Appeared in *Proceedings of the Sixth USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)*, pages 199–212, Boston, MA, Apr. 2009.

Modeling and Emulation of Internet Paths

Pramod Sanaga Jonathon Duerig Robert Ricci Jay Lepreau

University of Utah, School of Computing
{*pramod, duerig, ricci, lepreau*}*@cs.utah.edu*

Abstract

Network emulation subjects real applications and protocols to controlled network conditions. Most existing network emulators are fundamentally *link* emulators, not *path* emulators: they concentrate on faithful emulation of the transmission and queuing behavior of individual network hops in isolation, rather than a path as a whole. This presents an obstacle to constructing emulations of observed Internet paths, for which detailed parameters are difficult or impossible to obtain on a hop-by-hop basis. For many experiments, however, the experimenter's primary concern is the end-to-end behavior of paths, not the details of queues in the interior of the network.

End-to-end measurements of many networks, including the Internet, are readily available and potentially provide a good data source from which to construct realistic emulations. Directly using such measurements to drive a link emulator, however, exposes a fundamental disconnect: link emulators model the *capacity* of resources such as link bandwidth and router queues, but when reproducing Internet paths, we generally wish to emulate the measured *availability* of these resources.

In this paper, we identify a set of four principles for emulating entire paths. We use these principles to design and implement a path emulator. All parameters to our model can be measured or derived from end-to-end observations of the Internet. We demonstrate our emulator's ability to accurately recreate conditions observed on Internet paths.

1 Introduction

In network emulation, a real application or protocol, running on real devices, is subjected to artificially induced network conditions. This gives experimenters the opportunity to develop, debug, and evaluate networked systems in an environment that is more representative of the Internet than a LAN, yet more controlled and predictable than running live across deployed networks such as the Internet. Due to these properties, network emulation has become a popular tool in the networking and distributed systems communities.

Network emulators work by forwarding packets from an application under test through a set of queues that approximate the behavior of router queues. By adjusting the parameters of these queues, an experimenter can control the emulated capacity of a link, delay packets, and introduce packet loss. Popular network emulators include Dummynet [22], ModelNet [27], NIST Net [7], and Emulab (which uses Dummynet) [32]. These emulators focus on *link emulation*, meaning that they concentrate on faithful emulation of individual links and queues.

In many cases, particularly in distributed systems, the system under test runs on hosts at the edges of the network. Experiments on these systems are concerned with the end-to-end characteristics of the *paths* between hosts, not with the behavior of individual queues in the network. For such experiments, detailed modeling of individual queues is not a necessity, so long as end-to-end properties are preserved. One way to create emulations with realistic conditions is to use parameters from real networks, such as the Internet, but it can be difficult or impossible to obtain the necessary level of detail to recreate real networks on a hop-by-hop basis. Thus, in order to run experiments using conditions from real networks, there is a clear need for a new type of emulator that models paths as a whole rather than individual queues.

In this paper, we identify a set of principles for path emulation and present the design and implementation of a new path emulator. This emulator uses an abstract and straightforward model of path behavior. Rather than requiring parameters for each hop in the path, it uses a much smaller set of parameters to describe the entire path. The parameters for our model can be estimated or derived from end-to-end measurements of Internet paths. In addition to the simplicity and efficiency benefits, this end-to-end focus makes our emulator suitable for recreating observed Internet paths inside a network testbed, such as Emulab, where experiments are predictable, repeatable, and controlled.

1.1 Path Emulation Approaches

One approach to emulating paths is to use multiple instances of a link emulator, creating a series of queues for

the traffic under test to pass through, much like the series of routers it would pass through on a real path. ModelNet and Emulab in particular are designed for use in this fashion. Building a path emulator in this way, however, requires a router-level topology. While such topologies can be generated from models or obtained for particular networks, obtaining detailed topologies for arbitrary Internet paths is very difficult. Worse, to construct an accurate emulation, capacity, queue size, and background traffic for each link in the path must be known, making reconstruction of Internet paths intractable.

Another alternative is to approximate a path as a single link, using the desired end-to-end characteristics such as available bandwidth, and observed round-trip time, to set the parameters of a single link emulator. Because these properties can be measured from the edges of the network, this is an attractive approach. A recent survey of the distributed systems literature [29] shows that many distributed or network systems papers published in top venues [4, 5, 8, 18, 19, 23, 26, 30]—nearly one third of those surveyed—include a topology in which a single hop is used to approximate a path.

On the surface, this seems like a reasonable approximation: distributed systems tend to be sensitive to high-level network characteristics such as bandwidth, latency, and packet loss rather than the fine-grained queuing behavior of every router along a path. However, as we discuss in Section 2 and demonstrate in Section 4, using a single link emulator to model a measured path can often fail even simple tests of accuracy. This is due to a fundamental mismatch between the fact that link emulators model the *capacity* of links, and the fact that end-to-end measurements reveal the *availability* of resources on those links. This difference can result in flows being unable to achieve the bandwidth set by the experimenter or seeing unrealistic round-trip times, and these errors can be quite large. This model also does not capture interactions *between* paths, such as shared bottlenecks, or *within* paths, such as the reactivity of background flows.

1.2 Path Emulation Principles

What is needed is a new approach to emulation that models entire Internet paths rather than individual links within those paths. We have identified four principles for designing such an emulator:

- **Model capacity and available bandwidth separately.** Existing link emulators model links with limited *capacity*. We show why this is not always sufficient to create a path emulation with a particular target *available bandwidth*. We provide the mathematical basis for deciding how much capacity and how much cross-traffic are necessary to produce the desired effect.

- **Pick appropriate queue sizes.** Much work has been done in choosing "good" values for queue sizes in real routers, but the issues that apply to emulation are somewhat different. We define concrete upper and lower bounds for queue sizes in emulation and simulation. These bounds are derived from the delay and available bandwidth parameters of the emulated paths to ensure that the configured bandwidth is actually achievable.

- **Use an abstracted model of the reactivity of background flows.** Real networks have cross-traffic that reacts in complex ways to foreground traffic. Available bandwidth can change in reaction to foreground flows, and thus is a function of the load offered by the system under test. Discovering the characteristics of background traffic from the edge of the network is very difficult—even the degree of statistical multiplexing is obscured by TCP unfairness in the presence of disparate RTTs [15]. We show that we can model reactivity by concentrating only on the effect that the reactivity of the background flows has on foreground flows.

- **Model shared bottlenecks.** When modeling a set of paths, it is likely that some of those paths share bottlenecks, and that this will affect the properties seen by foreground flows. Such bottleneck sharing occurs naturally in router-level emulation, but must be explicitly modeled in an abstracted emulation.

Note that any of these principles can, individually, be applied to a link emulator; indeed, our path emulator implementation, presented in Section 3, is based on the Dummynet link emulator. Our contribution lies in identifying all four principles as being fundamental to path emulation, and in implementing a path emulator based on them so that they can be empirically evaluated. Although our focus in this paper is on emulation, these principles are also applicable to simulation.

2 Path Modeling

Our path model grows out of these four principles. It takes as input a set of five parameters: base round-trip time (RTT), available bandwidth (ABW), capacity, shared bottlenecks, and functions describing the reactivity of background traffic. As shown in Section 3.3, it is possible to measure each of these parameters from end hosts on the Internet, making it feasible to build reconstructions of real paths. We discuss the ways in which these parameters are interrelated, and contrast our model with the approach of using end-to-end measurements as input to a single link emulator, showing the deficiencies of such an approach and how our model corrects them.

Our model focuses on accommodating foreground TCP flows, leaving emulation for other types of foreground flows as future work. We also concentrate on emulating stationary conditions for paths; in principle, any or all parameters to our model can be made time-varying to capture more dynamic network behavior.

2.1 Base RTT

The round-trip time (RTT) of a path is the time it takes for a packet to be transferred in one direction plus the time for an acknowledgment to be transferred in the opposite direction. We model the RTT of a path by breaking it into two components: the "base RTT" [6] (RTT_{base}) and the queuing delay of the bottleneck link.

The base RTT includes the propagation, transmission, and processing delay for the entire path and the queuing delay of all non-bottleneck links. When the queue on the bottleneck link is empty, the RTT of the path is simply the base RTT. In practice, the minimum RTT seen on a path is a good approximation of its base RTT. Because transmission and propagation delays are constant, and processing delays for an individual flow tend to be stable, a period of low RTT indicates a period of little or no queuing delay.

The base RTT represents the portion of delay that is relatively insensitive to network load offered by the foreground flows. This means that we do not need to emulate these network delays on a detailed hop-by-hop basis: a fixed delay for each path is sufficient.

2.2 Capacity, Available Bandwidth, and Queuing

The bottleneck link controls the bandwidth available on the path, contributes queuing delay to the RTT, and causes packet loss when its queue fills. Thus, three properties of this link are closely intertwined: link capacity, available bandwidth, and queue size.

We make the common assumption that there is only one bottleneck link on a path in a given direction [9] at a given time, though we do not assume that the same link is the bottleneck in both directions.

2.2.1 Capacity and Available Bandwidth

Existing link emulators fundamentally emulate limited *capacity* on links. The link speed given to the emulator is used to determine the rate at which packets drain from the emulator's bandwidth queue, in the same way that a router's queue empties at a rate governed by the capacity of the outgoing link. The quantity that more directly affects distributed applications, however, is *available bandwidth*, which we consider to be the maximum rate sustainable by a foreground TCP flow. This is the rate at which the foreground flow's packets empty from the bottleneck queue. Assuming the existence of competing traffic, this rate is lower than the link's capacity.

It is not enough to emulate available bandwidth using a capacity mechanism. Suppose that we set the capacity of a link emulator using the available bandwidth measured on some Internet path: inside of the emulator, packets will drain more slowly than they do in the real world. This difference in rate can result in vastly different queuing delays, which is not only disastrous for latency-sensitive experiments, but as we will show, can cause inaccurate bandwidth in the emulator as well.

Let q_f and q_r be the sizes of the bottleneck queues in the forward and reverse directions, respectively, and let C_f and C_r be the capacities. The maximum time a packet may spend in a queue is $\frac{q}{C}$, giving us a maximum RTT that can be observed on the path:

$$RTT_{max} = RTT_{base} + \frac{q_f}{C_f} + \frac{q_r}{C_r} \quad (1)$$

If we were to use ABW_f and ABW_r—the available bandwidth measured from some real Internet path—to set C_f and C_r, Equation 1 would yield much larger queuing delays within the emulator than seen on the real path (assuming the queues sizes on the path and in the emulator are the same).

For instance, consider a real path with $RTT_{base} = 50$ ms, a bottleneck of symmetric capacity $C_f = C_r = 43$ Mbps (a T-3 link) and available bandwidth $ABW_f = ABW_r = 4.3$ Mbps. For a small q_f and q_r of 64 KB (fillable by a single TCP flow), the RTT on the path is bounded at 74 ms, since the forward and reverse directions each contribute at most 12 ms of queuing delay. However, if we set $C_f = C_r = 4.3$ Mbps within an emulator (keeping queue sizes the same), each direction of the path can contribute up 120 ms of queuing delay. The total resulting RTT could reach as high as 290 ms.

This unrealistically high RTT can lead to two problems. First, it fails to accurately emulate the RTT of the real path, causing problems for latency-sensitive applications. Second, it can also affect the bandwidth available to TCP, a problem we discuss in more detail in Section 2.2.2.

One approach reducing the maximum queuing delay would be to simply reduce the q_f and q_r inside of the emulator. This may result in queues that are simply too small. In the example above, to reduce the queuing delay within the path emulator to the same level as the Internet path, we would we would have to reduce the queue size by a factor of 10 to 6.4 KB. A queue this small will cause packet loss if a stream sends a small burst of traffic, preventing TCP from achieving the requested available bandwidth. We also discuss minimum queue size in more detail in Section 2.2.2.

The solution to these queuing problems is to separate the notions of *capacity* and *available bandwidth* in our path emulation model: they are independent parameters to each path. When we wish to emulate a path with competing traffic at the bottleneck, we set $C > ABW$. To model links with no background traffic, we can still set $C = ABW$, as is done implicitly in a link emulator.

Of course, when $C > ABW$, we must fill the excess capacity to limit foreground flows to the desired ABW. A common solution to this problem has been to add a number of background TCP flows to the bottleneck. The problem with this technique is one of measurement. When the emulation is constructed using end-to-end observations of a real path, discovering the precise behavior or even the number of competing background flows is not possible from the edges of the network. Adding reactive background flows to our emulation would not mirror the reactivity on the real network, and would result in an inexact ABW in the emulator.

Since there is not enough information to replicate the background traffic at the bottleneck, we separately emulate its *rate* and its *reactivity*. We can precisely emulate a particular level of background traffic using non-responsive, constant-bit-rate traffic. This mechanism allows us to provide an independent mechanism for emulating reactivity, described in Section 2.3. The reactivity model can change the level of background traffic to emulate responsiveness while providing a precise available bandwidth to the application at every point in time.

2.2.2 Queue Size

Much work has been done in choosing appropriate values for queue sizes in real routers [1], but the set of constraints for emulation are somewhat different: we have a relatively small set of foreground flows and a specific target ABW that we wish to achieve. Although queue sizes can be provided directly as parameters to our model, we typically calculate them from other parameters. We do this for two reasons. First, it is difficult to measure the bottleneck queue size from the endpoints of the network due to interference from cross-traffic. Second, the bottleneck queue size affects applications only through additional latency or reduced bandwidth it might cause. Because our primary concern is emulating application-visible effects, we include a method for selecting a queue size that enables accurate emulation of those effects.

We look at queue sizes in two ways: in terms of *space* (their capacity in bytes or packets) and in terms of *time* (the maximum queuing delay they may induce). This leads to two constraints on queue size:

- The queue must be large enough in space that a TCP stream is able to get the full desired ABW; it should not drop bursts of packets.
- The queue must not be so large in time that the queuing delay from a full queue causes excessive RTTs, as seen in Equation 1.

Lower bound. Finding the lower bound is straightforward. Current best practices suggest that for a small number of flows, a good lower bound on queue size is the sum of the bandwidth-delay products of all flows traversing that link [1]. Here, a "small number" of flows is fewer than about 500. Because we are concerned only with flows of a foreground application, the number of flows *on a specific path* will typically be much smaller than this. For a TCP flow f, the window size w_f is roughly equal to its bandwidth-delay product, and is capped by w_{max}, the maximum window size allowed by the TCP implementation. Thus, for a given path in a given direction, we sum over the set of flows F, giving us a lower bound on q:

$$q \geq \sum_{f \in F} min(w_f, w_{max}) \tag{2}$$

This bound applies to the queues in both directions on the path, q_f and q_r. Intuitively, the queue must be large enough to hold at least one window's worth of packets for each flow traversing the queue.

Upper bound. The upper bound is more complex. The maximum RTT tolerable for a given flow on a given path, before it becomes window-limited, is given by (using the empirically derived TCP performance model demonstrated by Padhye et al. [16]):

$$RTT_{max} = \frac{w_{max}}{ABW} \tag{3}$$

where ABW is the available bandwidth we wish the flow to experience. If the RTT grows above this limit, the bandwidth-delay product exceeds the maximum window size w_{max}, and the flow's bandwidth will be limited by TCP itself, rather than the ABW we have set in the emulator. Since our goal is to accurately emulate the given ABW, this would result in an incorrect emulation.

It is important to note that a single flow along a path cannot cause itself to become window-limited, as it will either fill up the bottleneck queue before it reaches w_{max}, or stabilize on an average queue occupancy no larger than w_{max}. Two or more flows, however, can induce this behavior in each other by filling a queue to a greater depth than can be sustained by either one. Even flows crossing a bottleneck in opposite directions can cause excessive RTTs, as each flow's ACK packets must wait in a queue with the other flow's data packets. The value of w_{max} may be defined by several factors, including limitations of the TCP header and configuration options in the TCP stack, but is essentially known and fixed for a given experiment.

Flows may travel in both directions along a path, and while both will see the same RTT, they may have different RTT_{max} values if the ABW on the path is not symmetric. Without loss of generality, we define the "forward" direction of the path to be the one with the higher ABW. From Equation 3, flows in this direction have the smaller RTT_{max}, and since we do not want either flow to become window-limited, we use ABW_f to find the upper bound.

Because most (Reno-derived) TCP stacks tend to reach a steady state in which the bottleneck queue is full [16], bottleneck queues tend to be nearly full, on average. Thus, we can expect flows to experience RTTs near the maximum given by Equation 1 in steady-state operation. For our emulation of ABW to be accurate, then, Equation 1 (the maximum observable RTT) must be less than or equal to Equation 3 (the maximum tolerable RTT). If we set the two capacities to be equal and solve for the queue sizes, this gives us:

$$q_f + q_r \leq C \cdot \left(\frac{w_{max}}{ABW_f} - RTT_{base} \right) \quad (4)$$

Because all terms on the right side are either fixed or parameters of the path, we have a bound on the total queue size for the path. (It is not necessary for the forward and reverse capacities to be equal to solve the equation. We do so here for simplicity and clarity.)

Setting the Queue Size. To select sizes for the queues on a path, we must simply split the total upper bound in Equation 4 between the two directions, in such a way that neither violates Equation 2.

These two bounds have a very important property: it is not necessarily possible to satisfy both when $C = ABW$. When either bound is not met, the emulation will not provide the desired network characteristics. The capacity C acts as a scaling factor on the upper bound. By adjusting it while holding ABW constant, we can raise or lower the maximum allowable queue size, making it possible to satisfy both equations.

Figure 1 illustrates this principle by showing valid queue sizes as a function of capacity. As capacity changes, the upper bound increases while the lower bound remains constant. When capacity is at or near available bandwidth, the upper bound is *below* the lower bound, which means that no viable queue size can be selected. As capacity increases, these lines intersect and yield an expanding region of queue sizes that fulfill both constraints. This underscores the importance of emulating available bandwidth and capacity separately.

Asymmetry. Throughput artifacts due to violations of Equation 3 are exacerbated when traffic on the path is bidirectional and the available bandwidth is asymmetric. In this case, the flows in each direction can tolerate different maximum RTTs, with the flow in the forward (higher ABW) direction having the smaller upper bound.

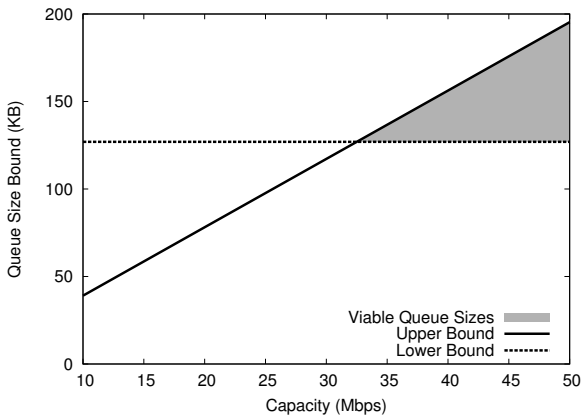


Figure 1: The relationship between capacity and the bounds on queue size for a path with $ABW_f = ABW_r = 10$ Mbps, $RTT_{base} = 20$ ms, and $w_{max} = 65$ KB. Low capacities prevent any viable queue size.

This means that it is disproportionately affected by high RTTs. Others have described this phenomenon [2], and we demonstrate it empirically in Section 4.1.

To determine how common paths with asymmetric ABW are in practice, we measured the available bandwidth on 7,939 paths between PlanetLab [17] nodes. Of those paths, 30% had greater ABW in one direction than the other by a ratio of at least 2:1, and 8% had a ratio of at least 10:1. Because links with asymmetric capacities (e.g., DSL and cable modems) are most common as last-mile links, and because PlanetLab has few nodes at such sites, it is highly likely that most of this asymmetry is a result of bottlenecks carrying asymmetric traffic. Our experiments in Section 4 shows that on a path with an available bandwidth asymmetry ratio as small as 1.5:1, a simple link emulation model that does not separate capacity and ABW, and does not set queue sizes carefully, can result in a 30% error in achieved throughput.

2.2.3 Putting It Together

Figure 2 shows an overview of our model as described thus far. We model the bottleneck of a path with a queue that drains at a fixed rate, and a constant bit-rate cross-traffic source. The rate at which the queue drains is the capacity, and the difference between the injection rate of the cross-traffic and the capacity is the available bandwidth. The remainder of the delay on the path is modeled by delaying packets for a constant amount of time governed by RTT_{base}. The two halves of the path are modeled independently to allow for asymmetric paths.

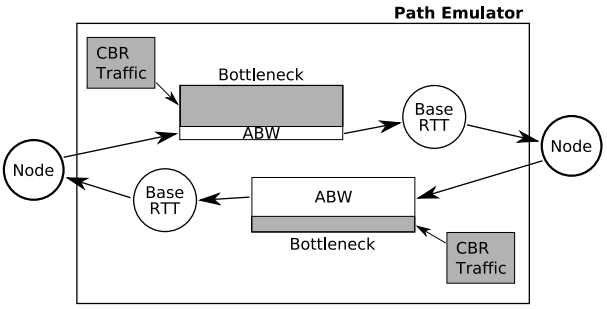


Figure 2: Modeling a single path, in both the forward and reverse directions.

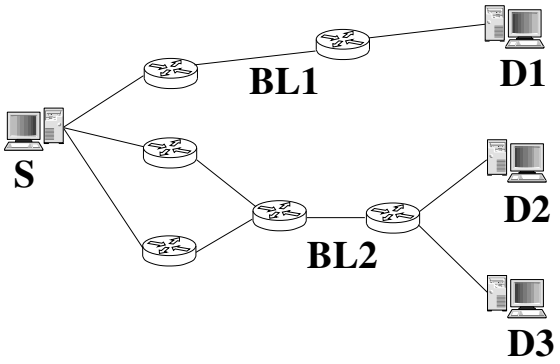


Figure 3: A router-level topology, showing two bottleneck links. One (*BL2*) is shared by two paths from source *S*: the paths to destinations *D2* and *D3*.

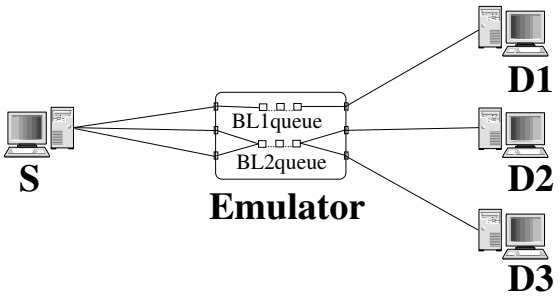


Figure 4: An abstracted view of Figure 3, with the bottleneck links represented as bottleneck queues.

2.3 Interactions Between Flows

In addition to emulating the behavior of the foreground flows' packets in the bottleneck queue, we must also emulate two important interactions: the interaction of multiple foreground flows on different paths that share bottlenecks, and the interaction of foreground flows with responsive background traffic.

Shared Bottlenecks. To properly emulate sets of paths, we must take into account bottlenecks that are shared by more than one path. Consider the simple case in Figure 3. If we do not model the bottleneck *BL2* (shared by the paths from source *S* to destinations *D2* and *D3*), we will allow multiple paths to independently use bandwidth that should be shared between them. Doing so could result in the application getting significantly more bandwidth within the emulator than it would on the real paths [21].

We do not, however, need to know the full router-level topology of a set of paths in order to know that they share bottlenecks. Existing techniques [12] can detect the existence of such bottlenecks from the edges of the network, by correlating the observed timings of simultaneous packet transmissions on the paths.

To model paths that share a bottleneck, we abstract shared bottlenecks in a simple manner: instead of giving each path an independent bandwidth queue, we allow multiple paths to share the same queue. Traffic leaving a node is placed into the appropriate queue based on which destinations, if any, share bottlenecks from that source.

This is illustrated in Figure 4: the two bottleneck links in the original topology are represented as bottleneck queues inside the path emulator. While paths sharing a bottleneck link share a bottleneck queue, each still has its own base RTT applied separately. Because base RTT represents links in the path other than the bottleneck link, links with a shared bottleneck do not necessarily have the same RTT. With this model, it is also possible for a path to pass through a different shared bottleneck in each direction.

Reactivity of Background Traffic. Flows traversing real Internet paths interact with cross-traffic, and this cross-traffic typically has some reactivity to the foreground flows. Thus, ABW on a path is not constant, even under the assumption that the set of background flows does not change. Simply setting a static ABW for a path can miss important effects: if more than one flow is sent along the path, the aggregate ABW available to all foreground flows may be greater, as the background traffic backs off further in reaction to the increased load. This is particularly important when the bottleneck is shared between two or more paths; the load on the bottleneck is the sum of the load on all paths that pass through it.

While it is possible to create reactivity in the emulation by sending real, reactive cross-traffic (such as competing TCP flows) across the bottlenecks, doing so in a way that faithfully reproduces conditions on an observed link is problematic. The number, size, and RTT of these background flows all affect their reactivity, and such detail is not easily observed from endpoints. We turn to our guiding principle of abstraction, and model the *reactivity* of the background traffic to our foreground flows, rather than the details of the background traffic itself.

We look at *ABW* as a function of offered load: $ABW_d(L_d)$ gives the aggregate bandwidth available in direction d (forward or reverse) of a given path, as a

Name	Type	Description
C_f, C_r	fixed	Capacity of the bottleneck in the forward and reverse directions. Fixed to value sufficient to make satisfaction of queue bounds possible for most experiments.
$ABW_f(\|F_f\|)$, $ABW_r(\|F_r\|)$	measured	Table giving available bandwidth for the forward and reverse directions, as a function of the number of flows traversing the path in that direction.
RTT_{base}	measured	Base RTT of the path, split evenly between the two directions.
$S_{p \in P}$	measured	A subset of paths p from the set of all paths P that share a common bottleneck. Multiple instances of this parameter may be given.
q_f, q_r	derived	The queue size for each bottleneck is derived from the measured values of ABW, RTT_{base}, and the fixed capacity. If ABW is adjusted based on the reactivity table, queue size is as well.

Table 1: The parameters to our path emulation. All parameters except $S_{p \in P}$ are given on a per-path basis.

function of the offered load L_d in that direction on that path. A set of such functions, one for each direction on each path, is supplied as a parameter to the emulation. Note that this offered load—and with it the available bandwidth—will likely vary over time during the emulated experiment. The *ABW* function can be created analytically based on a model or it can be measured directly from a real path, by offering loads at different levels and observing the resulting throughput. The emulation can then provide—with high accuracy—exactly the desired ABW. Once we have used the reactivity functions to determine the aggregate bandwidth available on a path, we can set both the capacity and queue sizes as described in Section 2.2.2.

Because an *ABW* function is a parameter of a particular path, when multiple paths share a bottleneck, we must combine their functions. There are multiple ways that the *ABW* functions may be combined. Ideally, we would like to account for every possible combination of flows using every possible set of paths that share the bottleneck. The combinatorial explosion this creates, however, quickly makes this infeasible for even a modest number of paths. Instead, the simple strategy that we currently employ is to take the mean of the *ABW* values for each individual path sharing the bottleneck, weighted by the number of flows on each path. We are exploring the possibility that more complicated approaches may yield more realistic results.

3 Implementing a Path Emulator

Although the model we have discussed is applicable to both simulation and emulation, we chose to do our initial implementation in an emulator. Our prototype path emulator is implemented as a set of enhancements to the Dummynet [22] link emulator. We constructed our prototype within the Emulab network testbed [32], but it is not fundamentally linked to that platform.

3.1 Basis: The Dummynet Link Emulator

Dummynet is a popular link emulator implemented in the FreeBSD kernel. It intercepts packets coming through an incoming network interface and places them in its internal objects—called *pipes*—to emulate the effects of delay, limited bandwidth, and probabilistic random loss. Each pipe has one or more queues associated with it. Given the capacity or the delay of a pipe, Dummynet schedules packets to be emptied from the corresponding queues and places them on the outgoing interface.

Dummynet can be configured to send a packet through multiple pipes on its path from an incoming interface to an outgoing interface. One pipe may enforce the base delay of the link, and a subsequent pipe may model the capacity of the link being emulated. Dummynet uses the IPFW packet filter to direct packets into pipes, and can therefore use many different criteria to map packets to pipes.

In network emulation testbeds, "shaping nodes" are interposed on emulated links, each acting as a transparent bridge between the endpoints. In Emulab, the shaping nodes' Dummynet is configured with one or more pipes to handle traffic in each direction on the emulated link, allowing for asymmetric link characteristics. Shaping nodes can also be used in LAN topologies by placing a shaping node between each node and switch implementing the LAN. Thus traffic between any two nodes passes through two shaping nodes: one between the source and the LAN, and one between the LAN and the destination.

3.2 Enhancements for Path Emulation

To turn Dummynet into a path emulator, we made a number of enhancements to it. The parameters to the resulting path emulator are summarized in Table 1.

Capacity and Available Bandwidth. Dummynet implements bandwidth shaping in terms of a bandwidth pipe, which contains a bandwidth queue that is drained at a specified rate, modeling some capacity C. To separate

the emulation of capacity from available bandwidth, we modified Dummynet to insert "placeholder" packets into the bandwidth queues at regular, configurable intervals. These placeholder packets are neither received from nor sent to an actual network interface; their purpose is simply to adjust the rate at which foreground flows' packets move through the queue. The placeholders are sent at a constant bit rate of $C - ABW$, setting the bandwidth available to the experimenter's foreground flows. ABW can be set as a function of offered load, using the mechanism described below.

Base Delay. We leave Dummynet's mechanism for emulating the constant base delay unchanged. Packets pass through "delay" queues, where they remain for a fixed amount of time.

Queue Size. We use Equation 4 to set the queue size for the bandwidth queues in each direction of each path, dividing the number of bytes equally between the forward and reverse directions. Because the model assumes that packets are dropped almost exclusively by the bottleneck router, modeled by the bandwidth queues, the size of the delay queues is effectively infinite.

Background Traffic Reactivity. We implement the ABW_f and ABW_r functions as a set of tables that are parameters to the emulator. Each path is associated with a distinct table in each direction. We measure the offered load on a path by counting the number of foreground flows traversing that path. We do this for two reasons. First, it makes the measurement problem more tractable, allowing us to measure a relatively small, discrete set of possible offered loads on the real path. Second, our goal is to recreate inside the emulator the behavior that one would see by sending the same flows on the real network. The complex feedback system created by the interaction of foreground flows with background flows is captured most simply by measuring entire flows, as it is strongly related to TCP dynamics. It does have a downside, however, in that it makes the assumption that the foreground flows will be full-speed TCP flows. During an execution of the emulator, a traffic monitor counts the number of active foreground flows on each path, and informs the emulator which table entry to use to set the aggregate ABW for the path. This target ABW is achieved inside the emulator by adjusting the rate at which placeholder packets enter the bandwidth queue. Our implementation also readjusts bottleneck queue sizes in reaction to these changes in available bandwidth.

Shared Bottlenecks. We implement shared bottlenecks by allowing a bandwidth pipe to shape traffic to more than one destination simultaneously. For each endpoint host in the topology, the emulator takes as a parameter a set of "equivalence classes": sets of destination hosts that share a common bottleneck, and thus a common bandwidth pipe. Packets are directed into the proper bandwidth pipe using IPFW rules. Our current implementation only supports bottlenecks that share a common source. We are in the process of extending our prototype to implement other kinds of bottlenecks, such as those that share a destination.

3.3 Gathering Data from the Real World

To create and run experiments with the path emulator, we need a source of input data for the parameters shown in Table 1. Although it is possible to synthesize values for these parameters, we concentrate here on gathering them from end-to-end measurements of the Internet.

We developed a system for gathering data for these parameters using hosts in PlanetLab [17], which gives us a large number of end-site vantage points around the world. Each node in the emulation is paired with a PlanetLab node; measurements taken from the PlanetLab node are used to configure the paths to and from the emulated node.

To gather values for RTT_{base}, we use simple `ping` packets, sent frequently over long periods of time [10]. The smallest RTT seen for a path is presumed to be an event in which the probe packet encountered no significant queuing delay, and thus representative of the base RTT.

To detect shared bottlenecks from a source to a set of destinations, we make use of a wavelet-based congestion detection tool [12]. This tool sends UDP probes from a source node to all destination nodes of interest and records the variations in one-way delays experienced by the probe packets. Random noise introduced in the delays by non-bottleneck links is removed using a wavelet-based noise-removal technique. The paths are then grouped into different clusters, with all the paths from the source to the set of destinations going through the same shared bottleneck appearing in a single cluster. The shared bottlenecks found by this procedure are passed to the emulator as the $S_{p \in P}$ sets.

Our goal is that a TCP flow through the emulator should achieve the same throughput as a TCP flow sent along the real path. So, we use a definition of ABW that differs slightly from the standard one—we equate the available bandwidth on a path to the throughput achieved by a TCP flow. We also need to measure how this ABW changes in response to differing levels of foreground traffic. While we cannot observe the background traffic on the bottleneck directly, we *can* observe how different levels of foreground traffic result in different amounts of bandwidth available to that foreground traffic. Although packet-pair and packet-train [9, 13, 20] measurement tools are efficient, they do not elicit reactions from background traffic. For this reason, we use the following methodology to concurrently estimate the ABW and

reactivity of background traffic on a particular path.

To measure the reactivity of Internet cross-traffic to the foreground flows, we run a series of tests using `iperf` between each pair of PlanetLab nodes, with the number of concurrent flows ranging from one to ten. We use the values obtained from these tests between all paths of interest to build the reactivity tables for the path emulator. However, running such a test takes time: only one test can be active on each path at a time, and `iperf` must run long enough to reach a steady state. Thus, our measurements necessarily represent a large number of snapshots taken at different times, rather than a consistent snapshot taken at a single time. The cross-traffic on the bottleneck may vary significantly during this time frame. So, the reactivity numbers are an approximation of the behavior of cross-traffic at the bottleneck link. This is a general problem with measurements that must perturb the environment to differing levels. The time required to gather these measurements is also the main factor limiting the scale of our emulations.

Another problem that arises is the proper ABW value for shared bottlenecks. Because paths that share a bottleneck do not necessarily have the same RTTs, they may evoke different levels of response from reactive background traffic. It is not feasible to measure every possible combination of flows on different paths through the same shared bottleneck. Thus, we use the approximation discussed in Section 2.3 to set ABW for shared bottlenecks.

Our current implementation does not measure the bottleneck link capacities C_f and C_r on PlanetLab paths, due to the difficulty of obtaining accurate packet timings on heavily loaded PlanetLab nodes [25]. We set the capacity of all bottleneck links to 100 Mbps. In practice, we find that for $C \gg ABW$, the exact value of C makes little difference on the emulation, and thus we typically set it to a fixed value. We demonstrate this in Section 4.3.

4 Evaluation

The goal of our evaluation is to show that our path emulator accurately reproduces measurements taken from Internet paths. We demonstrate, using micro-benchmarks and a real application, that our path emulator meets this goal under conditions in which approximating the path using a single link emulator fails to do so. In the experiments described below, we concentrate on accurately reproducing TCP throughput and observed RTT.

All of our experiments were run in Emulab on PCs with 3 GHz Pentium IV processors and 2 GB of RAM. The nodes running application traffic used the Fedora Core Linux distribution with a 2.6.12 kernel, with its default BIC-TCP implementation. The link emulator was Dummynet running in the FreeBSD 5.4 kernel, and our path emulator is a set of modifications to it. All measurements of Internet paths were taken using PlanetLab hosts.

4.1 Effect on TCP Throughput

We begin by running a micro-benchmark, `iperf`, a bulk-transfer tool that simply tries to achieve as much throughput as possible using a single TCP flow.

We performed a series of experiments to compare the behavior of `iperf` when run on real Internet paths, an unmodified Dummynet link emulator, and our path emulator. We used a range of ABW and RTT values, some taken from measurements on PlanetLab and some synthetic. The ABW values from PlanetLab were measured using `iperf`, and thus the emulators' accuracy can be judged by how closely `iperf`'s performance in the emulated environment matches the ABW parameter. In the link emulator, we set the capacity to the desired ABW (as there is only one bandwidth parameter), and in the path emulator, we set capacity to 100 Mbps. The link emulator uses Dummynet's default queue size of 73 KB, and the path emulator's queue size was set using Equation 4. Reactivity tables and shared bottlenecks were not used for these experiments. We started two TCP flows simultaneously on the emulated path, one in each direction, and report the mean of five 60-second runs.

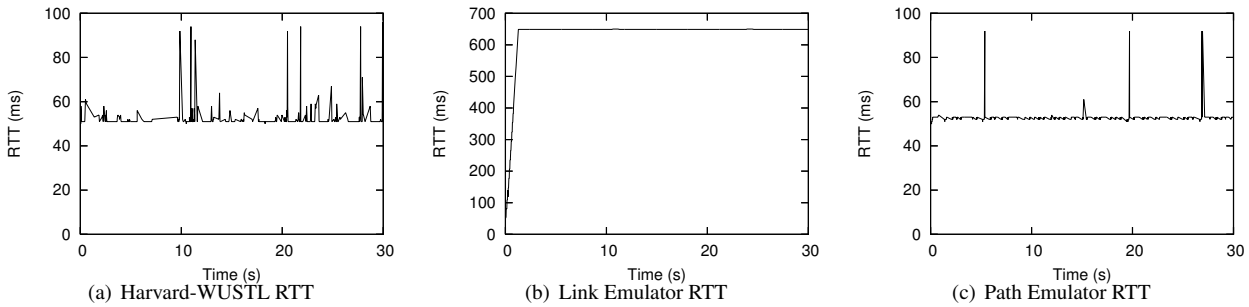
The results of these experiments are shown in Table 2. It is clear from the percent errors that the path emulation achieves higher accuracy than the link emulator in many scenarios. While both achieve within 10% of the specified throughput in the first test (a low-bandwidth, symmetric path), as path asymmetry and bandwidth-delay product increase, the effects discussed in Section 2.2 cause errors in the link emulator. While our path emulator remains within approximately 10% of the target ABW, the link emulator diverges by as much as 66%. The forward direction, with its higher throughput, tends to suffer disproportionately higher error rates. Because the measured values come from real Internet paths, they do not represent unusual or extreme conditions.

The first two rows of synthetic results demonstrate that, even in cases of symmetric bandwidth, the failure to differentiate between capacity and available bandwidth hurts the link emulator's accuracy. The third demonstrates divergence under highly asymmetric conditions.

To evaluate the importance of selecting proper queue sizes, we reran two earlier experiments in our path emulator, this time setting the queue sizes greater than the upper limits allowed by Equation 4. These results are shown in the bottommost section of Table 2 (labeled "Bad Queue Size"). The RTT for each flow grows until the flows reach their maximum window sizes, preventing them from utilizing the full ABW of the emulated path and resulting in large errors.

Test Type	Configured ABW (Kbps) Forward	Reverse	Configured Base Delay (ms)	Emulator	Queue size (KB)	Achieved Tput (Kbps) Forward	Reverse	ABW Error (%) Forward	Reverse
Measured	2,251	2,202	64	link	73	2,070	2,043	8.0	7.2
				path	957	2,202	2,163	2.1	1.8
	4,061	2,838	29	link	73	2,774	2,599	31.7	8.4
				path	957	3,822	2,706	5.8	4.6
	6,436	2,579	12	link	73	3,176	2,358	50.6	8.5
				path	844	6,169	2,448	4.1	5.0
	25,892	17,207	4	link	73	20,608	15,058	20.4	12.5
				path	197	23,237	15,644	10.2	9.1
Synthetic	8,000	8,000	45	link	73	6,228	6,207	22.0	22.4
				path	237	7,493	7,420	6.3	7.2
	12,000	12,000	30	link	73	9,419	9,398	21.5	21.6
				path	158	11,220	11,208	6.5	6.6
	10,000	3,000	30	link	73	3,349	2,705	66.5	9.8
				path	265	9,150	2,690	8.5	10.3
Bad Queue Size	25,892	17,207	4	link	—	—	—	—	—
				path	390	21,012	15,916	18.8	7.5
	10,000	3,000	30	link	—	—	—	—	—
				path	488	7,641	2,768	23.6	7.7

Table 2: Throughput achieved by simultaneous TCP flows along both directions of a number of paths, using a link emulator and using our path emulator.



(a) Harvard-WUSTL RTT (b) Link Emulator RTT (c) Path Emulator RTT

Figure 5: RTT over the lifetime of a 30-second TCP flow. Note that the range of the Y-axis in the center graph is seven times larger than the other two graphs.

4.2 Effect on Round-Trip Time

In addition to TCP throughput, our path emulator also has a significant effect on the RTT observed by a flow, producing RTTs much more similar to those on real paths than those seen in a simple link emulator. To evaluate this difference, we measured the path between the PlanetLab nodes at Harvard and those at Washington University in St. Louis (WUSTL). The ABW was 409 Kbps from Harvard to WUSTL, and 4,530 Kbps from WUSTL to Harvard. The base RTT was 50 ms. To isolate the effects of distinguishing ABW and capacity from other differences between the emulators, we set the queue size in both to the same value (our Linux kernel's maximum window size of 32 KB), and exercised only one direction of the path.

Figure 5 shows the round-trip times seen during a 30-second `iperf` run from Harvard to WUSTL, and the round-trip times seen under both link and path emulation. Both emulators achieved the target bandwidth, but dramatically differ in the round-trip times and packet-loss characteristics of the flows. Figure 5(b) and Figure 5(c) show the round-trip times observed on the link and path emulators respectively. As TCP tends to keeps the bottleneck queue full, it quickly plateaus at the length of the queue in time. Because the link emulator's queue drains at the rate of *ABW*, rather than the much larger rate of *C*, packets spend much longer in the queue in the link emulator. The average RTT for the link emulator was 629 ms, an order of magnitude higher than the average RTT of 53.1 ms observed on the actual path (Figure 5(a)). Because the path emulator separates capacity and ABW, it

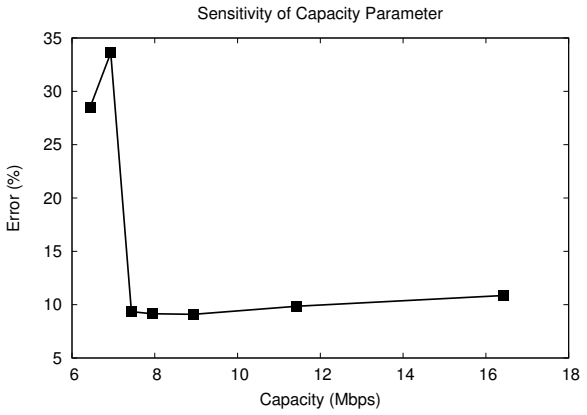


Figure 6: Experiments on an emulated path with 6.5 Mbps available bandwidth in the forward direction. A constant queue size is maintained while capacity is varied.

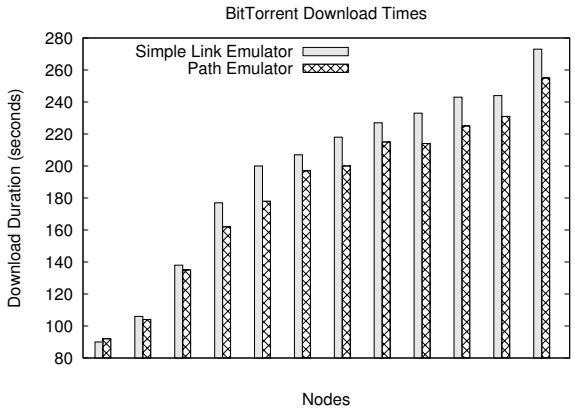


Figure 7: Time taken by participants in a BitTorrent swarm to download a file. Download times are shown for each node for both path and simple link emulation.

gives an average RTT of 53.2 ms, which is within 1% of the value on the real path. The standard deviation inside of the path emulator is 3.0 ms, somewhat lower than the 5.1 ms seen on the real path.

To get comparable RTTs from the link emulator, its queue would have to be much smaller, around 2.5 KB, which is not large enough to hold two full-size TCP packets. We reran this experiment in the link emulator using this smaller queue size, and a unidirectional TCP flow was able to achieve close to the target 409 Kbps throughput. However, when we ran bidirectional flows, the flow along the reverse direction was only able to achieve a throughput of around 200 Kbps, despite the fact that the ABW in that direction was set to 4,530 Kbps (the value measured on the real path). This demonstrates that adjusting queue size by itself is not sufficient to fix excessive RTTs, as it can cause significant errors in ABW emulation.

4.3 Sensitivity Analysis of Capacity

As we saw in Figure 1, once the capacity has grown sufficiently large, it is possible to satisfy both the upper and lower bounds on queue size. Our next experiment tests how sensitive the emulator is to capacity values larger than this intersection point.

We ran several trials with a fixed available bandwidth (6.5 Mbps) but varying levels of capacity. All other parameters were left constant. Figure 6 shows the relative error in achievable throughput as we vary the capacity. While error peaks when capacity is very near available bandwidth, outside of that range, changing the capacity has very little effect on the emulation. This justifies the decision in our implementation to use a fixed, large capacity, rather than measuring it for each path.

4.4 BitTorrent Application Results

We demonstrated in Section 4.1 that using path parameters in a simple link emulator causes artifacts in many situations. We now show that these artifacts cause inaccuracies when running real applications and are not just revealed using measurement traffic. Though this experiment uses multiple paths, to isolate the effects of capacity and queue size, it does not model shared bottlenecks or reactivity.

Figure 7 shows the download times of a group of BitTorrent clients using simple link emulation and path emulation with the same parameters, which were gathered from PlanetLab paths. Each pair of bars shows the time taken to download a fixed file on one of the twelve nodes. The simple link emulator limits available bandwidth inaccurately under some circumstances, which increases the download duration on many of the nodes. As seen in the figure, each node downloads an average of 6% slower in the link emulator than it does when under path emulator. The largest difference is 12%. This shows that the artifacts we observe with micro-benchmarks also affect the behavior of real applications.

4.5 Network Reactivity

Our next experiment examines the fidelity of our reactivity model. We ran reactivity tests on a set of thirty paths between PlanetLab nodes. For each path, we measured aggregate available bandwidth with a varying number of foreground `iperf` flows, ranging from one to eight. We used this data as input to our emulator, in the form of reactivity tables, then repeated the experiments inside of the emulator. In this experiment, the paths are tested independently at different times, so no shared bottlenecks are exercised.

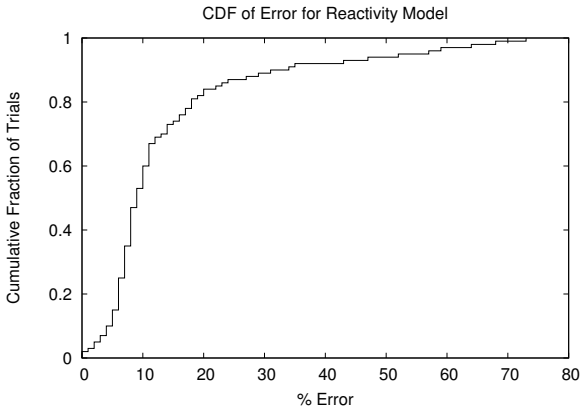


Figure 8: A CDF showing percentage of error over paths with multiple foreground flows.

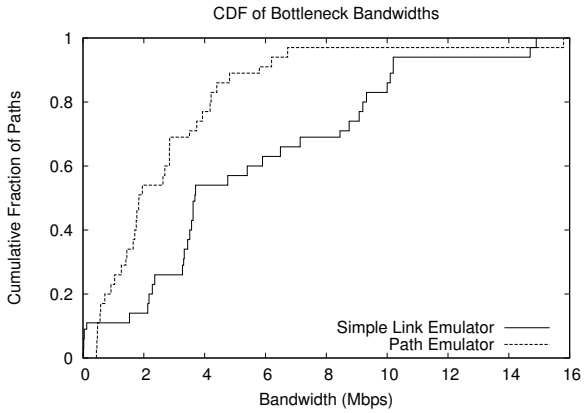


Figure 9: A CDF showing bandwidth achieved at shared bottlenecks.

By comparing the throughputs achieved inside of the emulator to those obtained on the real path, we can test the accuracy of our reactivity model. Figure 8 shows the results of this experiment. For each trial (a specific number of foreground flows over a specific path), we computed the error as the percentage difference between the aggregate bandwidth measured on PlanetLab and that recreated inside the emulator. Our emulator was quite accurate; 80% of paths were emulated to within 20% of the target bandwidth.

There are some outliers, however, with significant error. These point to limitations of our implementation, which currently sets capacities to 100 Mbps and has a 1 MB limit on the bottleneck queue size. Some paths in this experiment had very high ABW: as high as 78 Mbps in aggregate for eight foreground flows. As we saw in Figure 6, when ABW is close to capacity, significant errors can result. With high bandwidths and multiple flows, the lower bound on queue sizes (Equation 2) also becomes quite large, producing two sources of error. First, if this bound becomes larger than our 1 MB implementation limit, we are unable to provide sufficient queue space for all flows to achieve full throughput. Second, our limits on capacity limit the amount we can adjust the upper bound on queue size, Equation 4, meaning that we may end up in a situation where it is not possible to satisfy both the upper and lower bounds.

It would be possible to raise these limits in our implementation by improving bandwidth shaping efficiency and allowing larger queue sizes. The underlying issues are fundamental ones, however, and would reappear at higher bandwidths: our emulator requires that capacity be significantly larger than the available bandwidth to be emulated, and providing emulation for large numbers of flows with high ABW requires large queues.

4.6 Shared Bottlenecks

Finally, we examine the effects of shared bottlenecks on bandwidth. We again measured the paths between a set of PlanetLab nodes, finding their bandwidth, reactivity, and shared bottlenecks After characterizing the paths in the real world, we configured two emulations. The first is a simple link emulation, approximating each path with an independent link emulator. The second uses our full path emulator, including its modeling of shared bottlenecks and reactivity. In order to stress and measure the system, we simultaneously ran an instance of `iperf` in both directions between every pair of nodes. This causes competition on the shared bottlenecks and also ensures that every path is being exercised in both directions at the same time.

Figure 9 shows a CDF of the bandwidth achieved at the bottlenecks in both the link emulator and our path emulator, demonstrating that failure to model shared bottlenecks results in higher bandwidth. To isolate the effects of shared bottlenecks and reactivity, only flows passing through those bottlenecks are shown. In the link emulator, each flow receives the full bandwidth measured for the path. In the path emulator, flows passing through shared bottlenecks are forced to compete for this bandwidth, and as a result, each receives less of it. Modeling of reactivity plays an important role here: in the path emulator, each shared bottleneck is being exercised by multiple flows, and thus the aggregate bandwidth available is affected by the response of the cross-traffic. The few cases in which the path emulator achieves higher bandwidth than the link emulator are caused by highly asymmetric paths, where the effects demonstrated in Section 4.1 dominate.

5 Related Work

There is a large body of work on measuring the Internet and characterizing its paths. The focus of our work is not to create novel measurement techniques, but to create accurate emulations based on existing techniques. Our contribution lies in the identification of principles that can be used to accurately emulate paths, given these measurements.

Our emulator builds on the Emulab [32] and Dummynet [22] link emulators to reproduce measured end-to-end path characteristics. ModelNet [27] also emulates router-level topologies on a link-by-link basis. Capacity and delay are set for each link on the path. To create shared bottlenecks with a certain degree of reactivity, it is up to the experimenter to carefully craft a router topology and introduce cross-traffic on a particular link of the path. ModelNet includes tools for simplifying router-level topologies, but does not abstract them as heavily as we do in this work. NIST Net [7], a Linux-based network emulator, is an alternative to Dummynet. However, it is also a link emulator and does not distinguish between capacity and available bandwidth. Our model abstracts the important characteristics of the path, thereby simplifying their specification and faithfully reproducing those network conditions without the need for a detailed router-level topology.

Appenzeller et al. [1] show that the queuing buffer requirements for a router can be reduced provided that a large number of TCP flows are passing through the router and they are desynchronized. They also provide reasoning as to why setting the queue sizes to the bandwidth-delay product works for a reasonably small number of TCP flows. We use the bandwidth delay product as the lower limit on the queue sizes of the paths being modeled. We are also concerned about low capacity links (asymmetric or otherwise) causing large queuing delays that adversely affect the throughput of TCP. Our model separates capacity from available bandwidth and determines queue sizes such that the TCP flows on the path do not become window-size limited.

Researchers have investigated the effects of capacity and available bandwidth asymmetry on TCP performance [2, 3, 11, 14]. They proposed modifications to either the bottleneck router forwarding mechanism, or the end node TCP stack. We do not seek to minimize the queue sizes at the router, but rather to calculate the right queue size for a path to enable the foreground TCP flows to fully utilize the ABW during emulation. We modify neither router forwarding nor the TCP stack and our model is independent of the TCP implementation used on the end nodes.

Harpoon [24], Swing [28], and Tmix [31] are frameworks that characterize the traffic passing through a link and then generate statistically similar traffic for emulating that link or providing realistic workloads. Our work, in contrast, does not seek to characterize or recreate background traffic in great detail. We characterize cross-traffic at a much higher level, solely in terms of its reactivity to foreground flows. We are able to do this characterization with end-to-end measurements, and do not need to directly observe the packets comprising the cross-traffic.

6 Conclusion and Future Work

We have presented and evaluated a new path emulator that can accurately recreate the observed end-to-end conditions of Internet paths. The path model within our emulator is based on four principles that combine to enable accurate emulation over a wide range of conditions. We have compared our approach to two alternatives that make use of simple link emulation. Unlike router-level emulation of paths, our approach is suitable for reconstructing real paths solely from measurements taken from the edges of a network. As we have shown, using a single link emulator to approximate a measured multi-hop path can fail to produce accurate results. Our path model corrects these problems, enabling recreations of real paths in the repeatable, controlled environment of an emulator.

Much of our future work will concentrate on improving the reactivity portion of our model. Our method of measuring reactivity is currently the most intensive part of our data gathering: it uses the most bandwidth, and takes the most time. Improving it will allow our system to run at larger scale. Viewing ABW as a function of the number of full-speed foreground TCP flows limits us both to TCP and to applications that are able to fill their network paths. In future refinements of our design, we hope to characterize ABW in terms of lower-level metrics that are not intrinsically linked to TCP's congestion control behavior. Finally, our averaging of ABW values for paths that share a bottleneck could use more study and validation.

Another future direction will be the expansion of our work to the simulation domain. Simulators handle links and paths in much the same way as do emulators, and the model we describe in Section 2 can be directly applied to them as well.

Acknowledgments

We thank our colleagues Eric Eide and Mike Hibler for their comments and proofreading assistance. Eric provided significant typesetting help as well. We also thank the anonymous NSDI reviewers, whose comments helped us to improve this paper greatly. This mate-

rial is based upon work supported by the National Science Foundation under Grant Nos. 0205702, 0335296, 0338785, and 0709427.

References

[1] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. SIGCOMM*, pages 281–292, Portland, OR, Aug.–Sept. 2004.

[2] H. Balakrishnan, V. Padmanabhan, G. Fairhurst, and M. Sooriyabandara. TCP performance implications of network path asymmetry. Internet RFC 3449, Dec. 2002.

[3] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *Proc. MobiCom*, pages 77–89, Budapest, Hungary, 1997.

[4] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. In *Proc. NSDI*, pages 155–168, San Jose, CA, May 2006.

[5] J. M. Blanquer, A. Batchelli, K. Schauser, and R. Wolski. Quorum: Flexible quality of service for Internet services. In *Proc. NSDI*, pages 159–174, Boston, MA, May 2005.

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

[7] M. Carson and D. Santay. NIST Net: a Linux-based network emulation tool. *Comput. Commun. Rev.*, 33(3):111–126, July 2003.

[8] P. B. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proc. SIGCOMM*, pages 147–158, Pisa, Italy, Sept. 2006.

[9] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Trans. Networking*, 11(4):537–549, Aug. 2003.

[10] D. Johnson, D. Gebhardt, and J. Lepreau. Towards a high quality path-oriented network measurement and storage system. In *Proc. PAM*, pages 102–111, Cleveland, OH, Apr. 2008.

[11] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. Improving TCP throughput over two-way asymmetric links: Analysis and solutions. In *Proc. SIGMETRICS*, pages 78–89, Madison, WI, June 1998.

[12] M. S. Kim et al. A wavelet–based approach to detect shared congestion. In *Proc. SIGCOMM*, pages 293–306, Portland, OR, Aug.–Sept. 2004.

[13] K. Lai and M. Baker. Nettimer: a tool for measuring bottleneck link, bandwidth. In *Proc. USITS*, San Francisco, CA, Mar. 2001.

[14] T. V. Lakshman, U. Madhow, and B. Suter. Window-based error recovery and flow control with a slow acknowledgement channel: A study of TCP/IP performance. In *Proc. INFOCOM*, pages 1199–1209, Kobe, Japan, Apr. 1997.

[15] R. Morris. TCP behavior with many flows. In *Proc. ICNP*, pages 205–211, Washington, DC, Oct. 1997.

[16] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation. *IEEE/ACM Trans. Networking*, 8(2):133–145, Apr. 2000.

[17] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. HotNets–I*, Princeton, NJ, Oct. 2002.

[18] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. NSDI*, pages 15–28, Cambridge, MA, Apr. 2007.

[19] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proc. SIGCOMM*, pages 367–378, Portland, OR, Aug.–Sept. 2004.

[20] V. Ribeiro et al. pathChirp: Efficient available bandwidth estimation for network paths. In *Proc. PAM*, San Diego, CA, 2003.

[21] R. Ricci et al. The Flexlab approach to realistic evaluation of networked systems. In *Proc. NSDI*, pages 201–214, Cambridge, MA, Apr. 2007.

[22] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Comput. Commun. Rev.*, 27(1):31–41, Jan. 1997.

[23] A. Shieh, A. C. Myers, and E. G. Sirer. Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In *Proc. NSDI*, pages 175–188, Boston, MA, May 2005.

[24] J. Sommer and P. Barford. Self-configuring network traffic generation. In *Proc. IMC*, pages 68–81, Taormina, Italy, Oct. 2004.

[25] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using PlanetLab for network research: myths, realities, and best practices. *Oper. Syst. Rev.*, 40(1):17–24, 2006.

[26] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. NSDI*, pages 253–266, San Jose, CA, May 2006.

[27] A. Vahdat et al. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI*, pages 271–284, Boston, MA, Dec. 2002.

[28] K. V. Vishwanath and A. Vahdat. Realistic and responsive network traffic generation. In *Proc. SIGCOMM*, pages 111–122, Pisa, Italy, Sept. 2006.

[29] K. V. Vishwanath and A. Vahdat. Evaluating distributed systems: Does background traffic matter? In *Proc. USENIX*, pages 227–240, Boston, MA, June 2008.

[30] M. Walfish et al. DDoS defense by offense. In *Proc. SIGCOMM*, pages 303–314, Pisa, Italy, Sept. 2006.

[31] M. C. Weigle et al. Tmix: a tool for generating realistic TCP application workloads in ns-2. *Comput. Commun. Rev.*, 36(3):65–76, July 2006.

[32] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Boston, MA, Dec. 2002.