full hop-by-hop simulations using the ns2 simulator with both 20 Mb/s and 80 Mb/s rings. The hop-by-hop emulation shows a roughly even distribution of flow bandwidths, matching the 20 Mb/s ns2 results. Here, each 20 Mb/s transit link has an offered load of 27.5 Mb/s, constraining the bandwidth of flows passing through the ring. In

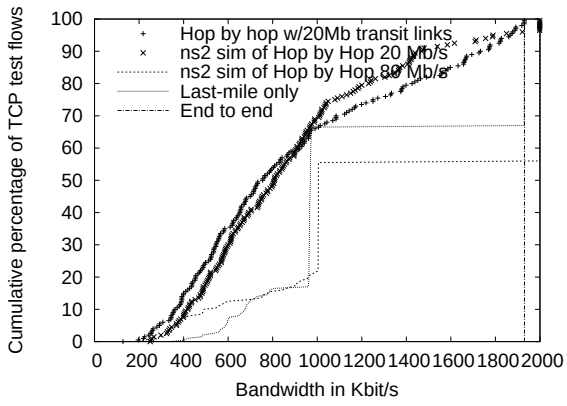

Figure 5: The effects of distillation on distribution of bandwidth in a ring topology.

the end-to-end emulation, which does not model contention in the ring, all flows achieve their full 2 Mb/s of bandwidth. The last-mile emulation models congestion only for the 64% of flows that share the same receiver; the bandwidths for these flows are 1 Mb/s or less, depending on the number of flows sharing a given receiver. The other 36% of flows all achieve 2Mb/s. The last-mile emulation is qualitatively similar to the ns2 results with an 80 Mb/s ring, since in this case the ring is adequately provisioned and the only contention is on the last hop to the receiver. The emulation results closely match the exhaustively verified ns2 simulation environment, improving our confidence in the accuracy of our implementation.

4.2 VN Multiplexing

The mapping of VNs to physical edge nodes also affects emulation accuracy and scalability. Higher degrees of multiplexing enable larger-scale emulations, but may introduce inaccuracies due to context switch overhead, scheduling behavior, and resource contention on the edge nodes. The maximum degree of safe multiplexing is application-dependent. For many interesting application scenarios (e.g., Web load generators), the end node CPU is not fully utilized and higher multiplexing degrees are allowable. The multiplexing degree is lower for resource-intensive VNs, such as Web servers.

The accuracy cost of VN multiplexing also varies with the concurrency model. In general, each VN runs as one or more processes over the edge node host OS; all sockets open in those processes bind to the VN's assigned IP address. Mapping the sockets of a single process onto multiple VNs may allow

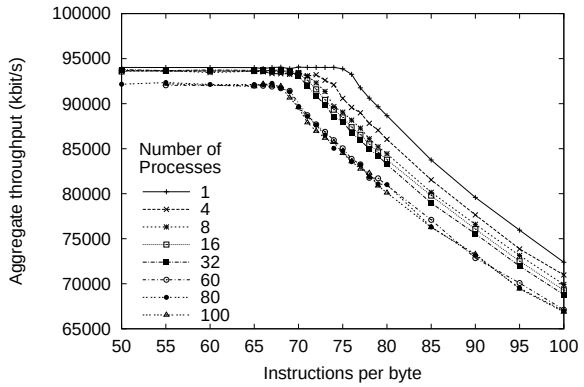

Figure 6: Effects of multiplexing processes on an edge node.

higher VN multiplexing degrees with less likelihood of saturating the edge node if the process manages its concurrency efficiently using threads or an event-driven model. ModelNet users can achieve this per-socket VN mapping in a straightforward manner using a variant of the socket interposition library that maps each open socket to a different VN.

To illustrate the accuracy tradeoff of VN multiplexing, we ran an experiment to show its effect on VNs running modified netperf/netserver processes that exchange 1500-byte UDP packets with a configurable amount of computation per packet. The experiment uses three physical nodes: one for netperf sources, one for netserver sinks, and one for the ModelNet core. We ran experiments with varying multiplexing degrees, varying the emulated topology so that each of *nprog* netperf/netserver pairs is configured with $1/nprog$ of the physical 100 Mb/s link.

Figure 6 plots aggregate bandwidth across all processes as a function of the number of instructions *delay* (per byte) after each packet transmission, for a multiplexing degree *nprog* ranging from 1 to 100. The actual delay in instructions between packet transmissions on the x-axis should be multiplied by the packet size (1500 bytes). We expect context switch overhead to consume an increasing fraction of system resources as we increase *nprog*. With zero *delay*, the processes deliver an aggregate of approximately 95 Mb/s. With $nprog = 1$, there is no loss of throughput up to a delay of 76 instructions/byte. The theoretical maximum is 1Ghz*8/100Mb/s = 80 instructions per byte (assuming a CPI of 1.0). With $nprog = 2$, the allowable total computation per-byte degrades to 73 instructions due to context switching overheads; with $nprog = 100$ it falls to 65 instructions per transmitted byte.

We plan to use such a benchmarking tool to automate the process of determining the computation versus communication ratio of individual applications (with the help of the Pentium's cycle and instruction counters). Such an evaluation will assist ModelNet users to determine the accuracy tradeoff at various multiplexing degrees.

4.3 Changing Network Characteristics

An important goal of ModelNet is to enable researchers to evaluate adaptive Internet systems by subjecting them to competing traffic and observing their responses to the resulting variations in network performance. ModelNet users may do this directly by incorporating generators for competing traffic with specified properties (TCP, constant bit rate, etc.) into the VN application mix. While this is the most accurate way to emulate "background" cross traffic, it consumes resources at the edge nodes as well as ModelNet emulation bandwidth at the core. Another option is to modify the core kernels to insert dummy packets into the pipes according to a specified pattern. However, this approach also consumes significant core resources to generate the cross traffic and propagate it through the pipe network.

ModelNet allows users to inject cross traffic by modifying pipe parameters dynamically as the emulation progresses. The cross traffic at each point in time is specified as a matrix indicating communication bandwidth demand between each VN pair (i, j). These matrices may be generated from a synthetic background traffic pattern or probability distribution, or fetched from a stored set of "snapshot" profiles. An offline tool propagates the matrix values through the routing tables to determine the impact of the specified cross-traffic signals on each pipe in the distilled topology. During the emulation, a configuration script periodically installs derived pipe parameter settings into the ModelNet core nodes. The new settings represent cross traffic effects by increasing pipe latency to capture queueing delays, reducing pipe bandwidth to capture the higher link utilization, and reducing the queue size bound to model the impact on the steady-state queue length. Thus, a flow competing with synthetic cross traffic sees a reduced ability to handle packet bursts without loss, as well as increased latency and lower available bandwidth. We derive the new settings from a simple analytical queueing model for each impacted link.

This approach incurs low overhead and scales independently of both the cross traffic rate and the actual traffic rate. The drawback is that it does not capture all the details of Internet packet dynamics (slow start, bursty traffic, etc.). In particular, synthetic cross traffic flows are not responsive to congestion in our current approach; thus they introduce an emulation error that grows with the link utilization level.

ModelNet also adjusts pipe parameter settings to emulate other dynamic network changes, e.g., fault injection. For example, the user can direct ModelNet to change the bandwidth, delay, and loss rate of a set of links according to a specified probability distribution every x seconds. For node or link failures, the configuration script also updates the system routing tables. Currently, this is done by recalculating all-pairs shortest paths; we are currently investigating techniques for emulating realistic routing protocols within ModelNet. In this way, users can observe how their system responds to pre-specified stimuli, for example, network partitions, sudden improvements in available bandwidth, sudden increase in loss rate across a backbone link, etc. In particular, random stress tests are useful because it is often just as important to identify conditions under which services will fail than it is to demonstrate how well they behave in the common case.

5 Case Studies

In this section, we demonstrate the utility and generality of our approach by evaluating four sample distributed services on ModelNet. Beyond the tests below, the largest single experiment completed in our environment successfully evaluated system evolution and connectivity of a 10,000 node network of unmodified gnutella clients by mapping 100 VNs to each of 100 edge nodes in our cluster. To further explore the generality of our approach, we have also implemented extensions to our architecture to support emulation of ad hoc wireless environments. These changes involve supporting the broadcast nature of wireless communication (packet transmission consumes bandwidth at all nodes within communication range of the sender) and node mobility (topology change is the rule rather rather than the exception). While complete, we omit a detailed evaluation of this last case study for brevity.

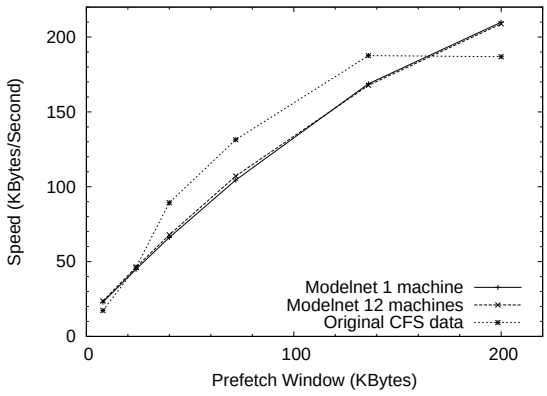

Figure 7: Download speed as a function of prefetch window for CFS/RON and CFS/ModelNet.

5.1 CFS

As a first demonstration of the utility of ModelNet, we set out to reproduce the published experimental results of another group's research in large-scale distributed computing. We chose the CFS implementation [6] running on top of the RON testbed [1]. CFS is an archival storage system layered on top of Chord [20], a fully distributed hash table. A CFS prototype was evaluated on 12 nodes spread across the Internet. We chose CFS running on RON because of their exemplary experimental practice: The CFS code is publicly available and the network characteristics (bandwidth, latency, loss rate) among all pairs of RON nodes are published.

We converted the publicly available RON interconnectivity characteristics into a ModelNet topology. The only limitation we encountered was that the authors did not record exactly which 12 of the 15 RON nodes hosted the reported experimental results. We tried a number of permutations of potential participants, but were not exhaustive because, as shown below, we were able to closely reproduce the CFS results in all cases.

Specifically, we reproduce the experimental results from Figures 6, 7, and 8 of the CFS paper [6]. CFS Figures 6 and 7 are two views of how varying the Chord prefetch size changes the download bandwidth for retrieving 1MB of data striped across all participating CFS nodes. CFS Figure 8 shows the bandwidth distribution for a series of pure TCP transfers between the RON nodes and is presented as a comparison between the transfer speeds delivered by CFS relative to TCP. We were able to extract the

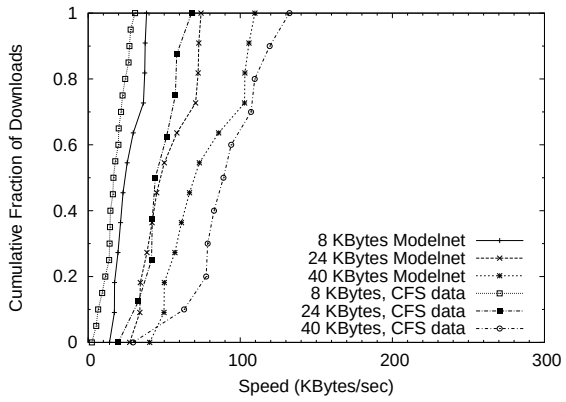

Figure 8: CDF of download speed for various prefetch windows for CFS/RON and CFS/ModelNet.

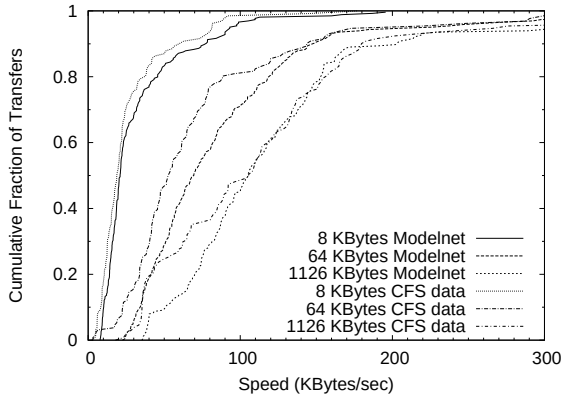

Figure 9: TCP transfer of 8, 64, and 1164KB files using TCP running on both the RON testbed and ModelNet.

raw data for these experiments from information contained within the encapsulated postscript of the figures in the publicly available postscript version of the CFS paper. We plot these curves along with our own experimental results of the CFS code running on ModelNet in Figures 7, 8, and 9.

We replicate the Chord prefetch experiment (Figures 7 and 8) in two ways. The "Modelnet 12 node" curve shows the results of running the experiment on 12 individual edge nodes, each with a single VN representing a RON node and running CFS code. Next, to further demonstrate ModelNet's ability to accurately multiplex multiple logical nodes onto a single physical node, we run 12 instances (VNs) of CFS on a single machine. The "ModelNet 1 node" curve in Figure 7 shows the results of this experiment. Figure 8 depicts the Chord prefetch results in more detail.

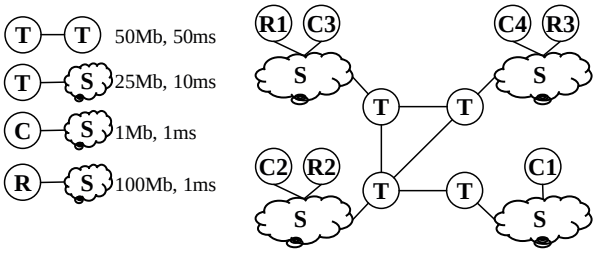

Figure 10: Sample network topology for investigating the effects of replication on client-perceived latency.

A final experiment in CFS measures basic TCP bandwidth between all RON nodes by transferring files of 8, 64 and 1164 KB. Figure 9 shows the results for each transfer size sorted by achieved throughput for both our ModelNet experiments and the CFS wide-area experiments. We used 12 physical edge nodes for this experiment.

Overall, the results of CFS/ModelNet experiments closely match the results for CFS/RON in all cases. Several potential sources of error may explain the discrepancies: i) the CFS experiments were run at a different time than the RON network characteristics were measured, ii) resource contention in the "middle" of the network were not captured by the end-to-end network characterization available to us, and iii) different hardware and operating systems were used to carry out the wide-area versus ModelNet experiments. Overall, however, we are quite satisfied with how closely we were able to reproduce these results. Running experiments on ModelNet is also a lightweight operation relative to coordinating experiments across multiple wide-area nodes: One experienced programmer was able to reproduce all the CFS experiments on ModelNet in one day.

5.2 Replicated Web Services

Recently, there has been increasing interest in wide-area replication of Internet services to improve overall performance and reliability. A principal motivation for our work is to develop an environment to support the study of replica placement and request routing policies under realistic wide-area conditions. While such a comprehensive study is beyond the scope of this paper, we present the results of some initial tests to demonstrate the suitability of ModelNet for evaluating replicated network services.

The goal of our experiment is to demonstrate that ModelNet support for realistic Internet topologies

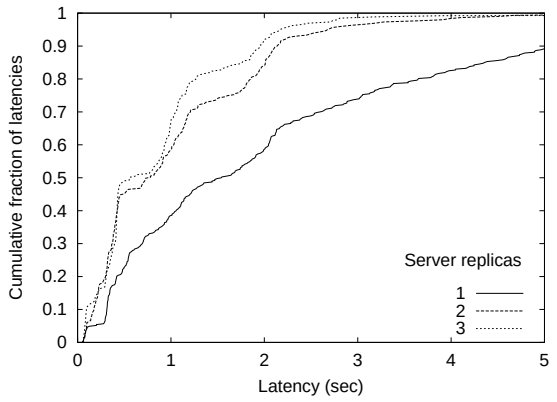

Figure 11: CDF of client-perceived latency as a function of number of replicas.

and emulation of contention for shared pipes supports complex experiments involving replica placement and request routing policies. We create a modified 320-node transit-stub [3] topology, as depicted in Figure 10. Clouds labeled "S" in the figure represent stub domains with more complex internal connectivity. These stub-stub links are set with 10 Mb/s bandwidth, and 5 ms latency. Additional link characteristics are set as described in the figure. For our experiments, we use locally developed software to play back in real time 2.5 minutes of a larger trace to IBM's main web site (`www.ibm.com`) from February 2001 [5]. Load varies from 60-100 requests/sec during this period. We use eight machines for this experiment: a single ModelNet core, 4 machines (hosting clients labeled $C1-C4$ in Figure 10) simultaneously play back the trace, and up to 3 machines (labeled $R1-R3$ in Figure 10) host Apache (version 1.3.24-3) servers. We map 30 VNs in the same stub domain to each of $C1-C4$. Each VN has its own dedicated 1 Mb/s, 1 ms link to the rest of the stub domain. We run three separate experiments. In the first, all clients $C1-C4$ make requests to $R1$. Server CPU utilization was 10% even when all requests are directed to this single server, indicating that the network rather than host CPU was the bottleneck in our experiments. In the second experiment, we manually configure clients at $C1$ and $C2$ to direct their requests to $R2$ (the remaining clients still go to $R1$). In the final experiment, we configure $C4$ to direct its requests to $R3$.

Figure 11 plots the cumulative distribution of all client latencies for the three different experiments described above. We see that with 1 server, client latencies are relatively large, with 10% taking longer than 5 seconds to complete. Instrumentation shows that this results from contention on the transit-to-transit links in the topology. An additional replica greatly improves client latency by largely eliminating such contention. A third replica provides only marginal benefit in this case. Note that these results would not be possible without ModelNet's ability to accurately emulate contention on interior links in the topology. Further, these simple results depict an upper bound on the benefits of wide-area replication because we assume a perfect request routing mechanism and static network conditions. Of course, a more comprehensive experiment must support dynamic request routing decisions (e.g., leveraging DNS in a content distribution network) and dynamically changing network conditions.

5.3 Adaptive Overlays

We use ACDC [9], an application-layer overlay system that dynamically adapts to changing network conditions, to demonstrate ModelNet's ability to subject systems to dynamically changing network conditions. ACDC attempts to build the lowest-cost overlay distribution tree that meets target levels of end-to-end delay. Cost and delay are two independent and dynamically changing metrics assigned to links in the underlying IP network. Nodes in the overlay use a distributed algorithm to locate parents that deliver better cost, better delay, or both. A key goal is scalability: no node maintains more than $O(\lg n)$ state or probes more than $O(\lg n)$ peers.

Full details of the algorithm and an ns2-based evaluation are available elsewhere [9]. We wrote ACDC to a locally developed compatibility layer that allows the same code to run both in ns2 and under live deployment. Thus, we are able to reproduce the results of an experiment running under both ns2 and ModelNet. We begin with a 600-node GT-ITM transit-stub topology and randomly choose 120 nodes to participate in the ACDC overlay. We assign transit-transit links a bandwidth of 155 Mb/s and a cost of 20-40, transit-stub links 45 Mb/s with a cost of 10-20 and stub-stub links 100 Mb/s with a cost of 1-5. GT-ITM determines propagation delay based on relative location in the randomly generated graph. We run ACDC using a single core and 10 edge nodes, each hosting 12 VNs.

Nodes initially join at a random point in the overlay and self-organize first to meet application-specified delay targets (1500 ms in this experiment) and then to reduce cost while still maintaining delay. Fig-

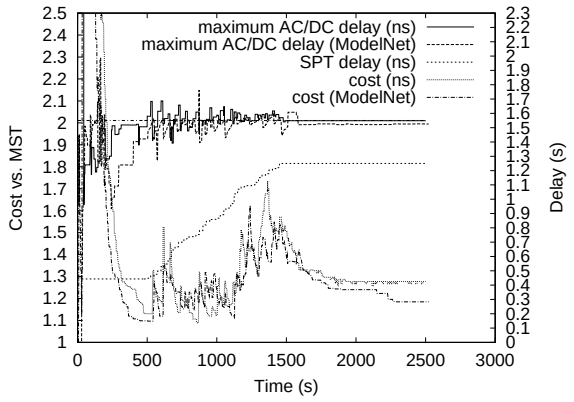

Figure 12: ACDC cost and delay running on ModelNet.

ure 12 plots as a function of time: i) the cost of the overlay relative to a minimum cost spanning tree (calculated offline) on the left y-axis, and ii) the worst-case delay of the overlay on the right y-axis. The graph also depicts the delay through the shortest path tree (again calculated offline) connecting the 120 participants. The closer the SPT delay is to the 1500ms target, the more difficult it is to achieve the performance goal. After allowing the overlay to stabilize for 500 seconds, ModelNet increases the delay on 25% randomly chosen IP links by between 0-25% of the original delay every 25 seconds. The figure shows ACDC's ability to self-organize to maintain target performance levels, sometimes sacrificing cost to do so. At $t = 1500$, network conditions subside and ACDC once again focuses on reducing cost. For comparison, Figure 12 also plots the results of an identical experiment under ns2. Overall, the results for ACDC/ns2 closely match those for ACDC/ModelNet.

6 Related Work

Many previous efforts investigate the use of emulation in support of their research [2, 11, 7, 14, 23]. Relative to ModelNet, these efforts largely focus on specific, static, and relatively small-scale systems. ModelNet, on the other hand, supports a scalable and flexible emulation environment useful for a broad range of research efforts and further performs full hop-by-hop network emulation, allowing it to capture the effects of contention and bursts in the middle of the network. Perhaps most closely related to our effort is Netbed (an outgrowth of Emulab) [21]. This tool allows users to configure a subset of network resources for isolated distributed systems and networking experiments. The testbed provides an integrated environment that allows users to set up target operating systems and network configurations. Relative to Emulab, we focus on scalable emulation of large-scale network characteristics. Overall, we plan to integrate our effort into this system in the future.

A number of collaborative efforts provide nodes across the wide-area to support "live" experimentation with new protocols and services. Examples include Access, CAIRN, PlanetLab [16], Netbed, RON [1] and the X-Bone. While these testbeds are extremely valuable, they are also typically limited to a few dozen sites and do not support controlled, large-scale, and reproducible experiments. We view ModelNet as complementary to these very necessary efforts.

One recent effort [13] uses emulation to evaluate the effects of wide-area network conditions on web server performance. They advocate emulating network characteristics at end hosts rather than in a dedicated core for improved scalability. However, this approach cannot capture the congestion of multiple flows on a single pipe, requires appropriate emulation software on all edge nodes, and must share each host CPU between the tasks of emulation and executing the target application.

7 Conclusions

Ideally, an environment for evaluating large-scale distributed services should support: i) unmodified applications, ii) reproducible results, iii) experimentation under a broad range of network topologies and dynamically changing network characteristics, and iv) large-scale experiments with thousands of participating nodes and gigabits of aggregate cross traffic. In this paper, we present the design and evaluation of ModelNet, a large-scale network emulation environment designed to meet these goals. ModelNet maps a user-specified topology to a set of core routers that accurately emulate on a per-packet basis the characteristics of that topology, including per-link bandwidth, latency, loss rates, congestion, and queueing. ModelNet then multiplexes unmodified applications across a set of edge nodes configured to route all their packets through the ModelNet core.

Of course, no cluster can capture the full scale and complexity of the Internet. Thus, a significant contribution of this work is an evaluation of a set of scalable techniques for approximating Internet conditions. In addition to a detailed architectural evalu-

ation, we demonstrate the generality of our approach by presenting our experience with running a broad range of distributed services on ModelNet, including a peer-to-peer file service, an adaptive overlay, a replicated web service, and an ad hoc wireless communication scenario.

References

[1] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, October 2001.

[2] Guarav Banga, Jeffrey Mogul, and Peter Druschel. A Scalable and Explicit Event Delivery Mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[3] Ken Calvert, Matt Doar, and Ellen W. Zegura. Modeling Internet Topology. *IEEE Communications Magazine*, June 1997.

[4] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *Proceedings of ACM SIGMETRICS*, June 2002.

[5] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, October 2001.

[6] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, October 2001.

[7] NIST Internetworking Technology Group. See `http://www.antd.nist.gov`.

[8] Yang hua Chu, Sanjay Rao, and Hui Zhang. A Case For End System Multicast. In *Proceedings of the ACM Sigmetrics 2000 International Conference on Measurement and Modeling of Computer Systems*, June 2000.

[9] Dejan Kostić, Adolfo Rodriguez, and Amin Vahdat. The Best of Both Worlds: Adaptivity in Two-Metric Overlays. Technical Report CS-2002-10, Duke University, May 2002. `http://www.cs.duke.edu/~vahdat/ps/acdc-full.pdf`.

[10] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the Use and Performance of Content Distribution Networks. In *SIGCOMM Internet Measurement Workshop*, 2001 November.

[11] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the 1997 International Symposium on Computer Architecture*, June 1997.

[12] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: An Approach to Universal Topology Generation. In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)*, August 2001.

[13] Erich M. Nahum, Marcel Rosu, Srinivasan Seshan, and Jussara Almeida. The Effects of Wide-Area Conditions on WWW Server Performance. In *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2001.

[14] Brian Noble, M. Satyananarayanan, Giao Nguyen, and Randy Katz. Trace-based Mobile Network Emulation. In *Proceedings of SIGCOMM*, September 1997.

[15] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/.

[16] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[17] Sylvia Ratnasamy, Paul Francis Mark Handley, Richard Karp, and Scott Shenker. A Content Addressable Network. In *Proceedings of SIGCOMM 2001*, August 2001.

[18] Luigi Rizzo. Dummynet and Forward Error Correction. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.

[19] Antony Rowstron and Peter Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-scale Peer-to-Peer Systems. In *Middleware'2001*, November 2001.

[20] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer to Peer Lookup Service for Internet Applications. In *Proceedings of the 2001 SIGCOMM*, August 2001.

[21] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[22] Kenneth G. Yocum and Jeffrey S. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proceedings of the USENIX Technical Conference*, June 2001.

[23] Haifeng Yu and Amin Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.