To appear in *Proc. of the Fourth Symposium on Networked Systems Design and Implementation (NSDI 2007)*, April 2007.

The Flexlab Approach to Realistic Evaluation of Networked Systems

Robert Ricci Jonathon Duerig Pramod Sanaga Daniel Gebhardt
Mike Hibler Kevin Atkinson Junxing Zhang Sneha Kasera Jay Lepreau

University of Utah, School of Computing

Abstract

Networked systems are often evaluated on overlay testbeds such as PlanetLab and emulation testbeds such as Emulab. Emulation testbeds give users great control over the host and network environments and offer easy reproducibility, but only artificial network conditions. Overlay testbeds provide real network conditions, but are not repeatable environments and provide less control over the experiment.

We describe the motivation, design, and implementation of Flexlab, a new testbed with the strengths of both overlay and emulation testbeds. It enhances an emulation testbed by providing the ability to integrate a wide variety of network models, including those obtained from an overlay network. We present three models that demonstrate its usefulness, including "application-centric Internet modeling" that we specifically developed for Flexlab. Its key idea is to run the application within the emulation testbed and use its offered load to measure the overlay network. These measurements are used to shape the emulated network. Results indicate that for evaluation of applications running over Internet paths, Flexlab with this model can yield far more realistic results than either PlanetLab without resource reservations, or Emulab without topological information.

1 Introduction

Public network testbeds have become staples of the networking and distributed systems research communities, and are widely used to evaluate prototypes of research systems in these fields. Today, these testbeds generally fall into two categories: *emulation testbeds* such as the emulation component of Emulab [37], which create artificial network conditions that match an experimenter's specification, and *overlay testbeds* such as PlanetLab [27], which send an experiment's traffic over the Internet. Each type of testbed has its own strengths and weaknesses. In this paper, we present Flexlab, which bridges the two types of testbeds, inheriting strengths from both.

Emulation testbeds such as Emulab and ModelNet [34] give users full control over the host and network environments of their experiments, enabling a wide range of experiments using different applications, network stacks, and operating systems. Experiments run on them are repeatable, to the extent that the application's behavior can be made deterministic. They are also well suited for developing and debugging applications—two activities that represent a large portion of the work in networked systems research and are especially challenging in the wide area [1, 31]. However, emulation testbeds have a serious shortcoming: their network conditions are artificial and thus do not exhibit some aspects of real production networks. Perhaps worse, researchers are *not sure* of two things: which network aspects are poorly modeled, and which of these aspects matter to their application. We believe these are two of the reasons researchers underuse emulation environments. That emulators are underused has also been observed by others [35].

Overlay testbeds, such as PlanetLab and the RON testbed [2], overcome this lack of network realism by sending experimental traffic over the real Internet. They can thus serve as a "trial by fire" for applications on today's Internet. They also have potential as a service platform for deployment to real end-users, a feature we do not attempt to replicate with Flexlab. However, these testbeds have their own drawbacks. First, they are typically overloaded, creating contention for host resources such as CPU, memory, and I/O bandwidth. This leads to a host environment that is unrepresentative of typical deployment scenarios. Second, while it may eventually be possible to isolate most of an experiment's host resources from other users of the testbed, it is impossible (by design) to isolate it from the Internet's varying conditions. This makes it fundamentally impossible to obtain repeatable results from an experiment. Finally, because hosts are shared among many users at once, users cannot perform many privileged operations including choosing the OS, controlling network stack parameters, and modifying the kernel.

Flexlab is a new testbed environment that combines the strengths of both overlay and emulation testbeds. In Flexlab, experimenters obtain networks that exhibit real Internet conditions *and* full, exclusive control over hosts. At the same time, Flexlab provides more control and repeatability than the Internet. We created this new environment by closely coupling an emulation testbed with an overlay testbed, using the overlay to provide network conditions for the emulator. Flexlab's modular framework qualitatively increases the range of network models that can be emulated. In this paper, we describe this framework and

three models derived from the overlay testbed. These models are by no means the only models that can be built in the Flexlab framework, but they represent interesting points in the design space, and demonstrate the framework's flexibility. The first two use traditional network measurements in a straightforward fashion. The third, "application-centric Internet modeling" (ACIM), is a novel contribution itself.

ACIM stems directly from our desire to combine the strengths of emulation and live-Internet experimentation. We provide machines in an emulation testbed, and "import" network conditions from an overlay testbed. Our approach is application-centric in that it confines itself to the network conditions relevant to a particular application, using a simplified model of that application's own traffic to make its measurements on the overlay testbed. By doing this in near real-time, we create the illusion that network device interfaces in the emulator are distributed across the Internet.

Flexlab is built atop the most popular and advanced testbeds of each type, PlanetLab and Emulab, and exploits a public federated network data repository, the Datapository [3]. Flexlab is driven by Emulab testbed management software [36] that we recently enhanced to extend most of Emulab's experimentation tools to PlanetLab slivers, including automatic link tracing, distributed data collection, and control. Because Flexlab allows different network models to be "plugged in" without changing the experimenter's code or scripts, this testbed also makes it easy to compare and validate different network models.

This paper extends our previous workshop paper [9], and presents the following contributions:
(1) A software framework for incorporating a variety of highly-dynamic network models into Emulab;
(2) The ACIM emulation technique that provides high-fidelity emulation of live Internet paths;
(3) Techniques that infer available bandwidth from the TCP or UDP throughput of applications that do not continually saturate the network;
(4) An experimental evaluation of Flexlab and ACIM;
(5) A flexible network measurement system for PlanetLab.
We demonstrate its use to drive emulations and construct simple models. We also present data that shows the significance on PlanetLab of non-stationary network conditions and shared bottlenecks, and of CPU scheduling delays.

Finally, Flexlab is currently deployed in Emulab in beta test, will soon be enabled for public production use, and will be part of an impending Emulab open source release.

2 Flexlab Architecture

The architecture of the Flexlab framework is shown in Figure 1. The application under test runs on emulator hosts, where the *application monitor* instruments its network operations. The application's traffic passes through the *path emulator*, which shapes it to introduce latency, limit bandwidth, and cause packet loss. The parameters for the path emulator are controlled by the *network model*, which may optionally take input from the monitor, from the *network measurement repository*, and from other sources. Flexlab's framework provides the ability to incorporate new network models, including highly dynamic ones, into Emulab. All parts of Flexlab except for the underlying emulation testbed are user-replaceable.

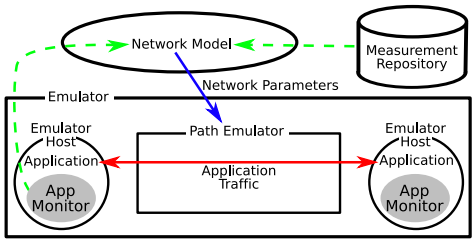

Figure 1: Architecture of the Flexlab framework. Any network model can be "plugged in," and can optionally use data from the application monitors or measurement repository.

2.1 Emulator

Flexlab runs on top of the Emulab testbed management system, which provides critical management infrastructure. It provides automated setup of emulated experiments by configuring hosts, switches, and path emulators within minutes. Emulab also provides a "full-service" interface for distributing experimental applications to nodes, controlling those applications, collecting packet traces, and gathering of log files and other results. These operations can be controlled and (optionally) fully automated by a flexible, secure event system. Emulab's portal extends all of these management benefits to PlanetLab nodes. This makes Emulab an ideal platform for Flexlab, as users can easily move back and forth between emulation, live experimentation, and Flexlab experimentation. New work [10] integrates a full experiment and data management system into Emulab—indeed, we used that "workbench" to gather and manage many of the results in this paper.

2.2 Application Monitor

The application monitor reports on the network operations performed by the application, such as the connections it makes, its packet sends and receives, and the socket options it sets. This information can be sent to the network model, which can use it to track which paths the application uses and discover the application's offered network load. Knowing the paths in use aids the network model by limiting the set of paths it must measure or compute; most applications will use only a small subset of the n^2 paths between n hosts. We describe the monitor in more detail later.

2.3 Path Emulator

The path emulator shapes traffic from the emulator hosts. It can, for example, queue packets to emulate delay, de-

queue packets at a specific rate to control bandwidth, and drop packets from the end of the queue to emulate saturated router queues. Our path emulator is an enhanced version of FreeBSD's Dummynet [28]. We have made extensive improvements to Dummynet to add support for the features discussed in Section 5.2, as well as adding support for jitter and for several distributions: uniform, Poisson, and arbitrary distributions determined by user-supplied tables. Dummynet runs on separate hosts from the application, both to reduce contention for host resources, and so that applications can be run on any operating system.

For Flexlab we typically configure Dummynet so that it emulates a "cloud," abstracting the Internet as a set of per-flow pairwise network characteristics. This is a significant departure from Emulab's typical use: it is typically used with router-level topologies, although the topologies may be somewhat abstracted. The cloud model is necessary for us because our current models deal with end-to-end conditions, rather than trying to reverse engineer the Internet's router-level topology.

A second important piece of our path emulator is its control system. The path emulator can be controlled with Emulab's event system, which is built on a publish/subscribe system. "Delay agents" on the emulator nodes subscribe to events for the paths they are emulating, and update characteristics based on the events they receive. Any node can publish new characteristics for paths, which makes it easy to support both centralized and distributed implementations of network models. For example, control is equally easy by a single process that computes all model parameters or by a distributed system in which measurement agents independently compute the parameters for individual paths. The Emulab event system is lightweight, making it feasible to implement highly dynamic network models that send many events per second, and it is secure: event senders can affect only their own experiments.

2.4 Network Model

The network model supplies network conditions and parameters to the path emulator. The network model is the least-constrained component of the Flexlab architecture; the only constraint on a model implementation is that it must configure the path emulator through the event system. Thus, a wide variety of models can be created. A model may be static, setting network characteristics once at the beginning of an experiment, or dynamic, keeping them updated as the experiment proceeds. Dynamic network settings may be sent in real-time as the experiment proceeds, or the settings may be pre-computed and scheduled for delivery by Emulab's event scheduler.

We have implemented three distinct network models, discussed later. All of our models pair up each emulator node with a node in the overlay network, attempting to give the emulator node the same view of network characteristics as its peer in the overlay. The architecture, however, does not require that models come directly from overlay measurements. Flexlab can just as easily be used with network models from other sources, such as analytic models.

2.5 Measurement Repository

Flexlab's measurements are currently stored in Andersen and Feamster's Datapository. Information in the Datapository is available for use in constructing or parameterizing network models, and the networking community is encouraged to contribute their own measurements. We describe Flexlab's measurement system in the next section.

3 Wide-area Network Monitoring

Good measurements of Internet conditions are important in a testbed context for two reasons. First, they can be used as input for network models. Second, they can be used to select Internet paths that tend to exhibit a chosen set of properties. To collect such measurements, we developed and deployed a wide area network monitor, Flexmon. It has been running for a year, placing into the Datapository half a billion measurements of connectivity, latency, and bandwidth between PlanetLab hosts. Flexmon's design provides a measurement infrastructure that is shared, reliable, safe, adaptive, controllable, and accommodates high-performance data retrieval. Flexmon has some features in common with other measurement systems such as S^3 [39] and Scriptroute [32], but is designed for shared control over measurements and the specific integration needs of Flexlab.

Flexmon, shown in Figure 2, consists of five components: *path probers*, the *data collector*, the *manager*, *manager clients*, and the *auto-manager client*. A path prober runs on each PlanetLab node, receiving control commands from a central source, the manager. A command may change the measurement destination nodes, the type of measurement, and the frequency of measurement. Commands are sent by experimenters, using a manager client, or by the auto-manager client. The purpose of the auto-manager client is to maintain measurements between all PlanetLab sites. The auto-manager client chooses the least CPU-loaded node at each site to include in its measurement set, and makes needed changes as nodes and sites go up and down. The data collector runs on a server in Emulab, collecting measurement results from each path prober and storing them in the Datapository. To speed up both queries and updates, it contains a write-back cache in the form of a small database instance.

Due to the large number of paths between PlanetLab nodes, Flexmon measures each path at fairly low frequency—approximately every 2.5 hours for bandwidth, and 10 minutes for latency. To get more detail, experimenters can control Flexmon's measurement frequency of any path. Flexmon maintains a global picture of the net-

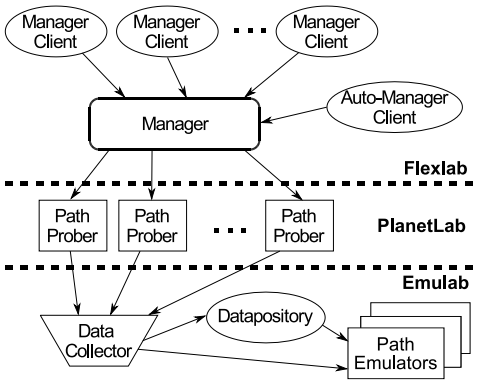

Figure 2: The components of Flexmon and their communication.

work resources it uses, and caps and adjusts the measurement rates to maintain safety to PlanetLab.

Flexmon currently uses simple tools to collect measurements: `iperf` for bandwidth, and `fping` for latency and connectivity. We had poor results from initial experiments with packet-pair and packet-train tools, including `pathload` and `pathchirp`. Our guiding principles thus far have been that the simpler the tool, the more reliable it typically is, and that the most accurate way of measuring the bandwidth available to a TCP stream is to use a TCP stream. Flexmon has been designed, however, so that it is relatively simple to plug in other measurement tools. For example, tools that trade accuracy for reduced network load or increased scalability [8, 13, 21, 23] could be used, or we could take opportunistic measurements of large file transfers by the CDNs on PlanetLab.

Flexmon's reliability is greatly improved by buffering results at each path prober until an acknowledgment is received from the data collector. Further speedup is possible by directly pushing new results to requesting Flexlab experiments instead of having them poll the database.

4 Simple Measurement-Driven Models

We have used measurements taken by Flexmon to build two simple, straightforward network models. These models represent incremental improvements over the way emulators are typically used today. Experimenters typically choose network parameters on an ad hoc basis and keep them constant throughout an experiment. Our *simple-static* model improves on this by using actual measured Internet conditions. The *simple-dynamic* model goes a step further by updating conditions as the experiment proceeds. Because the measurements used by these models are stored permanently in the Datapository, it is trivial to "replay" network conditions starting at any point in the past. Another benefit is that the simple models run entirely outside of the emulated environment itself, meaning that no restrictions are placed on the protocols, applications, or operating systems that run on the emulator hosts. The simple models do have some weaknesses, which we discuss in this section. These weaknesses are addressed by our more sophisticated model, ACIM, in Section 5.

4.1 Simple-static and Simple-dynamic

In both the simple-static and simple-dynamic models, each PlanetLab node in an experiment is associated with a corresponding emulation node in Emulab. A program called `dbmonitor` runs on an Emulab server, collecting path characteristics for each relevant Internet path from the Datapository. It applies the characteristics to the emulated network via the path emulator.

In simple-static mode, `dbmonitor` starts at the beginning of an experiment, reads the path characteristics from the DB, issues the appropriate events to the emulation agents, and exits. This model places minimal load on the path emulators and the emulated network, at the expense of fidelity. If the real path characteristics change during an experiment, the emulated network becomes inaccurate.

In simple-dynamic mode the experimenter controls the frequencies of measurement and emulator update. Before the experiment starts, `dbmonitor` commands Flexmon to increase the frequency of probing for the set of PlanetLab nodes involved in the experiment. Similarly, `dbmonitor` queries the DB and issues events to the emulator at the specified frequency, typically on the order of seconds. The dynamic model addresses some of the fidelity issues of the simple-static model, but it is still constrained by practical limits on measurement frequency.

4.2 Stationarity of Network Conditions

The simple models presented in this section are limited in the detail they can capture, due to a fundamental tension. We would like to take frequent measurements, to maximize the models' accuracy. However, if they are too frequent, measurements of overlapping paths (such as from a single source to several destinations) will necessarily overlap, causing interference that may perturb the network conditions. Thus, we must limit the measurement rate.

To estimate the effect that low measurement rates have on accuracy, we performed an experiment. We sent pings between pairs of nodes every 2 seconds for 30 minutes. We analyzed the latency distribution to find "change points" [33], which are times when the mean value of the latency samples changes. This statistical technique was used in a classic paper on Internet stationarity [41]; our method is similar to their "CP/Bootstrap" test. This analysis provides insight into the required measurement frequency—the more significant events missed, the poorer the accuracy of a measurement.

Table 1 shows some of the results from this test. We used representative nodes in Asia, Europe, and North America. One set of North American nodes was connected to the commercial Internet, and the other set to Internet2. The

Path	High	Low	Change
Asia to Asia	2	1	0.13%
Asia to Commercial	2	0	2.9%
Asia to Europe	4	0	0.5%
Asia to I2	6	0	0.59%
Commercial to Commercial	**20**	2	**39%**
Commercial to Europe	4	0	3.4%
Commercial to I2	**13**	1	**15%**
I2 to I2	4	0	0.02%
I2 to Europe	0	0	–
Europe to Europe	**9**	1	**12%**

Table 1: Change point analysis for latency.

first column shows the number of change points seen in this half hour. In the second column, we have simulated measurement at lower frequencies by sampling our high-rate data; we used only one of every ten measurements, yielding an effective sampling interval of 20 seconds. Finally, the third column shows the magnitude of the median change, in terms of the median latency for the path.

Several of the paths are largely stable with respect to latency, exhibiting few change points even with high-rate measurements, and the magnitude of the few changes is low. However, three of the paths (in bold) have a large number of change points, and those changes are of significant magnitude. In all cases, the low-frequency data misses almost all change points. In addition, we cannot be sure that our high-frequency measurements have found all change points. The lesson is that there are enough significant changes at small time scales to justify, and perhaps even necessitate, high-frequency measurements.

In Section 5, we describe application-centric Internet modeling, which addresses this accuracy problem by using the application's own traffic patterns to make measurements. In that case, the only load on the network, and the only self-interference induced, is that which would be caused by the application itself.

4.3 Modeling Shared Bottlenecks

There is a subtle complexity in network emulation based on path measurements of available bandwidth. This complexity arises when an application has multiple simultaneous network flows associated with a single node in the experiment. Because Flexmon obtains pairwise available bandwidth measurements using independent `iperf` runs, it does not reveal bottlenecks shared by multiple paths. Thus, independently modeling flows originating at the same host but terminating at different hosts can cause inaccuracies if there are shared bottlenecks. This is mitigated by the fact that if there is a high degree of statistical multiplexing on the shared bottleneck, interference by other flows dominates interference by the application's own flows [14]. In that case, modeling the application's flows as independent is still a reasonable approximation.

In the "cloud" configuration of Dummynet we model flows originating at the same host as being non-interfering.

Path	Sum of multiple TCP flows		
	1 flow	5 flows	10 flows
Commodity Internet Paths			
PCH to IRO	485 K	585 K	797 K
IRP to UCB-DSL	372 K	507 K	589 K
PBS to Arch. Tech.	348 K	909 K	952 K
Internet2 Paths			
Illinois to Columbia	3.95 M	9.05 M	9.46 M
Maryland to Calgary	3.09 M	15.4 M	30.4 M
Colorado St. to Ohio St.	225 K	1.20 M	1.96 M

Table 2: Available bandwidth estimated by multiple `iperf` flows, in bits per second. The PCH to IRO path is administratively limited to 10 megabits, and the IRP to UCB-DSL path is administratively limited to 1 megabit.

To understand how well this assumption holds, we measured multiple simultaneous flows on PlanetLab paths, shown in Table 2. For each path we ran three tests in sequence for 30 seconds each: a single TCP `iperf`, five TCP `iperf`s in parallel, and finally ten TCP `iperf`s in parallel. The reverse direction of each path, not shown, produced similar results.

Our experiment revealed a clear distinction between paths on the commodity Internet and those on Internet2 (I2). On the commodity Internet, running more TCP flows achieves only marginally higher aggregate throughput. On I2, however, five flows always achieve much higher throughput than one flow. In all but one case, ten flows also achieve significantly higher throughput than five. Thus, our previous assumption of non-interference between multiple flows holds true for the I2 paths tested, but not for the commodity Internet paths.

This difference may be a consequence of several possible factors. It could be due to the fundamental properties of these networks, including proximity of bottlenecks to the end hosts and differing degrees of statistical multiplexing. It could also be induced by peculiarities of PlanetLab. Some sites impose administrative limits on the amount of bandwidth PlanetLab hosts may use, PlanetLab attempts to enforce fair-share network usage between slices, and the TCP stack in the PlanetLab kernel is not tuned for high performance on links with high bandwidth-delay products (in particular, TCP window scaling is disabled).

To model this behavior, we developed additional simple Dummynet configurations. In the "shared" configuration, a node is assumed to have a single bottleneck that is shared by all of its outgoing paths, likely its last-mile link. In the "hybrid" configuration, some paths use the cloud model and others the shared model. The rules for hybrid are: If a node is an I2 node, it uses the cloud model for I2 destination nodes, and the shared model for all non-I2 destination nodes. Otherwise, it uses the shared model for all destinations. The bandwidth for shared pipes is set to the maximum found for any destination in the experiment. Flexlab users can select which Dummynet configuration to use.

Clearly, more sophisticated shared-bottleneck models are possible for the simple models. For example, it might be possible to identify bottlenecks with Internet tomography, such as iPlane [21]. Our ACIM model, discussed next, takes a completely different approach to the shared-bottleneck problem.

5 Application-Centric Internet Modeling

The limitations of our simple models lead us to develop a more complex technique, *application-centric Internet modeling*. The difficulties in simulating or emulating the Internet are well known [12, 20], though progress is continually made. Likewise, creating good *general-purpose* models of the Internet is still an open problem [11]. While progress has been made on measuring and modeling aspects of the Internet sufficient for certain uses, such as improving overlay routing or particular applications [21, 22], the key difficulty we face is that a general-purpose emulator, in theory, has a stringent accuracy criterion: it must yield accurate results for *any* measurement of *any* workload.

ACIM approaches the problem by *modeling the Internet as perceived by the application*—as viewed through its limited lens. We do this by running the application and Internet measurements simultaneously, using the application's behavior *running inside Emulab* to generate traffic *on PlanetLab* and collect network measurements. The network conditions experienced by this replicated traffic are then applied, in near real-time, to the application's emulated network environment.

ACIM has five primary benefits. The first is in terms of node and path scaling. A particular instance of any application will use a tiny fraction of all of the Internet's paths. By confining measurement and modeling only to those paths that the application actually uses, the task becomes more tractable. Second, we avoid numerous measurement and modeling problems, by assessing end-to-end behavior rather than trying to model the intricacies of the network core. For example, we do not need precise information on routes and types of outages—we need only measure their effects, such as packet loss and high latency, on the application. Third, rare or transient network effects are immediately visible to the application. Fourth, it yields accurate information on how the network will react to the offered load, automatically taking into account factors that are difficult or impossible to measure without direct access to the bottleneck router. These factors include the degree of statistical multiplexing, differences in TCP implementations and RTTs of the cross traffic, the router's queuing discipline, and unresponsive flows. Fifth, it tracks conditions quickly, by creating a feedback loop which contiually adjusts offered loads and emulator settings in near real-time.

ACIM is *precise* because it assesses only relevent parts of the network, and it is *complete* because it automatically accounts for all potential network-related behavior. (Of course, it is precise in terms of paths, not traffic.) Its concrete approach to modeling and its level of fidelity should provide an environment that experimenters can trust when they do not know their application's dependencies.

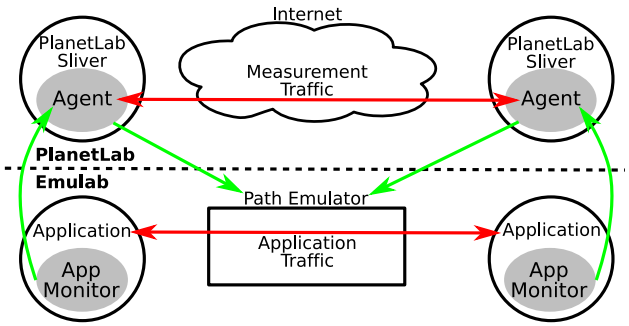

Figure 3: The architecture and data flow of application-centric Internet modeling.

Our technique makes two common assumptions about the Internet: that the location of the bottleneck link does not change rapidly (though its characteristics may), and that most packet loss is caused by congestion, either due to cross traffic or its own traffic. In the next section, we first concentrate on TCP flows, then explain how we have extended the concepts to UDP.

5.1 Architecture

We pair each node in the emulated network with a peer in the live network, as shown in Figure 3. The portion of this figure that runs on PlanetLab fits into the "network model" portion of the Flexlab architecture shown in Figure 1. The ACIM architecture consists of three basic parts: an application monitor which runs on Emulab nodes, a measurement agent which runs on PlanetLab nodes, and a path emulator connecting the Emulab nodes. The agent receives characteristics of the application's offered load from the monitor, replicates that load on PlanetLab, determines path characteristics through analysis of the resulting TCP stream, and sends the results back into the path emulator as traffic shaping parameters. We now detail each of these parts.

Application Monitor on Emulab. The application monitor runs on each node in the emulator and tracks the network calls made by the application under test. It tracks the application's network activity, such as connections made and data sent on those connections. The monitor uses this information to create a simple model of the offered network load and sends this model to the measurement agent on the corresponding PlanetLab node. The monitor supports both TCP and UDP sockets. It also reports on important socket options, such as socket buffer sizes and the state of TCP's `TCP_NODELAY` flag.

We instrument the application under test by linking it with a library we created called `libnetmon`. This library's purpose is to provide the model with information about the

application's network behavior. It wraps network system calls such as `connect()`, `accept()`, `send()`, `sendto()`, and `setsockopt()`, and informs the application monitor of these calls. In many cases, it summarizes: for example, we do not track the full contents of `send()` calls, simply their sizes and times. `libnetmon` can be dynamically linked into a program using the `LD_PRELOAD` feature of modern operating systems, meaning that most applications can be run without modification. We have tested `libnetmon` with a variety of applications, ranging from `iperf` to Mozilla Firefox to Sun's JVM.

By instrumenting the application directly, rather than snooping on network packets it puts on the wire, we are able to measure the application's *offered load* rather than simply the *throughput achieved*. This distinction is important, because the throughput achieved is, at least in part, a function of the parameters the model has given to the path emulator. Thus, we cannot assume that what an application is *able* to do is the same as what it is *attempting* to do. If, for example, the available bandwidth on an Internet path increases, so that it becomes greater than the bandwidth setting of the corresponding path emulator, offering only the achieved throughput on this path would fail to find the additional available bandwidth.

Measurement Agent on PlanetLab. The measurement agent runs on PlanetLab nodes, and receives information from the application monitor about the application's network operations. Whenever the application running on Emulab connects to one of its peers (also running inside Emulab), the measurement agent likewise connects to the agent representing the peer. The agent uses the simple model obtained by the monitor to generate similar network load; the monitor keeps the agent informed of the `send()` and `sendto()` calls made by the application, including the amount of data written and the time between calls. The agent uses this information to recreate the application's network behavior, by making analogous `send()` calls. Note that the offered load model does not include the application's packet payload, making it relatively lightweight to send from the monitor to the agent.

The agent uses `libpcap` to inspect the resulting packet stream and derive network conditions. For every ACK it receives from the remote agent, it calculates instantaneous throughput and RTT. For TCP, we use TCP's own ACKs, and for UDP, we add our own application-layer ACKs. The agent uses these measurements to generate parameters for the path emulator, discussed below.

5.2 Inference and Emulation of Path Conditions

Our path emulator is an enhanced version of the Dummynet traffic shaper. We emulate the behavior of the bottleneck router's queue within this shaper as shown in Figure 4. Dummynet uses two queues: a bandwidth queue, which emulates queuing delay, and a delay queue, which models

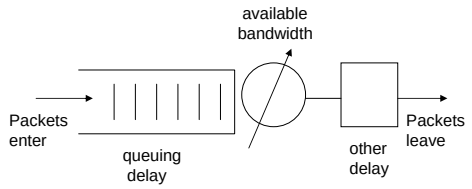

Figure 4: Path emulation

all other sources of delay, such as propagation, processing, and transmission delays. Thus, there are three important parameters: the size of the bandwidth queue, the rate at which it drains, and the length of time spent in the delay queue. Since we assume that most packet loss is caused by congestion, we induce loss only by limiting the size of the bandwidth queue and the rate it drains.

Because the techniques in this section require that there be application traffic to measure, we use the simple-static model to set initial conditions for each path. They will only be experienced by the first few packets; after that, ACIM provides higher-quality measurements.

Bandwidth Queue Size. The bandwidth queue has a finite size, and when it is full, packets arriving at the queue are dropped. The bottleneck router has a queue whose maximum capacity is measured in terms of bytes and/or packets, but it is difficult to directly measure either of these capacities. Sommers et al. [29] proposed using the maximum one-way delay as an approximation of the size of the bottleneck queue. This approach is problematic on PlanetLab because of the difficulty of synchronizing clocks, which is required to calculate one-way delay. Instead, we approximate the size of the queue in terms of *time*—that is, the longest time one of our packets has spent in the queue without being dropped. We assume that congestion will happen mostly along the forward edge of a network path, and thus can approximate the maximum queuing delay by subtracting the minimum RTT from the maximum RTT. We refine this number by finding the maximum queuing delay just before a loss event.

Available Bandwidth. TCP's fairness (the fraction of the capacity each flow receives) is affected by differences in the RTTs of flows sharing the link [18]. Measuring the RTTs of flows we cannot directly observe is difficult or impossible. Thus, the most accurate way to determine how the network will react to the load offered by a new flow is to offer that load and observe the resulting path properties.

We observe the inter-send times of acknowledgment packets and the number of bytes acknowledged by each packet to determine the instantaneous goodput of a connection: $goodput = (bytes\ acked)/(time\ since\ last\ ack)$. We then estimate the throughput of a TCP connection between PlanetLab nodes by computing a moving average of the instantaneous goodput measurements for the preceding half-second. This averages out any outliers, allowing for a

more consistent metric.

This measurement takes into account the reactivity of other flows in the network. While calculating this goodput is straightforward, there are subtleties in mapping to available bandwidth. The traffic generated by the measurement agent may not fully utilize the available bandwidth. For instance, if the load generated by the application is lower than the available bandwidth, or TCP fills the receive window, the throughput does not represent available bandwidth. When this situation is detected, we should not cap the emulator bandwidth to that artificially slow rate. Thus, we *lower* the bandwidth used by the emulator only if we detect that we are fully loading the PlanetLab path. If we see a goodput that is higher than the goodput when we last saturated the link, then the available bandwidth must have increased, and we *raise* the emulator bandwidth.

Queuing theory shows that when a buffered link is overutilized, the time each packet spends in the queue, and thus the observed RTT, increases for each successive packet. Additionally, `send()` calls tend to block when the application is sending at a rate sufficient to saturate the bottleneck link. In practice, since each of these signals is noisy, we use a combination of them to determine when the bottleneck link is saturated. To determine whether RTT is increasing or decreasing, we find the slope of RTT vs. sample number using least squares linear regression.

Other Delay. The measurement agent takes fine-grained latency measurements. It records the time each packet is sent, and when it receives an ACK for that packet, calculates the RTT seen by the most recent acknowledged packet. For the purposes of emulation, we calculate the "Base RTT" the same way as TCP Vegas [5]: that is, the minimum RTT recently seen. This minimum delay accounts for the propagation, processing, and transmission delays along the path with a minimum of influence by queuing delay.

We set the delay queue's delay to half the base RTT to avoid double-counting queuing latency, which is modeled in the bandwidth queue.

Outages and Rare Events. There are many sources of outages and other anomalies in network characteristics. These include routing anomalies, link failures, and router failures. Work such as PlanetSeer [40] and numerous BGP studies seeks to explain the causes of these anomalies. Our application-centric model has an easier task: to faithfully reproduce the effect of these rare events, rather than finding the underlying cause. Thus, we observe the features of these rare events that are *relevant* to the application. Outages can affect Flexlab's control plane, however, by cutting off Emulab from one or more PlanetLab nodes. In future work, we can improve robustness by using an overlay network such as RON [2].

Per-Flow Emulation. In our application-centric model, the path emulator is used to shape traffic on a per-flow rather than a per-path basis. If there is more than one flow using a path, the bandwidth seen by each flow depends on many variables, including the degree of statistical multiplexing on the bottleneck link, when the flows begin, and the queuing policy on the bottleneck router. We let this contention for resources occur in the overlay network, and reflect the results into the emulator by per-flow shaping.

5.3 UDP Sockets

ACIM for UDP differs in some respects from ACIM for TCP. The chief difference is that there are no protocol-level ACKs in UDP. We have implemented a custom application-layer protocol on top of UDP that adds the ACKs needed for measuring RTT and throughput. This change affects only the replication and measurement of UDP flows; path emulation remains unchanged.

Application Layer Protocol. Whereas the TCP ACIM sends random payloads in its measurement packets, UDP ACIM runs an application-layer protocol on top of them. The protocol embeds sequence numbers in the packets on the forward path, and on the reverse path, sequence numbers and timestamps acknowledge received packets. Our protocol requires packets to be at least 57 bytes long; if the application sends packets smaller than this, the measurement traffic uses 57-byte packets.

Unlike TCP, our UDP acknowledgements are selective, not cumulative, and we also do not retransmit lost packets. We do not need *all* measurement traffic to get through, we simply measure how much does. An ACK packet is sent for every data packet received, but each ACK packet contains ACKs for several recent data packets. This redundancy allows us to get accurate bandwidth numbers without re-sending lost packets, and works in the face of moderate ACK packet loss.

Available Bandwidth. Whenever an ACK packet is received at the sender, goodput is calculated as $g = s/(t_n - t_{n-1})$, where g is goodput, s is the size of the data being acknowledged, t_n is the receiver timestamp for the current ACK, and t_{n-1} is the last receiver ACK timestamp received. By using inter-packet timings from the receiver, we avoid including jitter on the ACK path in our calculations, and the clocks at the sender and receiver need not be synchronized. Throughput is calculated as a moving average over the last 100 acknowledged packets or half second, whichever is less. If any packet loss has been detected, this throughput value is fed to the application monitor as the available bandwidth on the forward path.

Delay measurements. Base RTT and queuing delay are computed the same way for UDP as they are for TCP.

Reordering and Packet Loss. Because TCP acknowledgements are cumulative, reordering of packets on the forward path is implicitly taken care of. We have to handle it explicitly in the case of UDP. Our UDP measurement protocol can detect packet reordering in both directions. Be-

cause each ACK packet carries redundant ACKs, reordering on the reverse path is not of concern. A data packet is considered to be lost if ten packets sent after it have been acknowledge. It is also considered lost if the difference between the receipt time of the latest ACK and the send time of the data packet is greater than $10 \cdot (average\ RTT + 4 \cdot standard\ deviation\ of\ recent\ RTTs)$.

5.4 Challenges

Although the design of ACIM is straightforward when viewed at a high level, there are a host of complications that limit the accuracy of the system. Each was a significant barrier to implementation; we describe two.

Libpcap Loss. We monitor the connections on the measurement agent with `libpcap`. The `libpcap` library copies a part of each packet as it arrives or leaves the (virtual) interface and stores them in a buffer pending a query by the application. If packets are added to this buffer faster than they are removed by the application, some of them may be dropped. The scheduling behavior described in Appendix A is a common cause of this occurrence, as processes can be starved of CPU for hundreds of milliseconds. These dropped packets are still seen by the TCP stack in the kernel, but they are not seen by the application.

This poses two problems. First, we found it not uncommon for all packets over a long period of time (up to a second) to be dropped by the `libpcap` buffer. In this case it is impossible to know what has occurred during that period. The connection may have been fully utilizing its available bandwidth or it may have been idle during part of that time, and there is no way to reliably tell the difference. Second, if only one or a few packets are dropped by the `libpcap` buffer, the "falseness" of the drops may not be detectable and may skew the calculations.

Our approach is to reset our measurements after periods of detected loss, no matter how small. This avoids the potential hazards of averaging measurements over a period of time when the activity of the connection is unknown. The downside is that in such a situation, a change in bandwidth would not be detected as quickly and we may average measurements over non-contiguous periods of time. We know of no way to reliably detect which stream(s) a `libpcap` loss has affected in all cases, so we must accept that there are inevitable limits to our accuracy.

Ack Bursts. Some paths on PlanetLab have anomalous behaviors. The most severe example of this is a path that delivers bursts of acknowledgments over small timescales. In one case, acks that were sent over a period of 12 milliseconds arrived over a period of less than a millisecond, an order of magnitude difference. This caused some over-estimation of delay (by up to 20%), and an order of magnitude over-estimation of throughput. We cope with this phenomenon in two ways. First, we use TCP timestamps to obtain the ACK inter-departure times on the receiver rather than the ACK inter-arrival times on the sender. This technique corrects for congestion and other anomalies on the reverse path. Second, we lengthened the period over which we average (to about 0.5 seconds), which is also needed to dampen excessive jitter.

6 Evaluation

We evaluate Flexlab by presenting experimental results from three microbenchmarks and a real application. Our results show that Flexlab is more faithful than simple emulation, and can remove artifacts of PlanetLab host conditions. Doing a rigorous *validation* of Flexlab is extremely difficult, because it seems impossible to establish ground truth: each environment being compared can introduce its own artifacts. Shared PlanetLab nodes can hurt performance, experiments on the live Internet are fundamentally unrepeatable, and Flexlab might introduce artifacts through its measurement or path emulation. With this caveat, our results show that for at least some complex applications running over the Internet, Flexlab with ACIM produces more accurate and realistic results than running with the host resources typically available on PlanetLab, or in Emulab without network topology information.

6.1 Microbenchmarks

We evaluate ACIM's detailed fidelity using `iperf`, a standard measurement tool that simulates bulk data transfers. `iperf`'s simplicity makes it ideal for microbenchmarks, as its behavior is consistent between runs. With TCP, it simply sends data at the fastest possible rate, while with UDP it sends at a specified constant rate. The TCP version is, of course, highly reactive to network changes.

As in all of our experiments, each application tested on PlanetLab and each major Flexlab component (measurement agent, Flexmon) are run in separate slices.

6.1.1 TCP iperf and Cross-Traffic

Figure 5 shows the throughput of a representative two minute run in Flexlab of `iperf` using TCP. The top graph shows throughput achieved by the measurement agent, which replicated `iperf`'s offered load on the Internet between AT&T and the Univ. of Texas at Arlington. The bottom graph shows the throughput of `iperf` itself, running on an emulated path and dedicated hosts inside Flexlab.

To induce a change in available bandwidth, between times 35 and 95 we sent cross-traffic on the Internet path, in the form of ten `iperf` streams between other PlanetLab nodes at the same sites. Flexlab closely tracks the changed bandwidth, bringing the throughput of the path emulator down to the new level of available bandwidth. It also tracks network changes that we did not induce, such as the one at time 23. However, brief but large drops in throughput occasionally occur in the PlanetLab graph but not the Flexlab

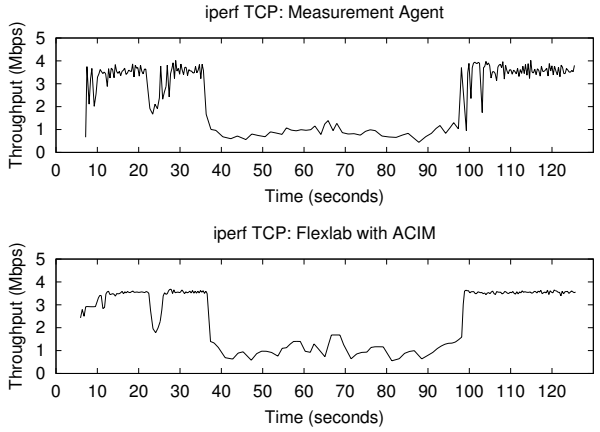

Figure 5: Application-centric Internet modeling, comparing agent throughput on PlanetLab (top) with the throughput of the application running in Emulab and interacting with the model (bottom).

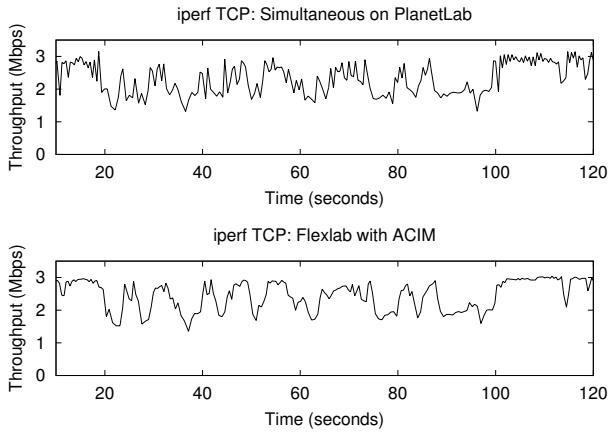

Figure 6: Comparison of the throughput of a TCP `iperf` running on PlanetLab (top) with a TCP `iperf` simultaneously running under Flexlab with ACIM (bottom).

graph, such as those starting at time 100. Through log file analysis we determined that these drops are due to temporary CPU starvation on PlanetLab, preventing even the lightweight measurement agent from sustaining the sending rate of the real application. These throughput drops demonstrate the impact of the PlanetLab scheduling delays documented in Appendix A. The agent correctly determines that these reductions in throughput are not due to available bandwidth changes, and deliberately avoids mirroring these PlanetLab host artifacts on the emulated path. Finally, the measurement agent's throughput exhibits more jitter than the application's, showing that we could probably further improve ACIM by adding a jitter model.

6.1.2 Simultaneous TCP iperf Runs

ACIM is designed to subject an application in the emulator to the same network conditions that application would

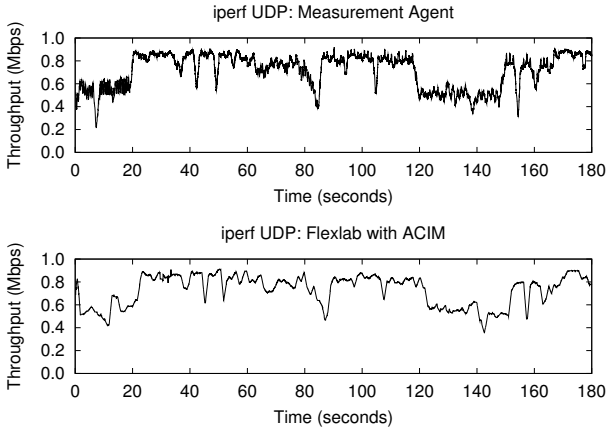

Figure 7: The UDP throughput of `iperf` (below) compared with the actual throughput successfully sent by the measurement agent (above) when using the ACIM model in Flexlab.

see on the Internet. To evaluate how well ACIM meets this goal, we compared two instances of `iperf`: one on PlanetLab, and one in Flexlab. Because we cannot expect runs done on the Internet at different times to show the same results, we ran these two instances simultanously. The top graph in Figure 6 shows the throughput of `iperf` run directly on PlanetLab between NEC Labs and Intel Research Seattle. The bottom graph shows the throughput of another `iperf` run at the same time in Flexlab, between the same "hosts." As network characteristics vary over the connection's lifetime, the throughput graphs correspond impressively. The average throughputs are close: PlanetLab was 2.30 Mbps, while Flexlab was 2.41 Mbps (4.8% higher). These results strongly suggest that ACIM has high fidelity. The small difference may be due to CPU load on PlanetLab; we speculate that difference is small because `iperf` consumes few host resources, unlike a real application on which we report shortly.

6.1.3 UDP iperf

We have made an initial evaluation of the UDP ACIM support, which is newer than our TCP support. We used a single `iperf` to generate a 900 Kbps UDP stream. As in Sec. 6.1.1, we measured the throughput achieved by both the measurement agent on PlanetLab and the `iperf` stream running on Flexlab. The graphs in Figure 7 closely track each other. The mean throughputs are close: 746 Kbps for Iperf, and 736 Kbps for the measurement agent, 1.3% lower. We made three similar runs between these nodes, at target rates varying from 800–1200 Kbps. The differences in mean throughput were similar: -2.5%, 0.4%, and 4.4%. ACIM's UDP accuracy appears very good in this range. A more thorough evaluation is future work.

6.2 Macrobenchmark: BitTorrent

This next set of experiments demonstrates several things: first, that Flexlab is able to handle a real, complex, distributed system that is of interest to researchers; second, that PlanetLab host conditions can make an enormous impact on the network performance of real applications; third, that both Flexlab and PlanetLab with host CPU reservations give similar and likely accurate results; and fourth, preliminary results indicate that our simple static models of the Internet don't (yet) provide high-fidelity emulation.

BitTorrent (BT) is a popular peer-to-peer program for cooperatively downloading large files. Peers act as both clients and servers: once a peer has downloaded part of a file, it serves that to other peers. We modified BT to use a static tracker to remove some–but by no means all—sources of non-determinism from repeated BT runs. Each experiment consisted of a seeder and seven BT clients, each located at a different site on Internet2 or GÉANT, the European research network.[1] We ran the experiments for 600 seconds, using a file that was large enough that no client could finish downloading it in that period.

6.2.1 ACIM vs. PlanetLab

We began by running BT in a manner similar to the simultaneous `iperf` microbenchmark described in Sec. 6.1.2. We ran two instances of BT simultaneously: one on PlanetLab and one using ACIM on Flexlab. These two sets of clients did not communicate directly, but they did compete for bandwidth on the same paths: the PlanetLab BT directly sends traffic on the paths, while the Flexlab BT causes the measurement agent to send traffic on those same paths.

Figure 8 shows the download rates of the BT clients, with the PlanetLab clients in the top graph, and the Flexlab clients in the bottom. Each line represents the download rate of a single client, averaged over a moving window of 30 seconds. The PlanetLab clients were only able to sustain an average download rate of 2.08 Mbps, whereas those on Flexlab averaged triple that rate, 6.33 Mbps. The download rates of the PlanetLab clients also clustered much more tightly than in Flexlab. A series of runs showed that the clustering was consistent behavior. Table 3 summarizes those runs, and shows that the throughput differences were also repeatable, but with Flexlab higher by a factor of 2.5 instead of 3.

[1]The sites were stanford.edu (10Mb), uoregon.edu (10Mb), cmu.edu (5Mb), usf.edu, utep.edu, kscy.internet2.planet-lab.org, uni-klu.ac.at, and tssg.org. The last two are on GÉANT; the rest on I2. Only the first three had imposed bandwidth limits. All ran PlanetLab 3.3, which contained a bug which enforced the BW limits even between I2 sites. We used the official BT program v. 4.4.0, which is in Python. All BT runs occurred in February 2007. 5 & 15 minute load averages for all nodes except the seeder were typically 1.5 (range 0.5–5); the seed (Stanford) had a loadavg of 14–29, but runs with a less loaded seeder gave similar results. Flexlab/Emulab hosts were all "pc3000"s: 3.0 Ghz Xeon, 2GB RAM, 10K RPM SCSI disk.

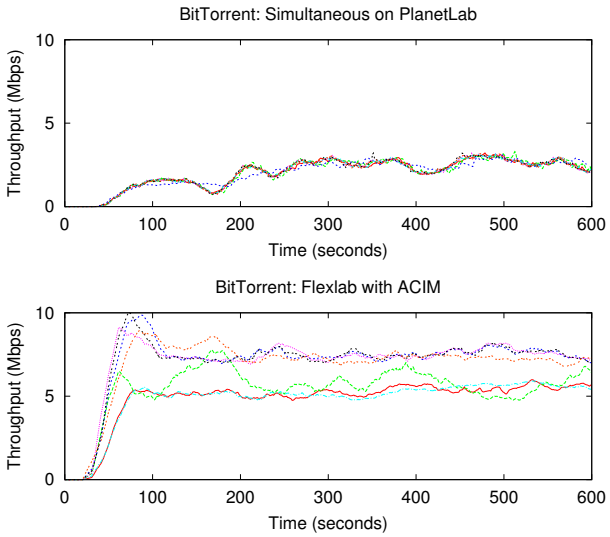

Figure 8: A comparison of download rates of BT running simultaneously on PlanetLab (top) and Flexlab using ACIM (bottom). The seven clients in the PlanetLab graph are tightly clustered.

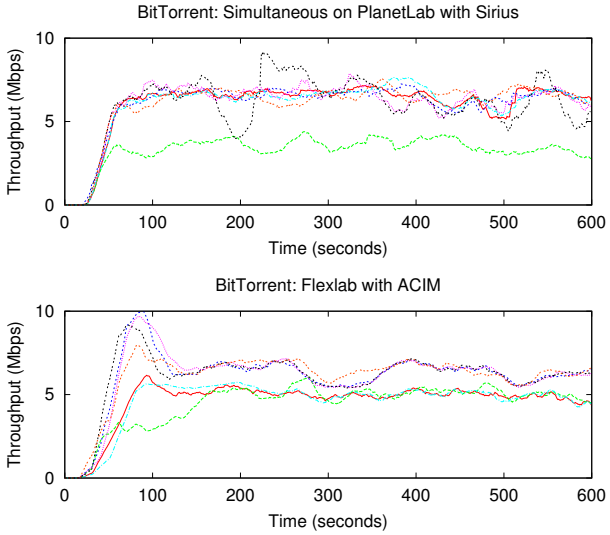

Figure 9: Download rates of BT simultaneously running on PlanetLab with Sirius (top), compared to Flexlab ACIM (bottom).

These results, combined with the accuracy of the microbenchmarks, suggest that BT's throughput on PlanetLab is constrained by host overload not found in Flexlab. Our next experiment attempts to test this hypothesis.

6.2.2 ACIM vs. PlanetLab with Sirius

Sirius is a CPU and bandwidth reservation system for PlanetLab. It ensures that a sliver receives at least 25% of its host's CPU, but does not give priority access to other host resources such as disk I/O or RAM. Normal Sirius also includes a bandwidth reservation feature, but to isolate the ef-

Experiment	Flexlab	PlanetLab	Ratio
No Sirius (6 runs)	5.78 (0.072)	2.27 (0.074)	2.55 (0.088)
Sirius (5 runs)	5.44 (0.29)	5.24 (0.34)	1.04 (0.045)

Table 3: Mean BT download rate in Mbps and std. dev. (in parentheses) of multiple Flexlab and PlanetLab runs, as in Sec. 6.2. Since these were run at a different time, network conditions may have changed.

fects of CPU sharing, we had PlanetLab operations disable this feature in our Sirius slice. Currently, only one slice, PlanetLab-wide, can have a Sirius reservation at a time. By using Sirius, we reduce the potential for PlanetLab host artifacts and get a better sense of Flexlab's accuracy.

We repeated the previous experiment fifteen minutes later, with the sole difference that the PlanetLab BT used Sirius. We ran BT on Flexlab at the same time; its measurement agent on PlanetLab did not have the benefit of Sirius. Figure 9 shows the download rates of these simultaneous runs. Sirius more than doubled the PlanetLab download rate of our previous PlanetLab experiment, from 2.08 to 5.80 Mbps. This demonstrates that BT is highly sensitive to CPU availability, and that the CPU typically available on PlanetLab is insufficient to produce accurate results for some complex applications. It also highlights the need for sufficient, reserved host resources on current and future network testbeds. In this run, the Flexlab and PlanetLab download rates are within 4% of each other, at 5.56 Mbps and 5.80 Mbps, respectively. These results are consistent, as shown by repeated experiments in Table 3. This indicates that Flexlab with ACIM provides a good environment for running experiments that need PlanetLab-like network conditions without host artifacts.

Resource Use. To estimate the host resources consumed by BT and the measurement agent we ran Flexlab with a "fake PlanetLab" side that ran inside Emulab. The agent took only 2.6% of the CPU, while BT took 37–76%, a factor of 14–28 higher. The agent's resident memory use was about 2.0MB, while BT used 8.4MB, a factor of 4 greater.

6.2.3 Simple Static Model

We ran BT again, this time using the simple-static model outlined in Sec. 4.1. Network conditions were those collected by Flexmon five minutes before running the BT experiment in Sec. 6.2.1, so we would hope to see a mean download rate similar to ACIM's: 6.3 Mbps.[2] We did three runs using the "cloud," "shared," and "hybrid" Dummynet configurations. We were surprised to find that the shared configuration gave the best approximation of BT's behavior on PlanetLab. The cloud configuration resulted in very high download rates (12.5 Mbps average), and the rates showed virtually no variation over time. Because six of the eight nodes used for our BT experiments are on I2, the hybrid configuration made little difference. The two GÉANT nodes now had realistic (lower) download rates, but the overall mean was still 10.7 Mbps. The shared configuration produced download rates that varied on timescales similar to those we have seen on PlanetLab and with ACIM. While the mean download rate was more accurate than the other configurations, it was 25% lower than that we would expect, at 5.1 Mbps.

This shows that the shared bottleneck models we developed for the simple models are not yet sophisticated enough to provide high fidelity emulation. The cloud configuration seems to under-estimate the effects of shared bottlenecks, and the shared configuration seems to over-estimate them, though to a lesser degree. Much more study is needed of these models and PlanetLab's network characteristics.

[2]The 6.2.1 experiment differed from this one in that the former generated traffic on PlanetLab from two simultaneous BT's, while this experiment ran only one BT at a time. This unfortunate methodological difference could explain much of the difference between ACIM and the simple cloud model, but only if the simultaneous BT's in 6.2.1 significantly affected each other. That seemed unlikely due to the high degree of stat muxing we expect on I2 (and probably GÉANT) paths, both apriori and from the results in Sec. 4.3. However, that assumption needs study.

7 Related Work

Network measurement to understand and model network behavior is a popular research area. There is an enormous amount of related work on measuring and modeling Internet characteristics including bottleneck-link capacity, available bandwidth, packet delay and loss, topology, and more recently, network anomalies. Examples include [7, 8, 30, 17, 29, 38]. In addition to their use for evaluating protocols and applications, network measurements and models are used for maintaining overlays [2] and even for offering an "underlay" service [22]. PlanetLab has attracted many measurement studies specific to it [31, 19, 40, 25]. Earlier, Zhang et al. [41] showed that there is significant stationarity of Internet path properties, but argued that this alone does not mean that the latency characteristics important to a particular application can be sufficiently modeled with a stationary model.

Monkey [6] collects live TCP traces near servers, to faithfully replay client workload. It infers some network characteristics. However, Monkey is tied to a web server environment, and does not easily generalize to arbitrary TCP applications. Jaisal et al. did passive inference of TCP connection characteristics [15], but focused on other goals, including distinguishing between TCP implementations.

Trace-Based Mobile Network Emulation [24] has similarities to our work, in that it used traces from mobile wireless devices to develop models to control a synthetic networking environment. However, it emphasizes production of a parameterized model, and was intended to collect application-independent data for specific paths taken by mobile wireless nodes. In contrast, we concentrate on measuring ongoing Internet conditions, and our key model

is application-centric.

Overlay Networks. Our ACIM approach can be viewed as a highly unusual sort of overlay network. In contrast to typical overlays designed to provide resilient or optimized services, our goal is to provide realism—to *expose* rather than mitigate the effects of the Internet. A significant practical goal of our project is to provide an experimentation platform for the development and evaluation of "traditional" overlay networks and services. By providing an environment that emulates real-world conditions, we enable the study of new overlay technologies designed to deal with the challenges of production networks.

Although our aims differ from those of typical overlay networks, we share a common need for measurement. Recent projects have explored the provision of common measurement and other services to support overlay networks [21, 22, 16, 26]. These are exactly the types of models and measurement services that our new testbed is designed to accept.

Finally, both VINI [4] and Flexlab claim "realism" and "control" as primary goals, but their kinds of realism and control are almost entirely different. The realism in VINI is that it peers with real ISPs so it can potentially carry real end-user traffic. The control in VINI is experimenter-controlled routing, forwarding, and fault injection, and provision of some dedicated links. In contrast, the realism in Flexlab is real, variable Internet conditions and dedicated hosts. The control in Flexlab is over pluggable network models, the entire hardware and software of the hosts, and rich experiment control.

8 Conclusion

Flexlab is a new experimental environment that provides a flexible combination of network model, realism, and control, and offers the potential for a friendly development and debugging environment. Significant work remains before Flexlab is a truly friendly environment, since it has to cope with the vagaries of a wide-area and overloaded system, PlanetLab. Challenging work also remains to extensively validate and likely refine application-centric Internet modeling, especially UDP.

Our results show that an end-to-end model, ACIM, achieves high fidelity. In contrast, simple models that exploit only a small amount of topology information (commodity Internet vs. Internet2) seem insufficient to produce an accurate emulation. That presents an opportunity to apply current and future network tomography techniques. When combined with data, models, and tools from the vibrant measurement and modeling community, we believe Flexlab with new models, not just ACIM, will be of great use to researchers in networking and distributed systems.

Acknowledgments: We are grateful to our co-workers for much implementation, evaluation, operations, discussion, design, and some writing: Sachin Goyal, David Johnson, Tim Stack, Kirk Webb, Eric Eide, Vaibhave Agarwal, Russ Fish, Leigh Stoller, and Venkat Chakravarthy; to our shepherd Srini Seshan, the reviewers, and Ken Yocum for their many useful comments, to Dave Andersen and Nick Feamster for the Datapository, to Dave for helpful discussion, to David Eisenstat and the PlanetLab team for their help and cooperation, to Vivek Pai and KyoungSoo Park for offering access to CoDeen measurements, to Jane-Ellen Long for her patience, and to NSF for its support under grants CNS–0335296, CNS–0205702, and CNS–0338785.

References

[1] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *ACM SIGOPS OSR*, 40(1):33–40, Jan. 2006.

[2] D. Andersen et al. Resilient Overlay Networks. In *Proc. SOSP*, pages 131–145, Mar. 2001.

[3] D. G. Andersen and N. Feamster. Challenges and Opportunities in Internet Data Mining. Technical Report CMU–PDL–06–102, CMU Parallel Data Laboratory, Jan. 2006.

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

[5] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. SIGCOMM*, pages 24–35, Aug.–Sept. 1994.

[6] Y.-C. Cheng et al. Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. In *Proc. USENIX*, pages 87–98, June–July 2004.

[7] M. Coates, A. O. Hero III, R. Nowak, and B. Yu. Internet Tomography. *IEEE Signal Processing Mag.*, 19(3):47–65, May 2002.

[8] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proc. SIGCOMM*, pages 15–26, Aug.–Sept. 2004.

[9] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera, and J. Lepreau. Flexlab: A Realistic, Controlled, and Friendly Environment for Evaluating Networked Systems. In *Proc. HotNets V*, pages 103–108, Nov. 2006.

[10] E. Eide, L. Stoller, and J. Lepreau. An Experimentation Workbench for Replayable Networking Research. In *Proc. NSDI*, Apr. 2007.

[11] S. Floyd and E. Kohler. Internet Research Needs Better Models. *ACM SIGCOMM CCR (Proc. HotNets-I)*, 33(1):29–34, Jan. 2003.

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

[13] P. Francis, S. Jamin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM TON*, 9(5):525–540, Oct. 2001.

[14] M. Jain and C. Dovrolis. Ten Fallacies and Pitfalls on End-to-End Available Bandwidth Estimation. In *Proc. Conf. on Internet Measurement (IMC)*, pages 272–277, Oct. 2004.

[15] S. Jaiswal et al. Inferring TCP Connection Characteristics through Passive Measurements. In *Proc. INFOCOM*, pages 1582–1592, Mar. 2004.

[16] B. Krishnamurthy, H. V. Madhyastha, and O. Spatscheck. ATMEN: A Triggered Network Measurement Infrastructure. In *Proc. WWW*, pages 499–509, May 2005.

[17] A. Lakhina, M. Crovella, and C. Diot. Mining Anomalies Using Traffic Feature Distributions. In *Proc. SIGCOMM*, pages 217–228, Aug. 2005.

[18] T. V. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss. *IEEE/ACM TON*, 5(3):336–350, 1997.

[19] S.-J. Lee et al. Measuring Bandwidth Between PlanetLab Nodes. In *Proc. PAM*, pages 292–305, Mar.–Apr. 2005.

[20] X. Liu and A. Chien. Realistic Large-Scale Online Network Simulation. In *Proc. Supercomputing*, Nov. 2004.

[21] H. V. Madhyastha et al. iPlane: An Information Plane for Distributed Services. In *Proc. OSDI*, pages 367–380, Nov. 2006.

[22] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *Proc. SIGCOMM*, pages 11–18, Aug. 2003.

[23] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proc. INFOCOM*, pages 170–179, June 2002.
[24] B. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-Based Mobile Network Emulation. In *Proc. SIGCOMM*, pages 51–61, Sept. 1997.
[25] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. Service Placement in a Shared Wide-Area Platform. In *Proc. USENIX*, pages 273–288, May–June 2006.
[26] K. Park and V. Pai. CoMon: A Mostly-Scalable Monitoring System for PlanetLab. *ACM SIGOPS OSR*, 40(1):65–74, Jan. 2006.
[27] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. *ACM SIGCOMM CCR (Proc. HotNets-I)*, 33(1):59–64, Jan. 2003.
[28] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM CCR*, 27(1):31–41, Jan. 1997.
[29] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving Accuracy in End-to-end Packet Loss Measurement. In *Proc. SIGCOMM*, pages 157–168, Aug. 2005.
[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. SIGCOMM*, pages 133–145, Aug. 2002.
[31] N. Spring, L. Peterson, V. Pai, and A. Bavier. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. *ACM SIGOPS OSR*, 40(1):17–24, Jan. 2006.
[32] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A Public Internet Measurement Facility. In *Proc. of USENIX USITS*, 2003.
[33] W. A. Taylor. Change-Point Analysis: A Powerful New Tool for Detecting Changes. http://www.variation.com/cpa/tech/changepoint.-html, Feb. 2000.
[34] A. Vahdat et al. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. OSDI*, pages 271–284, Dec. 2002.
[35] A. Vahdat, L. Peterson, and T. Anderson. Public statements at PlanetLab workshops, 2004–2005.
[36] K. Webb, M. Hibler, R. Ricci, A. Clements, and J. Lepreau. Implementing the Emulab-PlanetLab Portal: Experience and Lessons Learned. In *Proc. WORLDS*, Dec. 2004.
[37] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.
[38] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applications. In *Proc. SIGCOMM*, pages 169–180, Aug. 2005.
[39] P. Yalagandula, P. Sharma, S. Banerjee, S.-J. Lee, and S. Basu. S3: A Scalable Sensing Service for Monitoring Large Networked Systems. In *Proc. SIGCOMM Workshop on Internet Network Mgmt. (INM)*, pages 71–76, Sept. 2006.
[40] M. Zhang et al. PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-Area Services. In *Proc. OSDI*, pages 167–182, Dec. 2004.
[41] Y. Zhang, N. Du, V. Paxson, and S. Shenker. On the Constancy of Internet Path Properties. In *Proc. SIGCOMM Internet Meas. Workshop (IMW)*, pages 197–211, Nov. 2001.

A Scheduling Accuracy

To quantify the jitter and delay in process scheduling on PlanetLab nodes, we implemented a test program that schedules a sleep with the `nanosleep()` system call, and measures the actual sleep time using `gettimeofday()`. We ran this test on three separate PlanetLab nodes with load averages of roughly 6, 15, and 27, plus an unloaded Emulab node running a PlanetLab-equivalent OS. 250,000 sleep events were continuously performed on each node with a target latency of 8 ms, for a total of about 40 minutes.

Figure 10 shows the CDF of the unexpected additional

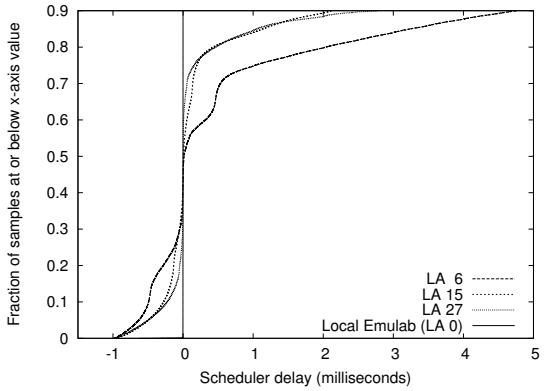

Figure 10: 90th percentile scheduling time difference CDF. The vertical line is "Local Emulab."

delay, up to the 90th percentile; Figure 11 displays the tail in log-log format. 90% of the events are within -1–5 scheduler quanta (msecs) of the target time. However, a significant tail extends to several hundred milliseconds. We also ran a one week survey of 330 nodes that showed the above samples to be representative.

This scheduling tail poses problems for the fidelity of programs that are time-sensitive. Many programs may still be able to obtain accurate results, but it is difficult to determine in advance which those are.

Spring et al. [31] also studied availability of CPU on PlanetLab, but measured it in aggregate instead of our timeliness-oriented measurement. That difference caused them to conclude that "PlanetLab has sufficient CPU capacity." They did document significant scheduling jitter in packet sends, but were concerned only with its impact on network measurment techniques. Our BT results strongly suggest that PlanetLab scheduling latency can greatly impact normal applications.

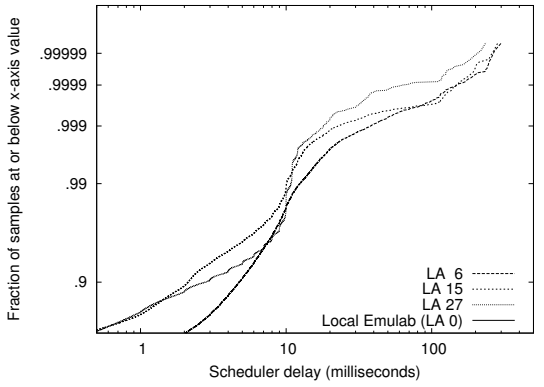

Figure 11: Log-log scale scheduling time difference CDF showing distribution tail. The "Local Emulab" line is vertical at $x = 0$.