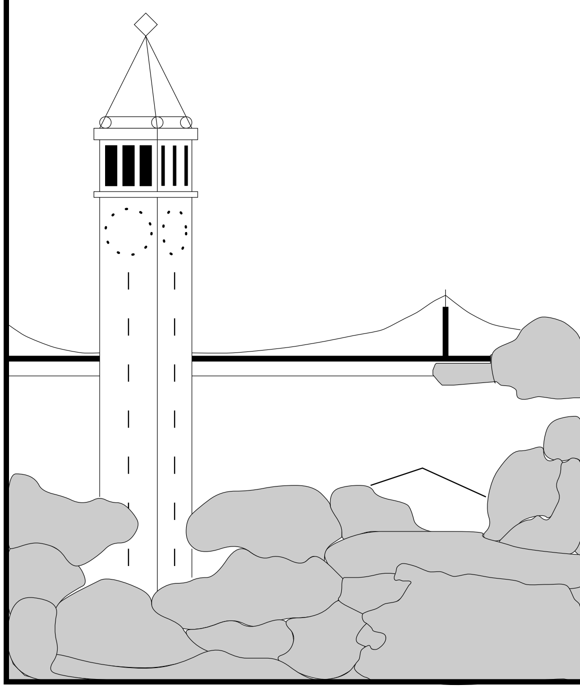
Building a Flexible and Efficient Routing Infrastructure: Need and Challenges

Karthik Lakshminarayanan *Ion Stoica* *Scott Shenker*



Report No. UCB/CSD-03-1254

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This research was supported by the NSF under Cooperative Agreement No. ANI-0225660, ITR Grant No. ANI-0085879, and Career Award No. ANI-0133811.

Building a Flexible and Efficient Routing Infrastructure: Need and Challenges

Karthik Lakshminarayanan Ion Stoica Scott Shenker

Abstract

The Achilles heel of the otherwise extremely successful Internet infrastructure has been its rigidity, which has primarily stemmed from the ossification of *shortest-path* routing in the basic architecture. The increasing effect of this rigidity of the current Internet infrastructure, coupled with the popular belief that basic IP routing cannot be changed, has led to many companies and researchers turn to infrastructure-based overlay networks to meet specific application requirements. These overlay networks, however, are mostly independent efforts, sharing nothing but the underlying IP infrastructure. We first try to argue for the need for reversing this trend, and in the process propose a panacea – a global shared overlay infrastructure.

We envision that a single set of overlay infrastructure nodes, supporting a few simple primitives, would allow end-hosts to choose routes over the infrastructure, thus enabing the end-hosts to achieve various services they desire. The foremost challenge we face here is to design flexible primitives that the infrastructure should export. The second requirement is to support a mechanism that allows end-hosts to find paths based on application-sensitive metrics. We achieve this by building a NEtwork Weather Service (*NEWS*) that measures performance characteristics of the infrastructure. The final requirement, also of paramount importance, is to make sure that the infrastructure is DoS resistant [1]. Thus, end-hosts, by querying the *NEWS* nodes build application-specific services using the routing primitives that the infrastructure exports. Experiments using an initial deployment of *NEWS* over PlanetLab have shown that our techniques perform very well.

1 Introduction

The widespread adoption and complete commercialization of the Internet over the past decade has led to a contradictory state of affairs. On one hand, the continuing advent of new Internet applications is posing an increasingly varied set of demands on the infrastructure. On the other hand, the Internet infrastructure is becoming more rigid and narrow; the infrastructure's size impedes change, and the few changes that do occur, such as the increasing use of middle-boxes, are often driven more by short-term profit than by long-term architecture and so frequently inhibit rather than enable the introduction of new applications.

This clash between an increasing set of needs and a diminished capacity to meet those needs has caused many researchers and companies to turn to infrastructure-based *overlay* networks (such as [3, 8, 19]). Overlay networks insert functionality at the application level, thereby circumventing the rigidity of the underlying Internet infrastructure. Infrastructure-based overlays run over a dedicated set of well-connected nodes in the network; in contrast, host-based overlay networks, such as those used in peer-to-peer networks and end-host multicast, only use the end-hosts engaged in the application itself. Roughly speaking, infrastructure-based overlays provide superior performance and reliability (thus most commercial overlays fall into this category), while host-based overlays are easier to deploy ("grass-roots" overlay applications, such as Gnutella, are host-based).

A case for sharing higher level overlay functionality

Infrastructure-based overlay networks have been built or proposed for a wide variety of uses, such as multicast [8, 12, 19, 11, 24, 28, 34, 33], content distribution [3], quality of service [32], route quality and reliability [30, 4], and data manipulations such as transcoding on data path [14, 27]. However, these various infrastructure-based overlay networks have been independent application-specific efforts; each services a different application (or application requirement) and each requires a large investment of design effort and/or deployment expense. This lack of synergy between overlays is a sad and particularly ironic fate for the Internet, which was based on the sharing of resources (packet level multiplexing) and the generality of the interface (minimally designed so that it could support a wide variety of applications).

A case for sharing network weather information

Since most of the overlay networks emerged in a bid to break away from the shackles of the restricted routing primitive offered by IP, they perform measurements to probe for paths that offer better end-to-end performance. Sharing the information gained thus would be beneficial as it would help the scalability (one could probe more paths with the same amount of resources) of the system and accuracy (one could probe more frequently with the same amount of resources) of the measured data.

From a design perspective, building path-probing mechanisms in the overlay application would be a huge design (and perhaps development) effort. Applications would ideally want an API to specify, to some level of granulariy, the characteristics they require of the paths.

In this paper, we ask whether we can have overlay networks be built on top of common overlay functionality.

[1] We do not address this directly, but refer the reader to [2]

Our vision is that a single set of overlay infrastructure machines, supporting a few simple primitives, would allow end-hosts to achieve the various services they desire. In this vision, the desired service, which previously was embedded in the overlay, is now constructed by the end-hosts using the primitives supported by the infrastructure nodes. Just as the Internet replaced telephony's application-specific smart-network/dumb-host combination with the current general-purpose dumb-network/smart-host architecture, our proposal replaces application-specific overlays with a shared set of simple, but general-purpose, overlay nodes combined with sophisticated end-host applications.[2] If successful, this approach would combine the best of both overlay approaches: the performance of infrastructure-based and the deployability of host-based.[3]

However, the crucial challenge in realizing this vision is to identify a few key primitives that, when embedded in each overlay node, are sufficient to support a wide range of overlay services. We discuss such a set of primitives, and how functionality should be divided, in Section 2. We then describe the implementation of these primitives in Section 2.3. Section 3 describes the different components of the system and their roles. In Section 4, we describe how the primitives allow hosts to measure overlay path characteristics, and in Section 5 we describe how to scale *NEWS* . How several applications can use this approach is described in Section 6 and, in Section 8, we report on an experimental evaluation of our design. We conclude in Section 10 with a discussion.

2 System design

We shall reason out how the functionality should be split among the different entities of the system, and then present a set of primitives to achieve the required functionality.

2.1 Extent of sharing

While many overlays have diverse purposes, at a very general level, one can divide their functionality into two pieces:

Routing control: this encompasses both the selection of paths and the replication of packets along the path (such as in multicast overlays).

Data manipulation: some overlays employ transcoding or otherwise manipulate data (including storing the data) as it is conveyed to the destination.

To what extent can these functions be shared is the question we must address now. However, it should be noted that functions can be shared *without embedding them in the overlay infrastructure*; instead, one can share functionality by using "third-party" services. By a third-party service, we refer to a service implemented by hosts, but offered to applications through the use of an open interface; these third-party services are thus application-level but not application-specific. Therefore, the issue of sharing breaks down into two separate questions:

- Should these functions be shared across different overlays?
- If so, should these functions be embedded in the overlay infrastructure, or supported by third-party services?

We first address these two questions for routing. Routing control is comprised of three pieces. First, there are the low-level mechanisms that control the forwarding and replication of packets. Forwarding and replication are quite general functions that require high performance, so it seems likely that they should be incorporated into the infrastructure nodes. Second, there are the policies that express where to forward (and replicate) packets; such policies would, for instance, dictate the relative importance of latency and bandwidth when choosing a path. These policies vary widely between applications; thus, they should probably not be shared and should definitely not be embedded in the infrastructure. For *e.g.,* , the topologies constructed for streaming media would be very different from that for a voice conference.

Third, there is the information on which these routing decisions will be based; for example, measurements of path bandwidth, latency, loss, and stability will be relevant to making routing decisions. It seems likely that such information could be usefully shared between applications. However, the set of measurements and the granularity of measurement data that one might need in order to make routing decisions is not so narrowly specified that one could reliably embed this into the infrastructure. Instead, we envision third-parties offering a shared, but application-level, *network weather service* that provides the relevant information. By a third-party service, we mean that it operates from hosts, but the information is shared across applications. While not as efficient as embedding measurements in the infrastructure, implementing the measurements in the end-hosts allows the data-collection algorithms to evolve over time in response to changing needs. However, the infrastructure should be designed to make it possible for third-parties to gather the necessary measurement data.

The second broad class of functionality is data manipulation. Data manipulation tends to be very application specific (*e.g.,* transcoding between two specific encodings) and will likely evolve over time. Thus, data manipulation, at least at this point, shouldn't be shared (though as particular transcodings become popular one could imagine third-parties offering such services). However, the infrastructure should make it easy to insert special-purpose computation on the data path; more precisely, it should make it easy to adjust the path to reach these computational nodes.

[2] As we will describe later, these applications can avail themselves of third-party services which are not part of the overlay infrastructure but yet can be shared across applications.

[3] This assumes that the shared infrastructure has already been deployed; while its deployment still faces significant barriers, these difficulties will be amortized over many subsequent uses.

2.2 Infrastructure Primitives

Our discussion of sharing points towards a design where there are two main shared capabilities, routing information and routing mechanism, which are shared at different levels. Routing-relevant information should be shared through application-level third-party services, and low-level routing mechanisms should be embedded in the shared infrastructure. Our discussion further indicates that these low-level routing mechanisms should enable end-hosts to:

1. Control forwarding and replication
2. Allow special-purpose data manipulation nodes to be included on data paths
3. Measure performance characteristics of paths in the infrastructure

We claim that two routing primitives are sufficient to meet these three requirements. The two primitives are:

Path selection: An end-host specifies a path $(a_1, a_2, \ldots, a_n)$ to be followed by a packet p, where $a_1, a_2, \ldots, a_{n-1}$ identify nodes in the infrastructure, and a_n identifies the destination end-host. Between two consecutive nodes along the path, a_i and a_{i+1}, the packet is forwarded by IP routing.

Packet replication: An end-host can request an arbitrary node a to replicate a packet p (that traverses node a) and forward the replica along a path $(b_1, b_2, \ \ldots, b_n)$. A packet p can be identified by a subset of fields in its header such as the source and destination IP addresses, and eventually the source and destination port numbers.

It is fairly clear that these primitives satisfy the first requirement (control of forwarding and replication). The satisfaction of the second requirement follows from satisfying the first (once hosts can control routing, they can direct routes toward the computational nodes). The fact that these two primitives satisfy the third requirement (enabling hosts to measure path characteristics) is less clear. We discuss this issue at length in Section 4, where we demonstrate that hosts can use these primitives to measure the characteristics (such as delay, loss rate) of the path between any two infrastructure nodes (we call these paths the "virtual links" in the overlay).

2.3 Implementation Alternatives

The most obvious possibility is to implement these primitives at the IP layer. In fact, IP already implements path selection in the form of *loose source routing* [26]. With loose source routing, the packet's IP header carries the path $(a_1, a_2, \ldots, a_n)$. Upon arriving at node a_i, the packet is forwarded to next node along the path, a_{i+1}, via IP. To implement packet replication, we can add a simple primitive that causes a node a to replicate a given packet p, and then replace the path in the header of replica p with a new path $(b_1, b_2, \ \ldots, b_n)$ provided by the end-host. In this case packet p is identified by its IP source and/or destination address.

Another alternative is to implement the path selection and packet replication primitives using loose source routing at the transport layer instead of the IP layer. In this case the hops along the path represent processes (identified by an IP address and a port number) instead of nodes. This scheme makes it easier to support data manipulation functionalities, as an infrastructure node can run multiple services, each identified by a port number, and end-hosts can use the path selection primitive to route the packets through these services.

In general, loose source routing implementations assume that the packet carries the path in its header. Another approach would be to *set-up* forwarding state at all hops along the path. Suppose a host wants to send a packet p along path $(a_1, a_2, \ldots, a_n)$. Then instead of inserting the path in the packet's header, the host inserts forwarding state $(packet\_id(p), a_{i+1})$ at each node a_i, $1 \leq i < n$, where $packet\_id(p)$ denotes the identifier of packet p. Upon, receiving a packet p, node a_i forwards the packet to next hop a_{i+1}. While this solution does not require packets to carry the path in their headers, it requires end-hosts to set-up the path before sending the packets. Thus, this solution is more appropriate when sending a large number of packets along the same path.

A variation of the previous solution is to use a protocol like *label switching* [6]. With this solution, each packet has a local identifier (or label) at each hop along the path, instead of a unique identifier across the entire network. Let $packet\_id(p, a_i)$ denote the identifier[4] of packet p at node a_i. Then an end-host inserts the forwarding entry $(packet\_id(p, a_i), packet\_id(p, a_{i+1}), a_{i+1})$ at every hop a_i along p's path. The source sends packets with identifier a_1. Upon receiving packet p, node a_i checks whether it has an entry for that packet, and if it does, it replaces p's identifier (*i.e.,* $packet\_id(p, a_i)$) with the packet identifier at the next hop, $packet\_id(p, a_{i+1})$, and then forwards p to a_{i+1} via IP.

While each of the above implementations approaches are viable, we have chosen to implement these primitives using $i3$; our reasons are threefold. First, $i3$ can support the two primitives without any changes. Second, $i3$ is robust in the presence of node failures because, if a node fails, the traffic is transparently routed around the failed node. Third, $i3$ is robust against denial of service attacks on both the infrastructure and end-hosts [2].

Implementation using $i3$:

We choose $i3$ as an instantiation of the two primitives. $i3$ indeed supports these primitives naturally, but our choice was influenced by the following reasons.

1. *Identification of infrastructure nodes:* End-hosts need a mechanism of identifying the nodes of the infrastructure as they need to explicitly choose the paths. One

[4]Here we assume that the packet has a special identifier field such as the flow label field in IPv6 that stores packet's identifier.

obvious possibility is to use the IP address of infrastructure nodes. However, this solution suffers from the following problems: (i) lack of anonymity of infrastructure nodes, and (ii) explicit failover mechanisms have to be embedded in end-hosts in the event of an infrastructure node failing. $i3$, on the other hand, provides a natural mapping of nodes to logical identifiers and robust failover mechanisms.

2. *Security:* Our related work [2] shows that $i3$ is robust to DoS attacks on both the infrastructure and the end-hosts that use it. Allowing end-hosts to perform measurements using the same communication primitives provides the infrastructure implicit protection. However, we note that a consequence of using $i3$ is that an end-host that performs measurements must have resources of the same order of magnitude as that consumed by performing the measurement. This implies that only powerful nodes (such as *NEWS* providers) would be able to maintain the map of the entire infrastructure.

There is no doubt that some of the functionality we discuss can be and has been implemented at the IP layer, e.g. loose source routing. However, security implications coupled with the inability to change functionality at the IP layer has been the reason for the lack of widespread deployment in the Internet. We believe that implementing the primitives at the overlay would be a first step in understanding how well they cope up with the demands of applications, and would probably lead to these primitives being pushed down the Internet protocol stack in the future.

Target deployment:

We conceive of a large-scale shared overlay infrastructure of nodes with very high connectivity. However, we are not sure about the economic model of the infrastructure and the *NEWS* system, whether there would be a single for-profit provider (e.g. Akamai), multiple for-profit providers (e.g. like ISPs today), or a cooperative nonprofit system (e.g. PlanetLab). However, since cooperation of ISPs is not required, third-parties can provide this service easily.

3 System architecture

Figure 1 illustrates the components of our system architecture and their interaction.

A. Overlay infrastructure The overlay infrastructure consists of a set of nodes that implement our two primitives: *path selection* and *packet replication*. In addition, some of the nodes may implement other services such as data manipulations, *e.g.,* transcoding. However, defining the interfaces for these more application-specific services is not the subject of this paper. Here we only assume that path selection allows end-hosts to insert the data manipulation services in the data path by controlling the routing of its packets.

B. NEtwork Weather Service (*NEWS*) The central component of our architecture is the *NEtwork Weather Service*

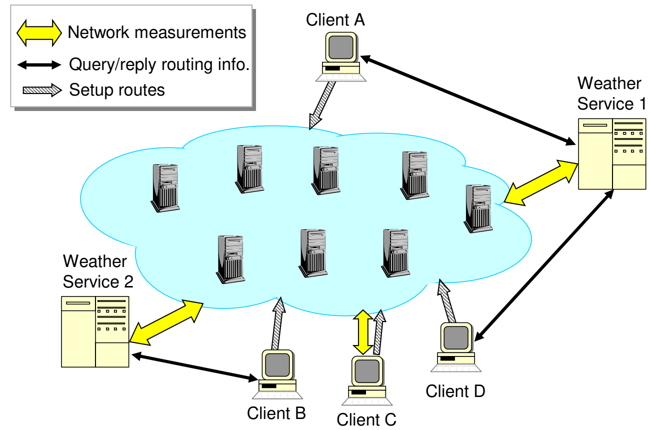


Figure 1: System Architecture. NEtwork Weather Service(*NEWS*) provider's agents measure and summarize performance characteristics in the overlay. Clients request routing information from these agents and, based on this information, set up its own routes in the overlay. Clients may also use to do their own measurements (e.g., client C).

(*NEWS*). *NEWS* is a third party service that uses the two primitives to measure the available bandwidth, loss rate, and latency between various nodes in the infrastructure. The *NEWS* system uses this information to maintain a performance map of the overlay infrastructure, and to provide clients with path information that satisfies clients' requirements. While in this paper we assume that there is only one *NEWS* provider, in general there can be several that monitor the infrastructure. Different *NEWS* providers can optimize their measurements for different application classes, such as file transfer, or video streaming.

C. End-hosts Assume a client wants to send data traffic between nodes n_1 and n_2 in the infrastructure. Then, the client queries one of the *NEWS* agents (which is a node of the *NEWS* system) about a route between nodes n_1 and n_2 by specifying its performance requirements, *e.g.,* minimize the delay or maximize the available bandwidth between the two nodes. In turn, the agent returns a route that meets the client's performance requirements. Upon receiving the reply, the client uses the path selection primitive to send data along the path that *NEWS* returns. Alternatively, a *NEWS* agent can return a set of paths between nodes n_1 and n_2, and let the client select the best path among all those paths.

Scaling a *NEWS* system:

Since the overlay infrastructure we are dealing with is extremely large (possibly 1000s of nodes), monitoring the entire set of N^2 overlay links (N is number of infrastructure nodes) is not feasible. By maintaining the overlay network as a hierarchical random graph, the *NEWS* system monitors only a small subset of links at the expense of providing marginally worse results to the clients. We discuss this later in the paper.

4 Performing measurements using infrastructure primitives

To provide greater flexibility, an infrastructure should allow any end-host to measure performance characteristics between arbitrary infrastructure nodes. In this section, we describe how end-hosts can do this measurement using only the two primitives that we have introduced. Before doing so, we discuss the pros and cons of this approach.

Advantages:

1. Flexibility: Anyone with sufficient resources can start a *NEWS* system tuned to the applications that they target. The particular algorithms for measurement can be changed as and when needed. Infrastructure cannot distinguish between data and measurement traffic, and so cannot "stop" any measurement.
2. Long-term efficiency: Measurement done outside the infrastructure has the advantage that it is performed only if necessary and only up to the granularity needed. E.g. if all applications need coarse bandwidth data, an infrastructure performing measurement at a fine granularity would be wasting bandwidth.
3. Security: We leverage the results from [2] that building our infrastructure using $i3$ guarantees that an end-host with limited resources cannot arbitrarily DoS the infrastructure. Introducing new primitives that performs certain active measurements (e.g. bandwidth measurement) might not have this property, and hence the infrastructure cannot allow anyone to use that primitive.

Disadvantages: Compared to direct measurements at the infrastructure nodes, our indirect techniques might have lower accuracy. Moreover, each instantiation of a measurement might consume more resources than direct techniques.

We do recognize the possibility of infrastructure nodes providing more support by performing measurements actively or passively. However, (i) this study is orthogonal to our work and would work well to improve our system (this is especially true of passive measurement techniques), and (ii) we have to protect against end-hosts launching DoS attack which one can envisage if infrastructure performs active measurements. We defer this study to the future.

4.1 Delay

Consider the problem of estimating the round-trip time (RTT) between two arbitrary nodes n_1, and n_2, respectively.[5] Figure 2(a) illustrates the technique used by a host R to perform this measurement. The main idea is to send packets (probes) along paths (R, n_1, R) and (R, n_1, n_2, n_1, R), respectively, and then compute the RTT between n_1 and n_2 as the difference between the RTTs of these packets.

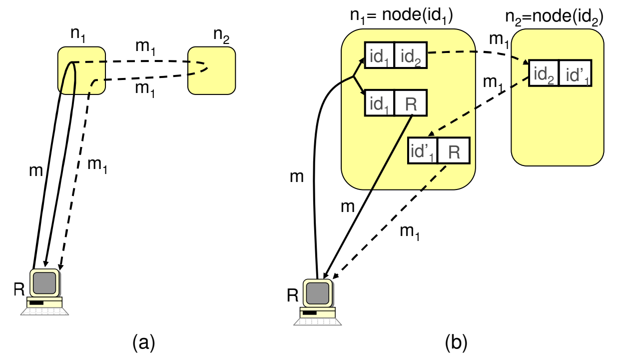


Figure 2: (a) Communication pattern used to measure the round-trip time (RTT) between two IDs from a remote host R. R measures the RTT between nodes n_1 and n_2 as the time interval between receiving back the original probe m and its copy m_1. (b) The implementation of the communication pattern in $i3$.

More precisely, R uses (1) the path selection primitive to periodically send a probe m along the path (n_1, R), and (2) the packet replication primitive to ask n_1 to replicate each probe m and then send the replica (m_1) along path (n_2, n_1, R). The following actions take place at each node as a result of sending a probe m:

- n_1: upon receiving m, n_1 sends the probe back to R, and at the same time creates a new copy of m (call it m_1) and sends it to n_2.
- n_2: upon receiving copy m_1, node n_2 sends the copy back to R via node n_1.
- R: R computes the RTT between n_1 and n_2 as the time interval between the arrival time of probe m, and the arrival time of m's replica m_1.

Figure 2(b) shows how this technique can be implemented in $i3$. R chooses three IDs id_1, id_2, and id'_1 such that id_1 and id'_1 identify (*i.e.,* are mapped on) node n_1, and id_2 identifies node n_2. Then R inserts the following four triggers: (id_1, R), (id_1, id_2), (id_2, id'_1), and (id'_1, R), and then sends a probe $m = [id_1, dummy]$ periodically.

4.2 Loss Rate

In this section we consider the problem of measuring the *unidirectional* loss rate between two nodes, n_1 and n_2, respectively. To achieve this, we use a similar setting as the one used to measure the RTT between nodes with the difference that R uses the packet replication primitive to ask node n_2 to replicate probe m_1 and send the new replica, m_2, back to R (see Figure 3).[6] Replica m_2 is used to differentiate between a loss on the virtual link $(n_1 \rightarrow n_2)$ and a loss on the virtual link $(n_2 \rightarrow n_1)$.

[5] Measuring the one way delay between n_1 and n_2, while more desirable, requires the clocks of the two nodes to be synchronized. Thus, measuring the one way delay is difficult even assuming full control on the two nodes.

[6] To generate this new copy in $i3$, R needs only to add a fifth trigger, (id_2, R), to the setting in Figure 2(b).

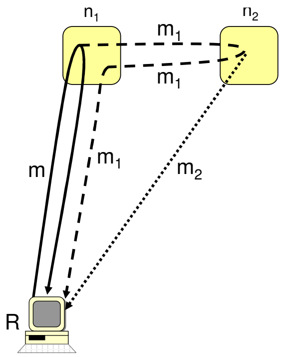


Figure 3: Communication pattern used to estimate the loss rates along virtual links $(n_1 \rightarrow n_2)$, and $(n_2 \rightarrow n_1)$, respectively.

After sending a probe m, R concludes that there was a loss on the virtual link $(n_1 \rightarrow n_2)$, *i.e.,* replica m_1 was lost between nodes n_1 and n_2, if R receives probe m back, but it does not receive any of the replicas m_1 and m_2.

Next, we give an intuition of why this test might work in practice by showing that probability of false positive is small. Let $P(l, a \rightarrow b)$ denote the probability that packet l is lost on link $(a \rightarrow b)$. Assume that the loss probability on each virtual link is p, where $p \ll 1$, and that the loss probabilities are not correlated. The probability of false positives, *i.e.,* the probability that R incorrectly decides that m_1 was lost on link $(n_1 \rightarrow n_2)$, is

$$\begin{aligned} P &= (1 - P(m_1, n_1 \rightarrow n_2)) \times \qquad (1) \\ &\quad (1 - [1 - P(m_1, n_2 \rightarrow n_1)][1 - P(m_1, n_1 \rightarrow R)]) \times \\ &\quad P(m_2, n_2 \rightarrow R) \\ &\simeq 2p^2 \end{aligned}$$

where the first term on the right hand side, $(1 - P(m_1, n_1 \rightarrow n_2))$ represents the probability that m_1 was not dropped on $(n_1 \rightarrow n_2)$, the second term represent the probability that m_1 was dropped either on $(n_2 \rightarrow n_1)$ or $(n_1 \rightarrow R)$, and the last term represents the probability that m_2 was lost on $(n_2 \rightarrow R)$. Thus, the probability of false positive, $2p^2$, is considerably smaller than the measured loss rate p. In Section 8.1.2 we use extensive wide area measurements to validate this technique.

While R can estimate the loss rate on the reverse link, $(n_2 \rightarrow n_1)$, by inverting the communication pattern shown in Figure 3, next we give a solution that allows R to estimate this loss rate without any additional measurements. In particular, while measuring the loss rate on the direct link, $(n_1 \rightarrow n_2)$, R also records the following two events:

1. the receiving of m_2 but not of m_1
2. the receiving of m_1 or m_2 but not of m

Let f_1 be the frequency of occurrence of event 1, and f_2 be the frequency of occurrence of event 2. Then f_1 estimates the loss rate along virtual path $(n_2 \rightarrow n_1 \rightarrow R)$, while f_2 estimates the loss rate on virtual link $(n_1 \rightarrow R)$. Finally, R estimates the loss rate on $(n_2 \rightarrow n_1)$ as $f_1 - f_2$. Note that this estimation procedure assumes that the losses on links $(n_2 \rightarrow n_1)$ and $(n_1 \rightarrow R)$ are not correlated. Finally, it can be shown that if the loss probability on each virtual link is $O(p)$, the probabilities of false positives for both f_1 and f_2 are $O(p^2)$.

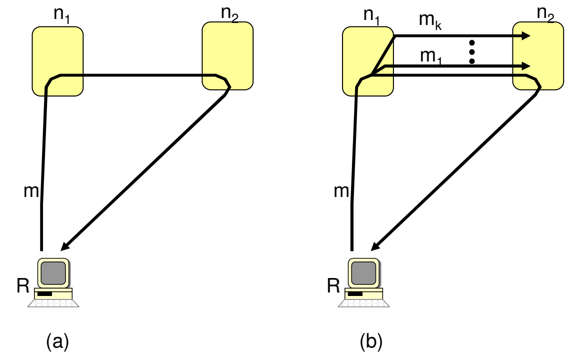


Figure 4: Communication pattern used to estimate the available bandwidth on virtual link $(n_1 \rightarrow n_2)$ when (a) the bottleneck is at $(n_1 \rightarrow n_2)$, and (b) when the bottleneck is either at $(R \rightarrow n_1)$ or at $(n_2 \rightarrow R)$.

4.3 Available Bandwidth

To measure the available bandwidth (*avail-bw*), we use a TCP-Vegas like algorithm [5]. Such an algorithm reacts to congestion when the RTT increases rather than waiting for a packet loss. This helps to minimize the impact of the measurement algorithm on the background traffic. Note that the technique we present here can be easily extended to say, TCP Reno, and so what we try to emphasize here is the flexibility of the indirect measurement technique, and not the fact that we use delay based technique.

To estimate the avail-bw on virtual link $(n_1 \rightarrow n_2)$, a host R sends traffic on path $(R \rightarrow n_1 \rightarrow n_2 \rightarrow R)$ as shown in Figure 3(a). R uses a slow start algorithm which exponentially increases the congestion window size, and consequently the sending rate, until the RTT exceeds the minimum RTT observed so far by a predefined threshold. When this happens R concludes that there is congestion on path $(R \rightarrow n_1 \rightarrow n_2 \rightarrow R)$. In order to determine whether $(n_1 \rightarrow n_2)$ is the congested virtual link, R:

1. uses packet replication to ask n_1 to replicate each packet R sends and to forward the replica to n_2
2. reduces the sending rate by half.

The net result of these operations is that while the rate of the traffic on both $(R \rightarrow n_1)$ and $(n_2 \rightarrow R)$ is halved, the rate on link $(n_1 \rightarrow n_2)$ remains unchanged, as each packet is now sent twice on this link.

If RTT does *not* decrease as a result, R concludes that $(n_1 \rightarrow n_2)$ is congested and that the *avail-bw* on the link is twice R's current sending rate. If not, R repeats the process. The process ends when a decrease of the sending rate and a corresponding increase of the replication factor k on link $(n_1 \rightarrow n_2)$ does not cause the RTT to decrease. Let b be the sending rate of R when the process terminates. Then the available bandwidth on link $(n_1 \rightarrow n_2)$ is estimated as $b \cdot k$. Furthermore, when replicating packets on $(n_1 \rightarrow n_2)$, R scales the TCP parameters appropriately such that it emulates a normal TCP flow.

subsectionBottleneck Bandwidth

To measure the bottleneck bandwidth between two nodes n_1 and n_2 we use a packet-pair like technique [20]. Consider a similar communication pattern as the one shown in Figure 4. R sends pairs of packets on the path $(R \rightarrow n_1 \rightarrow n_2 \rightarrow R)$. The inter-departure time between any two pair of packets is maintained fixed. Then, R monitors the inter-arrival time between each pair of packets, while increasing the number of replicas on the virtual link $(n_1 \rightarrow n_2)$. When the inter-arrival time starts to increase, R stops. Let d be the inter-arrival time, and k be the number of replicas at the end of the experiment. Then the bottleneck bandwidth of the virtual link $(n_1 \rightarrow n_2)$ is computed as $(kl)/d$, where l represents the packet length.

In our estimation we assume that the IP routers use a FIFO scheduling discipline. This is a reasonable assumption in today's Internet. In fact, all algorithms to estimate bottleneck bandwidth that we are aware of (such [10, 21]) use this assumption.

5 How to scale *NEWS*

Ideally, a WS would monitor all possible virtual links in the infrastructure. However, in a large infrastructure this is infeasible, as the total number of links is $O(N^2)$, where N is the number of nodes. In this section, we present three simple designs which exploit the trade-off between scalability and the "quality" of the paths returned by the WS.

5.1 Random Graphs

The first approach we explore to reduce the number of links that need to be monitored in the overlay is to monitor a random subset of the links. Equivalently, the WS maintains the overlay network as a random graph. To construct a random graph with an average degree d, WS picks $d/2$ random virtual links starting at every node in the infrastructure and monitor only those links. If one of those links (say $L = (a, b)$) fails, the WS will replace it with another random link. This link is chosen adjacent to the node, chosen from a and b, which has the lower degree. For $d = O(\log N)$ the graph is connected with high probability.

This simple approach has several advantages. First, a random graph is easy to construct and maintain. Second, they are efficient in terms of number of hops needed for reachability. A random graph with N nodes and average degree d has an average path length of $O(\log_d N)$. The diameter [7] of the graph is with high probability no larger than $2 \ \log_d N$. For comparison, note that $\log_d N$ represents the lower bound on the diameter of any graph of degree d.

A random graph does not optimize for any particular metric. As a result, a random graph can be suboptimal with respect to every single metric. We address this problem next.

5.2 Weighted-random Graph

In this section, we give a simple algorithm to construct a pseudo-random graph optimized for a given metric. For example, let us consider the problem of building a graph with average degree d that provides low latency paths. One way to achieve this goal is to pick for each node (i) $d_1/2$ virtual links to random nodes, and (ii) d_2 links to its closest d_2 neighbors; these links are called *proximity* links (Note that $d_1 + d_2 = d$). To find the proximity links of a node n, the WS can start with d_2 random links originating at n, and then constantly probe other links originating at n. If the WS finds a link originating at n that is shorter than a proximity link of n, then it replaces the largest proximity link of n with the new link.

The resulting graph is a superposition of a random graph with average degree d_1, and a graph in which each node knows its closest d_2 neighbors. Compared to a random graph with the same average degree, this graph is slightly less robust, but provides paths with lower latencies.

A possible disadvantage of weighted-random graphs is that a WS may need to construct and maintain a different graph for each metric. Indeed, a graph specifically built to optimize a metric is not always appropriate for another metric unless the two metrics are correlated.

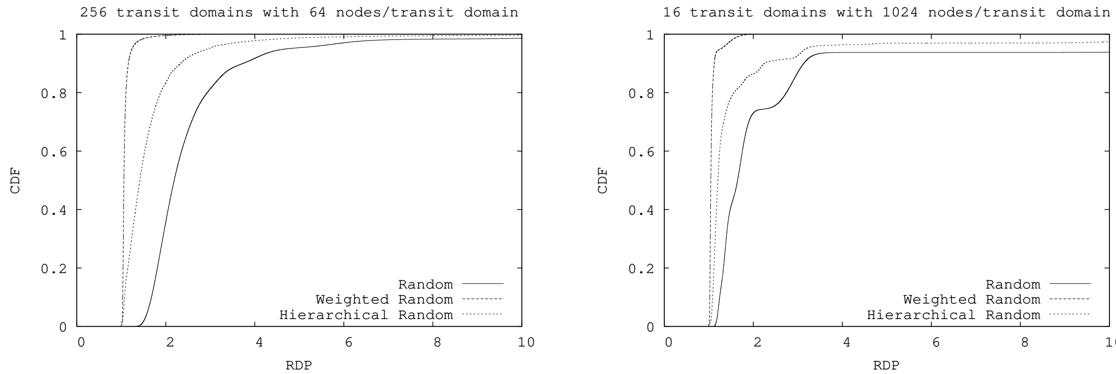
5.3 Hierarchical Random Graphs

So far we have implicitly assumed that the WS knows the entire graph. However such an approach will not scale for large networks consisting of thousands of nodes or more. In this section, we discuss a simple solution to alleviate this problem.

Our solution builds a two-level hierarchy. At the first level, nodes are randomly partitioned into buckets of roughly equal sizes. Each node has d_1 links to nodes in the same bucket, and d_2 links to the closest nodes that belong to other buckets. For reasons that will be clear soon, in this construction, we make sure that there is at least one link from each bucket to any other bucket.

The scalability of this solution stems from the fact that each bucket can be maintained by a different server. Let S_i be the server responsible for the nodes in bucket i. Then S_i will monitor all links originating at the nodes in bucket i. Assume a client wants to find a path from node n_1 to node n_2. Two cases arise: (i) If both nodes belong to the same bucket then the client needs to contact only the server responsible for that bucket. (ii) Suppose now that the two nodes belong to different buckets, *i.e.,* node n_1 belongs to bucket i and node n_2 belongs to bucket j. In this case the client needs to contact both servers S_i, and S_j. Upon receiving the client's request,

[7] The diameter of a graph is defined as the longest *shortest path* in the graph



(a)

(b)

Figure 5: The CDF of the relative delay penalty (RDP) for three overlay graphs: random, weighted random, and hierarchical. In all cases, the size of underlying graph is $16,384$, the number of overlay nodes is $4,096$ and their average degree is around 20.

server S_i returns all paths that originate at node n_1 and reach bucket j, while server S_j returns all paths that originate at node n_2 and reach bucket j. The client uses this information to compute the best path between n_1 and n_2.

We make some observations regarding the complexity of the algorithm. (i) Let N be the total number of nodes in the infrastructure, and let $M = \sqrt{N}$ be the number of buckets. Then, each WS server is responsible for only $O(M)$ measurements. (ii) In order to service requests from clients (to get shortest paths) quickly, a WS may decide to precompute all pairs shortest paths within its bucket. Since there are M^2 pairs, each having an average path length of $logM$, total space required would be $O(M^2 \ \log M)$. (iii) Finally, a WS reply contains d_2 paths on average. To put things in perspective assume an infrastructure with 10^4 nodes, and suppose $d_1 = d_2 = 10$. Then, each server needs to monitor only about $M \times (d_1 + d_2) = 2000$ virtual links on average, and store about 10^4 pre-computed paths. We believe these values are feasible in practice.

5.4 Comparison

In this section, we use simulations to compare the three graph construction methods: random, weighted random, and hierarchical. We use a transit-stub topology generated with the GT-ITM topology generator [15] with $16,384$ nodes, where link latencies are 100 ms for intra-transit domain links, 10 ms for transit-stub links and 1 ms for intra-stub domain links. There are $4,096$ infrastructure nodes randomly assigned to stub nodes.

Figure 5 shows the cumulative distribution function (CDF) of the relative delay penalty (RDP) for two transit stub hierarchies (see [1] for details). RDP is defined as the ratio of the shortest path in the overlay graph and the shortest path in the underlying network. The results are aggregated over 10 different runs. For each overlay graph we chose parameters d, d_1, and d_2 such that the average degree of a node is about 20. In the case of the weighted random graph, each node has 10 random neighbors and 10 closest neighbors on average. In the case of the hierarchical graph each node has around 10 closest neighbors to other buckets, around 6 random neighbors in the same bucket, and around 4 closest neighbors in the same bucket. As expected, the weighted random graph performs much better than the random graph, with the hierarchical random graph in between. More precisely, in Figure 5(a), the 90-th percentile of the RDP is 3.74 for the random graph, 1.16 for the weighted random graph, and 2.33 for the hierarchical graph, respectively. Similarly, the 90-th percentiles in Figure 5(b) are 3.14, 1.12, and 2.25, respectively.

6 Applications

In this section, we give several examples of how our infrastructure can be used by applications.

Adaptive Routing Several previous studies [4, 30] have observed that Internet routes are not always optimal and that routing packets through intermediate nodes (*i.e.,* use some form of loose source routing) can significantly improve end-to-end performance. In our system end-hosts can easily optimize for various performance metrics. Consider a sender s and a receiver r connected at the overlay nodes n_s and n_r respectively. Then, either s or r can query the WS for the best path between n_s and n_r given an application specific performance metric such as latency, bandwidth, or loss rate. Note that in our system, the quality of the paths depends only on how sophisticated the WS is. Because the WS is not a part of the infrastructure, the quality of the paths can constantly evolve without any changes to the infrastructure.

Multicast Using our infrastructure to implement single source overlay multicast trees is straightforward. Consider a multicast source S that is connected to overlay node n_s. Suppose a receiver r_i, connected at overlay node n_i, wants to join the multicast group. Receiver r_i asks the WS for an overlay path from n_s to n_i. Let $(n_s, n_1^i, \ldots, n_k^i, r_i)$ be this path. Then r_i asks node n_s to send a replica of the multicast packets to r_i along the nodes $(n_1^i, \ldots, a_k^i)$. We make some observations:

1. The multicast tree consists of the union of all unicast paths from n_s to each receiver. Thus, the path from sender to each receiver in the multicast tree is as efficient as the unicast path.
2. Each receiver can optimize its path in the multicast tree using any of the available metrics. Building a multicast tree that minimizes latency is as easy as building a multicast tree that maximizes the throughput.
3. Each parent in the multicast tree generates a number of replicas no larger than its degree in the overlay graph.
4. Each node in the multicast tree stores only an $i3$ trigger which is shared by all children. This trigger is refreshed by all children. An efficient algorithm that avoids refresh message implosions is given in [22].

Finding closest replica Many distributed file sharing systems, and content distribution systems require to find the best replica, where "best" generally means the "closest" in terms of delay. Consider m servers implementing the same service that are attached at the infrastructure nodes $n_1, \ldots n_s$. Then a client attached at node n_c can query the WS for the best paths between n_c and nodes $n_1, \ldots n_s$, and then chose the best path among these paths. Alternatively, to improve the efficiency, the WS may expose an interface that allows the client to query for the best path between a given node and a subset of other nodes.

7 Implementation Details

We have implemented a prototype of our system on top of $i3$ and deployed it in the PlanetLab testbed. In our implementation of $i3$, the length of IDs is 256 bits. Node IDs are chosen such that their 64-bit suffix is zero. This guarantees that all IDs with the same prefix are mapped on the same node. When returning a path, the WS identifies nodes only based on their 64-bit prefix. End hosts use these 64-bit IDs to construct the trigger IDs. Assume that the WS returns a path $(a_1, \ldots, a_n)$, where a_i is an 64-bit ID, to a host R. Host R will then insert triggers $(a_1|r_1, a_2|r_2), \ldots (a_{n-1}|r_{n-1}, a_n|r_n)$, where r_i is an 192-bit suffix chosen by the application, and "|" denotes the concatenation operator. Note that all triggers with IDs $a_i|r_i$ are stored at node a_i.

We have implemented a centralized WS and deployed it on a well-connected node in the network. Since the size of PlanetLab is relatively small (*i.e.,* about 110 nodes), this solution works well in practice. The WS employs the weighted random algorithm to compute the overlay graph. Currently the WS offers only the capability of finding the best path between two nodes in the infrastructure given corresponding to one of the following metrics: delay, loss, and available bandwidth. We expect that the API offered by the WS to evolve substantially as we gain more experience with applications that use the WS. For instance, WSes could expose a constrained routing primitive that allows the end-hosts to ask for a shortest delay path subject to available bandwidth constraints.

We have also instrumented $i3$ nodes to log all probe packets sent by the WS, and record in each probe packet the path followed, with the exact times when the packet traversed each hop. We use this information to compute the *actual* performance characteristics of the virtual links and use them to evaluate the accuracy of our estimation algorithms.

8 Experiments

In this section, we present the experimental evaluation of our design. All our experiments are conducted on the PlanetLab testbed which consists of about 110 machines located at over 40 locations in US, Europe, Asia and Australia. In some cases, all pairs of nodes did not have IP connectivity. The fact that PlanetLab machines are relatively well-connected is consistent with our design that assumes an overlay *infrastructure* rather than an end-host based overlay.

Section 8.1 evaluates the accuracy of our algorithms that estimate the virtual link characteristics (described in Section 4), Section 8.2 evaluates our path selection algorithms (presented Section 4) and the impact of estimation inaccuracies on path selection.

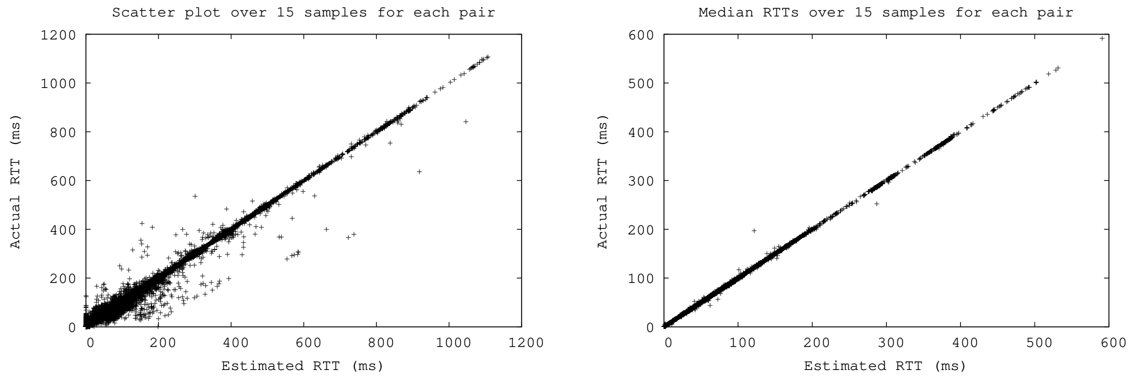
8.1 Virtual Link Experiments

In this section we evaluate the estimation algorithms described in Section 4. For each performance metric (*i.e.,* RTT, loss rate, available bandwidth, and bottleneck bandwidth) we compare the *actual* values to the values estimated by our algorithms. To compute the actual values we instrument all machines to perform pairwise measurements between them. In summary, our experiments show that our estimation algorithms are accurate particularly in the case of round-trip-time (RTT) and loss rate.

8.1.1 Round Trip Time (RTT)

Figure 6(a) shows the scatter plot of the estimated RTT versus the measured (actual) RTT between any two nodes in the infrastructure over an 150 sec time interval. Every virtual link is sampled every 10 sec, thus the plot contains around 15 samples per virtual link. Referring to Figure 2, recall that the estimated RTT is computed as the difference between the arrival time of copy m_1 and the arrival time of the original packet m at node R, while the actual RTT is computed as the time interval between sending and receiving copy m_1 at node n_1.

Figure 6(b) shows the scatter plot of the median values of the RTTs for the samples plotted in Figure 6(a). These results shows that our RTT estimation algorithm is very accurate. Out of a total of $90,000$ samples shown in Figure 6(a) less than 8% of samples have an error $> 10\%$. If we take the median among 15 consecutive samples, only 1.7% of the samples have a relative error $> 10\%$ (see Figure 6(b)). Most inaccuracies are due to estimating very low RTTs between machines on the same LAN. As a side note, the number of samples with RTTs larger than 600 ms is lower than 0.4%.



(a) (b)

Figure 6: The scatter plot of the actual and the measured RTT between two arbitrary IDs.

As expected these samples correspond to inter-continental links.

8.1.2 Loss Rate

Figure 7 shows the scatter plots of the actual versus the measured loss rates for up to 2250 pairs of nodes. To estimate the loss rate between two nodes we use the scheme described in Section 4.2. Each data point is the result of sending 1000 probes. In most cases the measured loss rates were quite small; only in 8% of cases we measured a loss rate larger than 2%.

Figures 7(a) and (c) show the loss rates for all links in the forward, and reverse directions. The points below the line $x = y$ are mainly due to false positives, *i.e.,* the WS wrongly decides that there was a loss on the monitored link. The points above $x = y$ are due to the fact that the WS ignores the probes for which it does not receive any response. As expected the estimation accuracy of the loss rate on the forward direction is better than the estimation accuracy of the loss rate on the reverse direction.

In both cases, the inaccuracies occur when the losses between the WS and the measure link are considerably larger than the loss rate on the virtual link. To verify this hypothesis we have identified the nodes that are responsible for the highest loss rates, and eliminate them from the measurements.[8] The remaining results are plotted in Figures 7(b) and (d), respectively. As expected, the estimation accuracy improves considerably especially on the reverse path. The estimation accuracy is 90% in 89% of the cases in the forward direction and 78% of the cases in the reverse direction.

We make two observations. First, as shown in Figures 7(b) and (d), we are more likely to overestimate than underestimate the loss rates (all points below $x = y$ represent overestimations). We do not expect over-estimations to be a serious problem in practice. If the WS over-estimates the loss rate on a particular link, the worst it can do is to not to return the best path to an application. However, in a network with rich connectivity we expect that the effects of such occasional sub-optimal paths to be minimal. Second, one can easily obtain better results for the reverse path by simply reversing the measurement setting for that links. The price to pay is doubling the overhead since now we have to send probes in both directions of the measured link.

8.1.3 Available Bandwidth

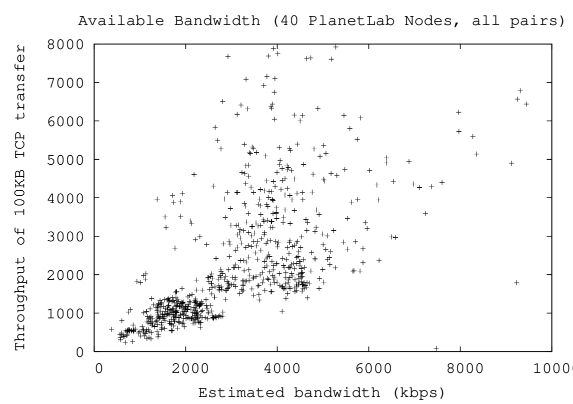


Figure 8: The scatter plot of the actual and the estimated available bandwidth.

To evaluate the technique presented in Section 4.3 for determining the available bandwidth, we chose 40 nodes, each at a different PlanetLab site [9]. Figure 8 shows the scatter plot of the estimated versus the actual available bandwidth for all pairs. The actual available bandwidth between any two nodes is measured by transferring an 100 KB file between the two nodes.

While the scatter plot in Figure 8 is quite spread out, we note that in 70% of pairs our estimates are within a factor of two of the actual values. We believe that these results are reasonable taking into account the fact that the available bandwidth

[8] During the experiments reported here we identified five such nodes: two at `cs.unibo.it`, two at `cuhk.edu.hk`, and one at `nbgisp.com`.

[9] Only one node per site was used to economize on the bandwidth that we use for our experiments

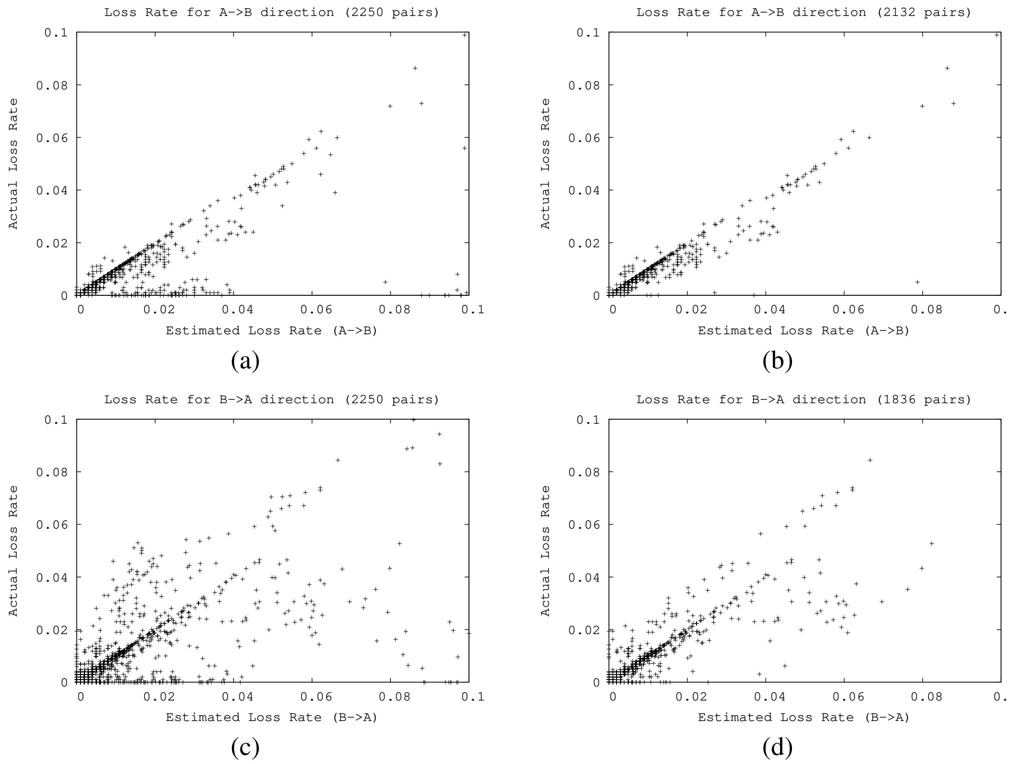


Figure 7: Scatter plots of the actual versus the measured loss rates: (a) forward path, (b) forward path after eliminating nodes that cause a high error rate, (c) reverse path for all links, (d) reverse path after eliminating nodes causing high error rates.

is usually a more dynamic phenomenon. Furthermore, we seriously under-estimate only in the cases when the available bandwidth is very large. This is because the WS limits the probing traffic to 100 KB in order to reduce the measurement overhead, and this is not enough to reach the TCP fair share.

8.1.4 Bottleneck Capacity

Using the algorithm presented in Section 4.3, we estimate the bottleneck capacity along a virtual link between two pairs of nodes. In the interest of space, we only summarize the results. We compare our results with direct measurements between any two nodes using the packet pair technique [20]. In most cases, the estimated bottleneck capacity was within 20% of the actual bottleneck capacity estimated by using the packet pair technique. However, when measuring the bottleneck capacity between two machines on the same LAN, our algorithm can under-estimate it by a factor of up to 2.5. In practice, bottleneck bandwidth estimates can be used only to distinguish between the capabilities of nodes at a coarse granularity (*e.g.,* between a $T1$ line and $T3$ line) since this metric does not give an indication of current congestion levels. Hence, we believe that our estimation techniques is accurate enough for practical purposes.

8.2 Unicast

In this section, we study the quality of the path in the graph maintained by the WS in terms of RTT and loss rate.

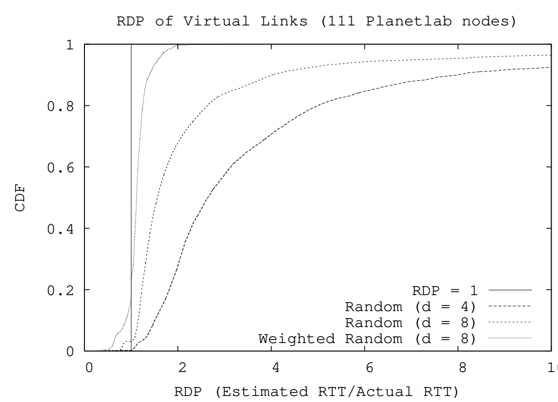


Figure 9: The cumulative distribution function (CDF) of the relative delay penalty (RDP) for all pairs of a 111 node network.

RTT Figure 9 plots the CDF of the relative delay penalty (RDP) between two arbitrary nodes in PlanetLab. The RDP between two nodes n_1 and n_2 is computed as the ration of (1) the lowest RTT path between n_1 and n_2 in the graph maintained by WS to (2) the RTT of the direct IP path between n_1 and n_2, respectively. For random generated graphs the RDP is quite large. When the average degree is 4 only 28% of pairs have an RDP value smaller than 2, while 7.5% of pairs have RDPs larger than 10. As expected, results improve when the degree increases. For a degree of 8, 68% of pairs have RDPs lower than 2. However, even in this case there are 3.5% pairs with RDPs larger than 10. The main cause for large RDPs is due to nodes that are very close to each other but which are not directly connected in the overlay graph.

The above problem is fixed by using weighted-random graphs. Indeed, even when the average degree of a weighted-random graph is 8 —*i.e.,* each node has links to two random nodes and to its three closest neighbors on average — more than 99.7% pairs have RDPs smaller than 2, and no pair has an RDP larger than 4. Furthermore, 13% of the pairs have RDPs smaller than one, *i.e.,* the latency of the path returned by WS is *smaller* than the latency of the direct IP path.

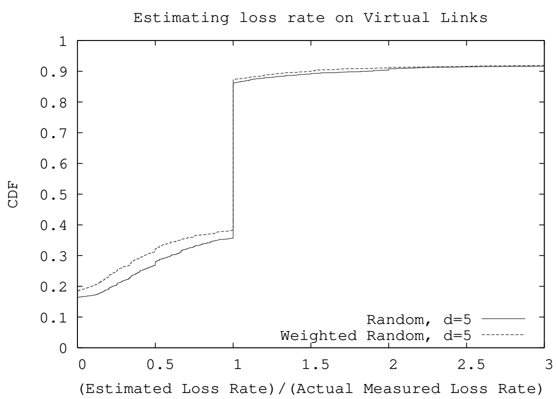


Figure 10: The cumulative distribution function (CDF) of the relative loss penalty (RLP) for all pairs of a 30 node network.

Loss Rate We compute the path with the lowest loss rate as the shortest path in the graph where the weight of each edge represents the loss rate of the corresponding virtual link. This assumes that losses experienced by different virtual links are not correlated. Figure 10 shows the CDF of relative loss penalty (RLP) (i.e. estimated loss rate as a fraction of actual loss rate) for all pairs for 30 nodes. The loss rates are computed over 1000 probes.

With a random degree of 5, the loss rate on 84% of the pairs was no worse than that on the underlying IP path between those pair of nodes, with 31% of the pairs getting paths with lower loss rates. In this case, weighted-random graphs produce marginal improvement over random graphs. This is because, in most cases, the loss rates are already very small and choosing low loss virtual links does not produce much improvement. However, in both cases, there are about 9% of the pairs that get paths with higher loss rates than in the underlying IP. This was predominantly due to inaccuracies in estimating the loss rate, which might arise if the loss rate between the WS and one of the nodes of the pair is much higher than the loss rate between the pair of nodes.

8.3 Multicast Example

A single source multicast tree built is merely a union of the unicast paths that the WS would return for each receiver. Using nodes on 37 PlanetLab sites, we built a single source low-delay multicast trees using the unicast paths that the WS returned. Figure 11 gives the tree that results with the source

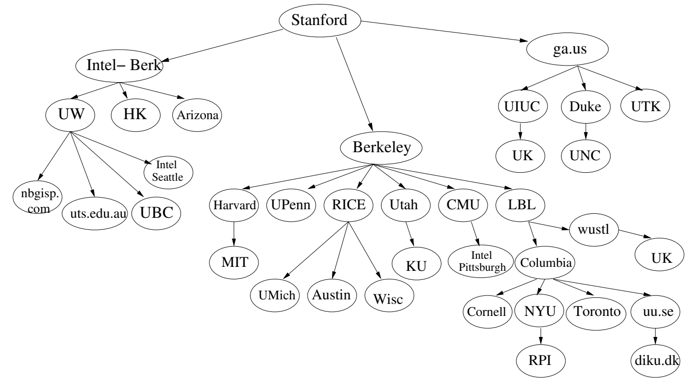


Figure 11: Delay-based multicast tree, with source at Stanford

at `planetlab-2.stanford.edu`. Since in most cases, the machines on the same site were adjacent in the multicast tree [10], we represent each site by one node in the graph. We make a few observations: (i) Link stress, i.e. the number of times a packet is replicated on a link, is small at almost all nodes (ii) The tree resembles the underlying geography to a good extent, for e.g., nodes in the vicinity of NY, i.e. Columbia, NYU, RPI, Cornell and Columbia are close together in the tree.

8.4 Overhead

In this section we evaluate the communication overhead incurred by the WS to maintain the performance map of the PlanetLab network consisting of 110 nodes, assuming the WS maintains an overlay graph with average degree 10.

The WS can use a single communication pattern, *i.e.,* the one presented in Figure 3, to measure both the RTT and the loss rate in both directions of a virtual link. To implement this communication primitive, the WS needs to maintain five triggers for each virtual link. For each probe, the WS can receive up to three replies back. Let T_l be the time period used to send a probe p, and let $T_{trigger}$ be the time period to refresh an $i3$ trigger. To maintain a graph with e edges, the WS has to send $e(1/T_l + 5/T_{triggers})$ refreshes/probes per second, and be able to receive $3e/T_l$ probe replies per second, where $T_{trigger} = 30$ sec. Since there are around $550 = 110 * 10/2$ edges in the overlay graph, and assuming that the WS probes every link each second ($T_l = 1$ sec), the WS needs to send 642 packets and receive 1650 packets per second. Since the length of an $i3$ refresh packet is 137 bytes and the length of a probe is 90 bytes (this includes the $i3$ header), the WS generates < 0.6 Mbps on the outgoing connection, and < 1.2 Mbps on the incoming connection.

The available bandwidth evaluation algorithm is more heavy weighted, as the WS needs to maintain $k + 2$ triggers, where k is the replication factor along the measured link. In our

[10] In fact, the WS can take this fact into account and return paths in which two machines on the same subnet have parent-child relationship.

implementation we bound $k = 10$, and bound the number of packets of one measurement to 100. Let $T_a = 3$ min be the time period of evaluating the bandwidth on each link. Since the probe packet in this case is 1 KB, the outgoing and incoming bandwidth required to estimate the available bandwidth of all virtual links is about 2.6 Mbps.

Finally, assuming that we monitor the bottleneck bandwidth on each link every 5 min, the measurement overhead on both incoming and outgoing connection is about 0.25 Mbps.

In summary, the overhead of maintaining the performance map of the current PlanetLab testbed (assuming an average degree of 10) is 3.6 Mbps for the outgoing bandwidth, and 4 Mbps for the incoming bandwidth. These figures are reasonable for a well connected machine. Alternatively, the measurements can be easily partitioned among different machines, thus reducing bandwidth requirements per machine.

9 Related Work

Jannotti has recently proposed two primitives, namely *path painting* and *path reflection*, that can significantly improve the efficiency of overlay networks [18]. Path painting allows two or more receivers to discover their common path from a given sender. Path reflection allows an end-host to ask a router to replicate the packets on its behalf. This is similar to our packet replication primitive except for the fact that a replica is not source routed. These two primitives are proposed in the context of IP and use the existing routing infrastructure. Thus, the goal of this work is different from ours; the primitives proposed by Jannotti enable overlays to use the existing IP routing infrastructure more efficiently, while our primitives are designed to allow hosts to directly control the routing of the packets.

ESP [7], a light-weight router-based building block, allows packets to create temporary state at routers via short, predefined computations. Though some applications have been shown to benefit from this, it cannot be directly used for the kind of services which require route selection based on application-sensitive metrics.

The Internet Indirection Infrastructure ($i3$) has been recently proposed to provide support for a rich set of communication primitives including mobility, anycast, multicast, and service composition [31]. $i3$ and the work we present in this paper are largely complementary. While $i3$ focuses on supporting basic communication primitives, we focus on supporting generic overlay applications. In fact, due to some desirable properties of $i3$ that we mention in Section 3, we choose $i3$ as an instantiation of our primitives.

The loose source routing option in IPv4 allows end-hosts to control the route of its packets by specifying a set of intermediate IP routers along the packet's route [26]. As we allude to in Section 3, loose source routing can implement the packet selection primitive. However, IP does not provide any primitive equivalent to packet replication.

Recently, efforts have been made to increase sharing among various overlays, the most notable example being PlanetLab [25]. PlanetLab's intent is to provide a common hardware infrastructure that can be shared among overlays during their initial testing and deployment. While PlanetLab is a perfect vehicle for research as it gives users complete control on overlay nodes, it raises significant security and efficiency challenges that make it inappropriate for commercial use. Furthermore, because each overlay runs in an independent *slice* of a PlanetLab machine, the sharing between overlays is mainly at the hardware level; there is little sharing of higher level design or functionality and thus each application has to re-implement the functionality it needs from scratch.

Dabek *et al* have proposed a common API for structured P2P overlay networks [9]. However, the proposed API assumes a design in which end-hosts need to run their own code on the infrastructure nodes to control routing.

Several measurement and monitoring systems have been recently proposed [23, 13, 16]. While these systems are more general in that they aim to estimate performance characteristics between any two hosts in the Internet (instead of between overlay nodes) they usually have one or more of the following limitations: (i) limited to estimating the path latency only, (ii) require to deploy their own infrastructure [13].

There is a large body of literature on algorithms and techniques to estimate the characteristics of IP paths such as latency, available bandwidth and bottleneck bandwidth [10, 21, 17]. While some of the estimation algorithms presented in this paper are similar in spirit (*e.g.,* bottleneck bandwidth estimation), we also leverage the packet replication primitive which is not available in IP.

10 Conclusions and Future Work

In this paper, we advocate a shared overlay infrastructure that exports two primitives. We also show how a network weather service (*NEWS*) that maintains a map of the entire infrastructure can be built using the primitives. The primitives, along with the weather service primitives enables a large variety of overlay applications, including adaptive routing, multicast, and coarse grained data manipulations such as transcoding. At the heart of our design lie three crucial decisions:

Delegate routing decisions to applications: This design decision can be viewed as an application of the end-to-end arguments [29] to routing. Applications know their performance and robustness requirements best, and thus they are in the best position to select the routes for their traffic.

Delegate performance measurements to (third-party) applications: This decision can be again viewed as a application of the end-to-end arguments to monitoring. Since the measurements are not embedded in the infrastructure, it allows applications to evolve the measurement algorithms to best suit their needs, or to support new applications, without changing the infrastructure.

Minimalist infrastructure functionality: One of our research goals in this paper was to identify a minimal set of primitives that can support many of today's overlay applications. After gaining a better understanding, a next step would be to consider other primitives such as QoS.

We do not claim to have a complete solution that when plugged into the Internet would solve all the problems; we are exploring what seems to be an interesting space of problems. We believe that this iterative process of experimentation and deployment would help us understand the needs and challenges better. We hope that the ideas in the paper, a first step towards a grand vision, would evolve enough for us to indeed realize our vision.

Current status and Future work

We have implemented *NEWS* over $i3$ and deployed it on PlanetLab. A preliminary evaluation of our techniques has yielded promising results. We are working on developing an API for people to develop applications using $i3$ and *NEWS* . We are also implementing a tool for visualizing the data that the *NEWS* provider captures. Using this, we are developing a multicast application which builds the topology based on the metrics specified. We are also studying how the measurement results can be improved by measuring from multiple vantage points. Another problem we are looking at is to determine the placement of the *NEWS* agents in the wide-area.

References

[1] `http://www.cs.washington.edu/homes/gummadi/icir/generating_topologies/`.

[2] D. Adkins, K. Lakshminarayanan, A. Perrig, and I. Stoica. Towards a More Functional and Secure Network Infrastructure. Technical Report UCB/CSD-03-1242, UC Berkeley, 2003.

[3] Akamai Technologies. `http://www.akamai.com`.

[4] D. Anderson, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, Banff, Canada, Oct. 2001.

[5] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *SIGCOMM*, pages 24–35, 1994.

[6] R. Callon, P. Doolan, N. Feldman, A. Fredette, G. Swallow, and A. Viswanathan. A Framework for Multiprotocol Label Switching, Nov. 1997. Internet Draft, draft-ietf-mpls-framework-02.txt.

[7] K. L. Calvert, J. Griffioen, and S. Wen. Lightweight Network Support for Scalable End-to-End Services. In *Proceedings of ACM SIGCOMM*, Pittsburgh, PA, 2002.

[8] Y. Chawathe, S. McCanne, and E. Brewer. An Architecture for Internet Content Distribution as an Infrastructure Service, 2000.

[9] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. Towards a Common API for Structured Peer-to-Peer Overlays. In *Proceedings of IPTPS'2003*, Berkeley, CA, feb 2003.

[10] A. B. Downey. Using pathchar to Estimate Link Characteristics. In *Proceedings of SIGCOMM'99*, pages 241–250, 1999.

[11] Fastforward Networks. `http://www.ffnet.com`.

[12] P. Francis. Yoid: Extending the Internet Multicast Architecture, 2000.

[13] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM ToN*, October 2001.

[14] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services, 2000.

[15] Georgia Tech Internet Topology Model. `http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html`.

[16] K. P. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In *Proceedings of IMW*, Marseille, France, November 2002.

[17] M. Jain and C. Dovrolis. End-to-End Available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput. In *Proc. ACM SIGCOMM*, 2002.

[18] J. Jannotti. *Network Layer Support for Overlay Networks*. PhD thesis, MIT, August 2002.

[19] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. J. W. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of OSDI 2000*, pages 197–212, San Diego, California, October 2000.

[20] S. Keshav. A Control-Theoretic Approach to Flow Control. *Proceedings of the Conference on Communications Architecture and Protocols*, pages 3–15, 1993.

[21] K. Lai and M. Baker. Measuring Bandwidth. In *Proceedings of INFOCOM'99*, pages 235–245, 1999.

[22] K. Lakshminarayanan, A. Rao, I. Stoica, and S. Shenker. Flexible and Robust Large Scale Multicast using *i3*. Technical Report CS-02-1187, University of California - Berkeley, 2002.

[23] T. S. E. Ng and H. Zhang. Global Network Positioning: A New Approach to Network Distance Prediction. *CCR*, 2001.

[24] D. Pendarakis, S. Shi, D. Verma, and M. Waldvogel. ALMI: An Application level Multicast Infrastructure, 2001.

[25] Planet Lab. `http://www.planet-lab.org`.

[26] J. Postel. Internet Protocol, Sept. 1981. Internet RFC 791.

[27] B. Raman and R. H. Katz. An Architecture for Highly Available Wide-Area Service Composition. *Computer Communications Journal, special issue on "Recent Advances in Communication Networking*, May 2003.

[28] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The Design of a Large-Scale Event Notification Infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[29] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[30] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A Case for Informed Internet Routing and Transport. Technical Report TR-98-10-05, 1998.

[31] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proc. of ACM SIGCOMM*, 2002.

[32] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: Offering QoS using Overlays. In *Proceedings of HotNets'2002*, Princeton, NJ, oct 2002.

[33] X. Xu, A. Myers, H. Zhang, and R. Yavatkar. Resilient Multicast Support for Continuous-media Applications. 1997.

[34] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *NOSSDAV*, 2001.