A Routing Underlay for Overlay Networks

Akihiro Nakao, Larry Peterson and Andy Bavier
Department of Computer Science
Princeton University

ABSTRACT

We argue that designing overlay services to independently probe the Internet—with the goal of making informed application-specific routing decisions—is an untenable strategy. Instead, we propose a shared routing underlay that overlay services query. We posit that this underlay must adhere to two high-level principles. First, it must take cost (in terms of network probes) into account. Second, it must be layered so that specialized routing services can be built from a set of basic primitives. These principles lead to an underlay design where lower layers expose large-scale, coarse-grained static information already collected by the network, and upper layers perform more frequent probes over a narrow set of nodes. This paper proposes a set of primitive operations and three library routing services that can be built on top of them, and describes how such libraries could be useful to overlay services.

1. INTRODUCTION

Overlays are increasingly being used to deploy network services that cannot practically be embedded directly in the underlying Internet [21, 22]. Examples include file sharing and network-embedded storage [18, 27, 13], content distribution networks [34], routing and multicast overlays [28, 9, 12], QoS overlays [33], scalable object location [10, 26, 32, 24], and scalable event propagation [14].

One common characteristic of these overlay services is that they implement an application-specific routing strategy. For example, object location systems construct logical topologies using distributed hash tables, multicast overlays build distribution trees that minimize link usage, and robust routing overlays attempt to find alternatives to Internet-provided routes. These overlays often probe the Internet, for example using ping and traceroute, in an effort to learn something about the underlying topology, thereby allowing them to construct more efficient overlay topologies. Some overlays also employ active measurement techniques in an effort to continuously monitor dynamic attributes like bandwidth and loss. Even overlays that construct purely logical topologies probe the Internet to select logical neighbors that are also physically nearby.

While having a single overlay probe the Internet in an attempt to discover its topology is not necessarily a problem, the strategy is not likely to scale. This is for two fundamental reasons. First, aggressive probing mechanisms that monitor dynamic attributes do not scale in the number of nodes that participate in the overlay. For example, the designers of RON [9] indicate that their approach does not scale beyond roughly 50 nodes. Second, when multiple overlays run on a single node (as is the case in an overlay-hosting platform like PlanetLab [22]) or on the same subnet (as might be the case if a site participates in multiple overlay services) it is not uncommon to see a measurable fraction of the traffic generated by a node being ping. On PlanetLab, for example, we recently measured 1GB-per-day of ping traffic (outbound only), corresponding to a little over one ping per second per node across approximately 125 nodes. Although it is difficult to quantify how many concurrent overlays a network could support—or what percentage of overall traffic should be allowed to be ping—having every overlay independently probe the network is difficult to defend architecturally.

The advantage that broad-coverage services like the ones cited above gain over traditional client-server applications is that, by being geographically distributed over the world, they have multiple vantage points of the network from which they are able to construct application-specific packet forwarding strategies. It should not be forgotten, however, that the network itself already has the advantage of these multiple viewpoints, and already has a fairly complete picture of the network. It is redundant for a single overlay network to re-discover this information for itself. It is architecturally silly for each overlay to duplicate this effort.

In response to this problem, we propose a new architectural element—a *routing underlay*—that sits between overlay networks and the underlying Internet. Overlay networks query the routing underlay when making application-specific routing decisions. The underlay, in turn, extracts and aggregates topology information from the underlying Internet. In addition to making a case for the routing underlay, this paper sketches one possible underlay design and evaluates its feasibility.

2. ARCHITECTURE

This section motivates our routing underlay architecture by first observing that many existing overlays could be implemented on top of a shared set of topology discovery services. It does not attempt to define a comprehensive set of services, sufficient to support all overlays, but rather, it identifies the kinds of high-level operations that overlays might effectively employ. The section then proposes a set of low-level primitives that an underlay would need to support to implement these operations. The section concludes by sketching a layered architecture suggested by this discussion.

2.1 Useful Services

Looking at the problem from the top-down, we observe that many

SIGCOMM'03, August 25–29, 2003, Karlsruhe, Germany.

recently proposed overlay services use similar approaches to topology discovery and self-organization, and for this reason, could benefit from a shared routing underlay. Such an underlay might also help some overlays take more scalable approaches to resource discovery. Below, we discuss a few representative overlays and identify some underlay services that they could exploit.

The RON routing overlay [9] aims to discover good-quality paths through an overlay of routing nodes, as well as to quickly fail-over to an alternate path when the current path goes down or becomes congested. RON organizes the N participating nodes into a clique and probes each of the N^2 edges to discover its latency; it then runs a link-state routing algorithm over the fully-connected logical topology to discover the lowest-cost routes through the overlay. However, the authors report that this approach does not scale for $N > 50$ nodes due to the amount of probes generated. A routing underlay could help the situation by providing RON with a sparsely connected *routing mesh* of overlay nodes; probing a mesh instead of a clique could reduce the total probing cost by an order of N. Alternatively, RON could be redesigned to probe the network at connection setup time, that is, on-demand. The underlay could return some (constant) number of *disjoint paths* through the network between the ingress and egress RON nodes for the connection, and RON could probe just these paths to select the best one. Since the paths would be disjoint, their performance should be independent; periodic probes of each path would allow RON to switch paths should the performance of one decline.

An end-system multicast (ESM) overlay [12, 11] organizes endhosts into a mesh and then runs a minimum spanning tree algorithm on the mesh to produce a multicast tree. ESM would benefit from an underlay operation that allowed it to find the overlay nodes that are the *nearest neighbors* to a given node prior to building the mesh. Alternatively, the underlay could provide a ready-built routing mesh based on knowledge of the underlying network topology, with ESM then pruning this mesh if necessary before running the MST algorithm. Nodes in peer-to-peer systems like file sharing networks are also interested in finding their nearest neighbors in order to peer with them. Conversely, a fault-tolerant P2P file system may want to find a far-away neighbor for data replication, in order to ensure that local disasters do not affect all copies.

The three candidate underlay services that we have suggested—finding the nearest neighbors to a node, finding disjoint paths between two nodes, and building a routing mesh—are by no means the only ones that could be shared among a wide number of overlays. However, we believe that they represent an interesting initial set from which to explore the space.

2.2 Topology Discovery

Looking at the problem from the bottom-up, we ask ourselves what can be known about the underlying network topology. This question is immediately complicated by the fact that the underlying network topology is often nested, with a link at one level actually implemented by a multi-link path at another level. It is likely that different overlays will need to see the topology at different *resolutions*. For example, one overlay might be satisfied to know that two nodes are connected to different autonomous systems (AS), while another overlay might need to know that two physical links between a pair of nodes do not go through the same harbor tunnel.

Although we would like to claim that we can justify a set of primitives for discovering network topology from first principles, the truth is that we can only propose an initial candidate set based on our experiences to-date. This set is also influenced by what we understand how to implement using the raw topology information already available in the underlying Internet. With these caveats in mind, we propose three primitives that the routing underlay should support:

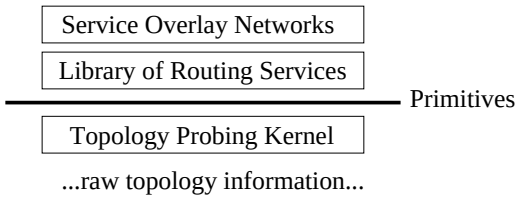


Figure 1: Structure of the Routing Underlay

- it should provide a graph of the known network connectivity at a specified resolution (e.g, ASes, routers, physical links) and scope (e.g., the Internet, some AS, everything within a radius of N hops);
- it should expose the actual route (path) a packet follows from one point to another, again at a specified resolution (e.g., a sequence of ASes, routers, or links);
- it should report topological facts about specific paths between a pair of points, according to a specified metric (e.g., AS hops, router hops, measured latency).

Keep in mind that these primitives represent an ideal interface; a real underlay implementation may not be able to provide all of it. Some information may be unavailable or incomplete. For example, the administrator of a particular domain may be unwilling to provide a feed from their BGP router to our underlay, and so the underlay cannot have knowledge of AS-level paths originating in that domain. Similarly, a carrier might not be willing to reveal the fact that seemingly independent links are carried in the same bundle. We could attempt to collect this information for ourselves, for example using network tomography [30, 19, 29], but such techniques are designed to drive simulation models rather than enable routing decisions, and may be too expensive. We have to deal with the reality that gathering more accurate topological information comes with a cost.

2.3 Layered Routing

Combining these top-down and bottom-up perspectives, we propose a layered approach to constructing the routing underlay. The primary feature of this approach is cost-consciousness: our underlay uses infrequent probes of the entire network at lower layers, and higher layers reduce the scope of the probes while increasing their frequency. Our hope is that a service overlay making application-specific routing decisions using our underlay will consider only a small, fixed-sized subset of the total set of overlay nodes. Typically, this means that lower layers use static information about the network (i.e., that needs to be probed infrequently) while upper layers probe dynamically changing network conditions.

Figure 1 depicts the structure of our routing underlay, which we summarize as follows:

- The bottom-most layer, which we label the *topology probing kernel*, provides the underlay primitives described in Section 2.2 using the raw topology information that is already available in, or can be directly extracted from, the underlying Internet. This layer hides the fact that probes are sent to remote sites, and it caches the results of previous probes. Note that while there is a cost to retrieve raw topology information from a *remote* site, we assume that accessing this

information from within the local site is free since the information is already being collected as part of the Internet's normal operation.

- The second level provides a *library of routing services*. These services answer higher-order questions about the overlay itself, such as those discussed in Section 2.1, using the primitives exported by the topology probing layer. Our prototype library uses heuristics that are inexpensive in terms of probes, robust in the face of incomplete information provided by the primitives, and produce good results. In other words, the topology probing layer provides the small set of "facts" that the underlay has learned about the Internet's topology, while the routing services library represents the higher-level "conclusions" one might derive based up on those facts. Of course a wide range of library services are possible to support the special needs of different overlay networks.

- The overlay services themselves represent the top-most layer. They are primarily distinguished from library services in that they are typically used directly by application programs rather than by other services.

Note that we have used the terms "kernel" and "library" to denote the fact that we assume a single topology probing layer running on each node (thus sharing state across a set of overlays), while we expect different library services to be linked into each overlay (thus not sharing state between overlays). There may be value in sharing topology library state among overlays, but we do not pursue this question any further in this paper.

The interesting question is whether we can actually enforce the use of the routing underlay, or if overlay applications will bypass the topology discovery primitives and directly probe the Internet themselves. On an infrastructure-based overlay like PlanetLab, we can enforce use of the underlay by implementing the probing layer in the OS kernel, and in doing so, pace the rate and limit the range of various probes. For pure end-system overlays, universal kernel enforcement is not likely, so we must fall back to two incentive-based arguments. First, the services offered by the routing underlay are so convenient that application writers will choose to use them, implying that the primary reason they currently use ping and traceroute is that these are the only tools available. Second, to the extent excessive probing traffic becomes a widely-recognized problem, the same social pressures that encourage the use of TCP-friendly congestion control will encourage the use of a sane routing underlay. In fact, this "encouragement" is likely to take the form of ISPs and network administrators blocking ping and traceroute traffic in an effort to limit excess probing. We can only hope to introduce an acceptable probing layer into the architecture before this happens.

3. TOPOLOGY PROBING KERNEL

The topology probing layer supports a set of primitive operations that report connectivity information about the Internet. We envision these primitives being supported on every overlay node. Much of the information exported by this layer is at the granularity of autonomous systems (AS), and is relatively static. This section identifies three such primitives, briefly sketches their implementation, and discusses their costs.

For the sake of this discussion, we assume each overlay node has access to the BGP routing table at a nearby BGP router. For multi-homed sites, a BGP speaker within the site could provide this routing table. For single-homed sites, we assume the routing table is retrieved from the site's ISP since the local table is likely to be uninteresting. Both are possible by configuring the router to treat the overlay node as a read-only peer. The relevance of the site being multi-homed (as opposed to single-homed) is that the routing table on a multi-homed router contains roughly 120k prefixes, spanning the roughly 15k autonomous systems in the Internet. This table, therefore, can be treated as a source-rooted tree of autonomous system (AS) paths to every other AS in the Internet.

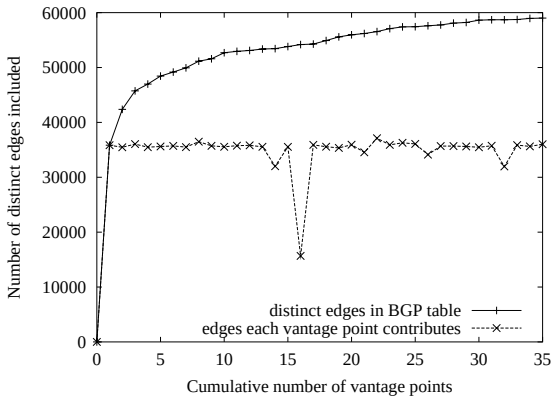


Figure 2: Number of distinct edges in PG as a function of vantage points, using RouteViews data set.

3.1 Peering Graph

The first primitive returns the *peering graph* (PG) for the Internet.

```
PG = GetGraph()
```

This graph represents the coarse-grain (AS-level) connectivity of the Internet, where each vertex in PG corresponds to an AS, and each edge represents a peering relationship between ASes. The Internet does not currently publish the complete PG, but it is easy to construct an approximation of the PG by aggregating BGP routing tables from multiple vantage points in the network, as is currently done by sites like RouteViews [6] and FixedOrbit [1]. That is, an edge exists between any two vertices X and Y in PG if some BGP routing table contains a path in which ASes X and Y are adjacent.

Given access to a modest number of BGP routing tables, this approximation contains nearly all the vertices (ASes) but is likely to be incomplete in the number of edges (peering relationships) it contains. For example, RouteViews aggregates BGP tables from 64 sites. One snapshot of BGP tables obtained from RouteViews contains roughly 120k routes, from which we are able to produce a PG with 14,381 vertices and 59,988 edges (29,944 bi-directional edges). Since a BGP table contains downstream AS paths—i.e. AS paths from the local router to arbitrary destinations—our strategy to construct PG is to add bi-directional edges between X and Y whenever we detect a path in either direction.

Although we could implement the GetGraph primitive by having all the overlay nodes send the BGP table they acquire to a centralized aggregation point (similar to RouteViews), and then download the result, it is possible for each overlay node to construct its own version of PG independently, simply by exchanging its PG with a small set of neighbors. As suggested by Figure 2—coupled with our knowledge of the full RouteViews data set and the total number of ASes in the Internet—aggregating BGP routing information from 30-35 vantage points results in a fairly complete peering graph. This argues that the PG can be constructed using a fixed

number of probes, independent of the number of overlay nodes in the network. In addition, since the PG is only an approximation and peering relationships change infrequently, this exchange can be done with very low frequency, on the order of once a week.

3.2 Path Probe

A path between a pair of nodes in the PG represents a possible route that packets might traverse from one AS to another, but only one such path is actually selected by BGP routers. The second primitive

Path = GetPath(src, dst)

returns the *verified AS path* traversed by packets sent from IP address src to IP address dst. Note that this primitive maps a pair of network prefixes to the sequence of AS numbers that connect them, much like a BGP routing table maps a network prefix to an AS path. Also, as will be seen shortly, we must limit the src and dst to addresses of overlay nodes because we need a point-of-presence within an AS in order to resolve this query.

Determining the actual AS path takes advantage of the BGP table we assume is exported by the underlying Internet. A node consults the local BGP table to answer queries for the case where src resides in the local AS. In case the local AS is so large that it has several BGP routers that use different BGP tables, src needs to be resolved into clusters sharing the same BGP table. For a src that does not reside in the local AS, the node forwards the query to an overlay node in the corresponding AS that contains src. This has the cost of a single probe. The local node also caches the reply for subsequent requests.

In both cases, we need to translate IP addresses to AS numbers. This can be done by selecting the route in the BGP table that matches a given IP address, and then inferring that the last AS number on the path in that route is the AS that contains the node with the given address.

3.3 Distance Probe

The final primitive reports the distance from the local node to some remote node target:

Distance = GetDistance(target, metric)

This query can report the latency using one of three metrics. First, based on the locally available BGP table it can respond with the number of AS hops from the local node to the remote node. Counting AS hops is weakly correlated with actual latency [20], but it may also be appropriate for applications that want to minimize peering points traversed. Plus, it can be implemented at no cost. Second, the local node can run traceroute to the target node (or consult a Rocketfuel-generated network map) and report the number of router hops. The value in doing this is that router hop count is more strongly correlated with latency than is AS hop count. Third, the local node can ping the target node, and return the corresponding round-trip time. The last two operations have the cost of one probe, although the result can be cached and used to respond to subsequent queries. Thus, this primitive is useful for discriminating among a set of nodes, but it may not be suitable for measuring the instantaneous round-trip time.

3.4 Remarks

Note that the GetDistance primitive is parameterized to reflect the resolution (accuracy) of the desired response: AS-hop-count, router-hop-count, or RTT. Similar generalizations are also possible for the GetGraph and GetPath primitives. For example, GetGraph could be parameterized by both *resolution* (possible values are AS-level, router-level, and physical-level) and *scope* (possible values are root, AS, and network). Our prototype implementation supports only the AS-level resolution and root scope, although an implementation that exploited ISP mapping tools like Rocketfuel might be able to do better [29].

Similarly, GetPath could be parameterized by *resolution*, with the same possible values as for GetGraph. In this case, the kernel might take advantage of traceroute to implement the router-level resolution. There is currently no good way to implement GetPath with resolution at the physical-level.

The rest of this paper assumes AS-level resolution, with the goal of understanding how much can be accomplished by exploiting the BGP information already collected by the Internet.

4. LIBRARY OF ROUTING SERVICES

This section suggests three library services that can be built on top of the topology probing layer. These services should be interpreted as representative examples, not a complete set. In particular, we elect to focus on services that can be provided at relatively low cost, where the interesting question to ask is how much information they provide at little cost. We believe that these services provide a useful foundation for a variety of overlay networks, as described in Section 2.1.

4.1 Finding Disjoint Paths

Our first routing service finds AS paths between two nodes that do not share a peering point with the default Internet route between the nodes. More specifically,

PathSet = DisjointPaths(u, v, N, k)

for a given pair of overlay nodes u and v and a set of candidate intermediate nodes N, the service returns k paths between u and v that (1) are edge-disjoint with respect to the default AS path between nodes u and v, and (2) pass through one of the intermediate nodes in N. We refer to these k paths simply as *disjoint* paths. Disjoint paths can provide both resilience and performance to higher-level overlay services, since it is unlikely that a disjoint path would share a bottleneck or a single point of failure with the default AS path. Additionally, paths involving a smaller number of AS hops should be more resilient to BGP failure or route changes so our service favors these paths over longer ones.

Note that, consistent with the results reported in [9], our experience is that single hop indirection gives us a good opportunity to find disjoint paths without complicating both disjoint path search and actual routing. That is, using multiple intermediate nodes almost always resulted in longer paths.

4.1.1 Implementation

Our service implementation involves three phases: inferring AS hop counts for indirect paths, trimming indirect paths that are not disjoint based on local information, and querying remote nodes to verify the inferences. Our goal is to minimize the number of queries to a small subset of candidate nodes based on the results of earlier phases.

First, we use the graph returned by GetGraph to guess the shortest disjoint paths (u, w, v), i.e., the paths from u to v through some node w such that $w \in N$. We assume that the route chosen by BGP always has the shortest number of AS hops. This is not always true due to the administrative policy of each BGP router [25], but we will correct for this later. However, we do make up a little for it at PG level, by excluding *illegal* paths using peering relationship inference [15], which basically says a *customer* AS does not carry

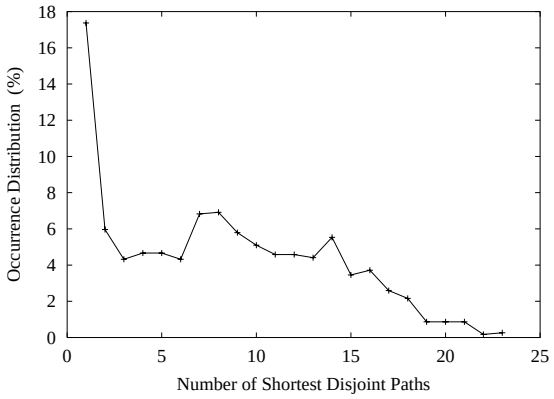


Figure 3: Distribution of shortest disjoint paths

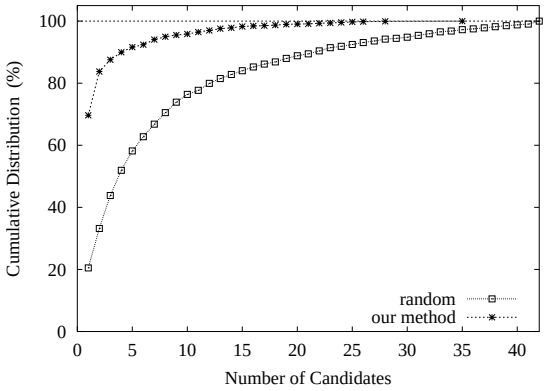


Figure 4: Cumulative distribution of disjoint paths found as a function of the number of probes.

traffic for its *provider* AS. For each node $w \in N$, we concatenate the AS sequences from the shortest AS path (u, w) and the shortest AS path (w, v) into the path (u, w, v). We sort the list of these composite paths by the AS count.

Second, we discard any intermediate nodes w for which we know that the path (u, w) is not edge-disjoint from the default AS path (u, v). We use the GetPath primitive to ensure that the sequence of ASes on the AS path (u, v) does not share two consecutive ASes (and hence a peering point) with the AS path (u, w); otherwise we drop the path from consideration.

Finally, we verify our inference about the path (w, v) by invoking GetPath(w, v). Since node w has access to its local BGP information, it can return the actual AS path (w, v) in response to this query. At this point we know all of the AS hops along the path (u, w, v), and can verify our inference about the AS sequence along the path, as well as discard all w such that (w, v) and (u, v) share a peering point.

Note that the algorithm just described selects k paths from the candidate set based on AS hop count. One could further discriminate among (or order) the candidates based on actual round-trip latency, that is, using the GetDistance with the RTT metric. This could be done at a cost of k probes.

4.1.2 Evaluation

We first try to get a feel for how many disjoint paths exist between two ASes using RouteViews [6], which approximates a fully implemented GetGraph primitive. One observation is that there is no disjoint path available if either of two end ASes are *single-homed*, i.e peer with only a single ingress/egress AS. Therefore, in our analysis we consider only multi-homed ASes.

The RouteViews router listens to BGP updates from 64 vantage points and stores them in its own BGP table. This means that the RouteViews BGP table should contain 64 entries for each network prefix. We use the NextHop field of each entry to match up entries with actual vantage points, and thus extract the local BGP table at that vantage point. This enables us to construct a clique of AS paths between 42 vantage points (the remaining 22 had incomplete BGP tables). For each pair of vantage points u and v, we then construct indirect paths (u, w, v) through every other vantage point w and check whether these paths are edge-disjoint with the direct path (u, v).

Among 1722 (=42×41) possible direct paths, we discard 374 paths due to inconsistencies, and another 113 because one of the endpoints is in a single-homed AS. Among the remaining 1235 direct paths, we find that 1157 (93.7%) have at least one disjoint path through an intermediate node.

Next, we evaluate how often our heuristic can find a disjoint path with the same number of AS hops as the direct path. Most of the direct paths have several such shortest disjoint paths, as shown by Figure 3. For instance, 17.4% have one shortest disjoint path, 6.0% have two, and so on. Figure 4 shows the cumulative distribution of direct paths for which our heuristic finds *at least one* of its shortest disjoint paths within a given number of queries. The plot compares our heuristics and a random scheme where we randomly pick a node and examine if that gives us a shortest disjoint path. As Figure 4 shows, we find a shortest disjoint path for 90% of the direct paths for which one exists within 5 probes.

4.2 Finding Nearest Neighbors

Our second library service finds the nearest overlay nodes in terms of distance:

Nodes = NearestNodes(N, k)

Relative to the local overlay node and a given set of candidate neighbor nodes N, this library returns k nodes in N that are closest to the local node, while minimizing the number of probes.

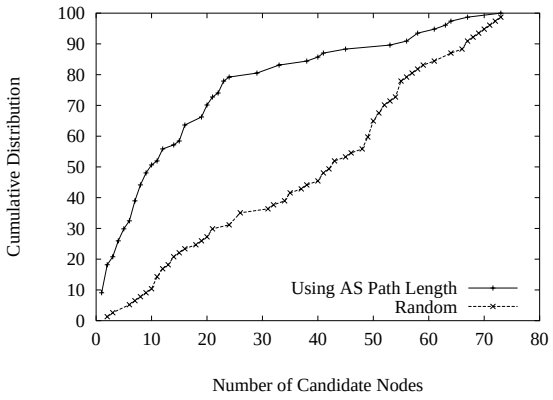
4.2.1 Implementation

The implementation of this service is relatively simple. As in the disjoint path service, we use information from the peering graph to narrow down the set of potential candidates before actively probing the network. First, we use GetPath to sort the list of candidate nodes by increasing number of hops from the source; this has no cost. It has been observed that latency and AS hop count are correlated [20], although with high variance, so we expect that nodes near the top of the list should enjoy better latency from the source. Next, we refine the result by invoking GetDistance on the top j nodes in the list, where $j \geq k$, and we choose the k with the lowest latency. The key is choosing the right value of j to send out as few probes as possible but still get a good result for k.

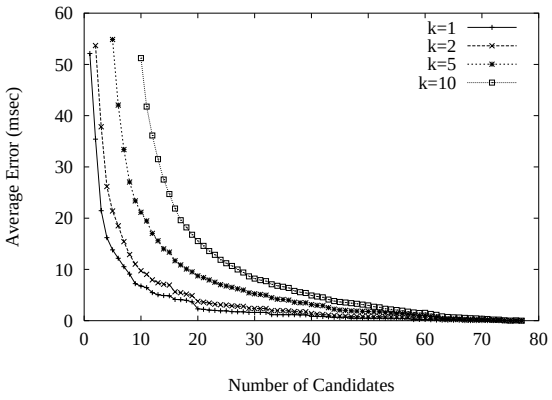
4.2.2 Evaluation

We evaluate the heuristic using 81 nodes (34 PlanetLab [22] nodes and 47 randomly selected public traceroute servers [17]). We also suggest a method for choosing the value j, the number of nodes to probe to find the best k.

First, we compare our heuristic against random guessing when trying to find the neighbor with the smallest latency. Figure 5(a)



(a) Number of probes required to find smallest latency neighbor (k=1)



(b) Absolute average error (msec) after a given number of probes

Figure 5: Evaluation of heuristic to find the k smallest-latency neighbors

shows the number of candidates j that were probed before finding the one that had the absolute lowest latency (i.e., $k = 1$), for both our heuristic and a random solution. Not surprisingly, we see that our method performs significantly better than random guessing. For example, about 50% of the nodes were able to find their optimal neighbor within 10 probes using our method, while random guessing requires 40 probes on average to achieve the same result. [1]

Second, we evaluate the quality of the result returned by the heuristic after probing j candidates and returning the top k. Figure 5(b) presents the results for $k = 1, 2, 5$ and 10. We note that even when our heuristic does not return the k best nodes, usually it can find k neighbors that have close to optimal latency with a small number of probes. For instance, to find k neighbors within 10 msec of the optimal latency required 7 probes on average for $k = 1$, 11 probes for $k = 2$, 18 probes for $k = 5$, and 27 probes for $k = 10$. Based on these observations, it appears possible to implement a table within our service that could translate the number of desired neighbors k and the error tolerance e to the number of probes j that the service needs to perform. More investigation is required to confirm this intuition.

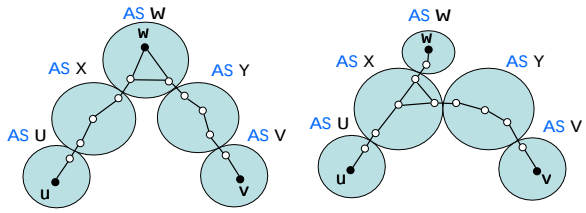
4.3 Building a Representative Mesh

Our third example routing service constructs a mesh that is physically representative of the underlying Internet, but with far fewer edges than the fully connected Internet allows. Specifically,

Mesh = BuildMesh(N)

Given a set of overlay nodes N, the BuildMesh call returns the local node's neighbor set in a mesh. Our mesh-building strategy is to identify and remove topologically redundant edges (virtual links) between overlay nodes, or said another way, retain only those edges that we can determine to be independent in the underlying physical network. Each overlay node performs this analysis independently using the GetPath primitive. The entire mesh could be formed by aggregating the neighbor sets; however, many routing overlays, such as RON and ESM, maintain only immediate neighbor sets at each node.

[1] The CDF curves reach 100% at less than 80 nodes because we had to exclude some paths due to invalid traceroute results.



(a) Topology A (b) Topology B

Figure 6: Black dots u, v, and w denote overlay nodes and the white dots denote routers. Virtual link (u, v) is redundant and can be removed from the mesh, since edges (u, w) and (w, v) connect u to v.

Our approach is limited to edges that can be removed without building a global picture of the network. An alternative strategy would be for a central authority to collect global network information, build the entire mesh, and distribute it throughout the overlay. We opt for a localized approach for reasons of scalability and cost, though the resulting mesh may not be as sparse as that produced by a centralized algorithm. One could imagine building a more sparse mesh on top of the one we build, for example by lowering the degree of each node by discriminating among the edges according to latency or throughput. Doing so would require a distributed algorithm to ensure that the resulting mesh does not become partitioned.

4.3.1 *Implementation*

Our algorithm prunes an edge from the local node u to remote node v if the AS path from u to v includes AS W, such that there is a node $w \in N$ that is located within AS W. This scenario is illustrated in Figure 6(a). We call the pruned edge (u, v) a *virtualized* edge. The simplest version of our algorithm prunes an edge (u, v), only when we find an intermediate node w such that both (u, w) and (w, v) are not virtualized *before pruning*. Note that these edges may become virtualized later by pruning other edges during the course of the algorithm. In order to examine this condition, the local node u must keep track of which edges are virtualized to verify the first half of the spliced connection (u, w) and also to answer

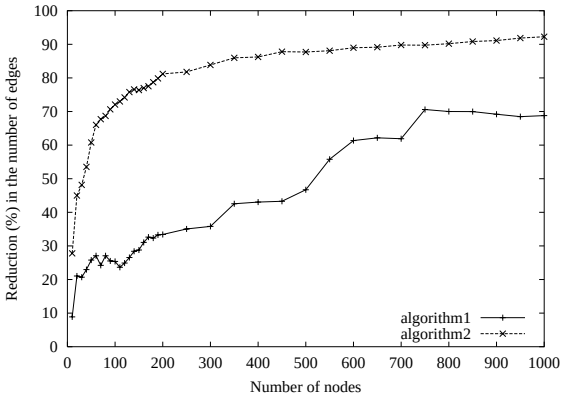


Figure 7: Reduction in the total number of edges in the peering graph after pruning

the queries for this local information from others. For example, u needs to ask the intermediate node w about the virtualization of the second half (w, v), which is local to the node w. Therefore, this library must implement a protocol to query candidate intermediate nodes about their current virtualization.

In addition, we can elect to prune edge (u, v) should an intermediate node w reside in an AS that is directly connected to the path from u to v, as illustrated in Figure 6(b). When the links into and out of the AS that contains w are poor, our algorithm may prune a better direct edge from u to v. In order to avoid this situation, we optionally probe the network to find out more precise information. For example, we examine latency from u to w and from u to v, and do not prune the edge if the difference is greater than some threshold.

Although each node u runs this algorithm locally, it does not build a disconnected mesh since it prunes the edge (u, v) only when it finds a physically similar alternate path to reach v. In order to select an intermediate node w for a virtual edge (u, v), local node u first gets the AS path from u to v, and then consults PG to collect all the ASes along the path or one AS hop away from the path. It only needs to check nodes w that live in one of these ASes, hence does not need to know the entire topology of the network.

4.3.2 Evaluation

We evaluate this strategy by measuring the percentage of the edges it prunes from the fully connected graph among the N overlay nodes. Our evaluation involves the following steps.

First, we combine BGP tables from RouteViews [6] and the BGP feeds at six PlanetLab locations to construct a peering graph with 15,396 ASes and 69,496 peering edges. We assume that BGP paths are the shortest (symmetric) paths in this peering graph for the sake of simplicity, while we acknowledge that this does not precisely reflect reality due to BGP's administrative policies [25]. Second, we take a sample of 1,000 potential overlays nodes, each of which resides in a distinct AS. Although the degree distributions of these sample nodes is roughly the same as the entire set, it is unclear if this condition alone ensures that these 1,000 nodes are topologically representative of the entire Internet. Third, we generate a sequence of these sample nodes and then in each round of the algorithm, we add one additional node from the sequence to the overlay network, thus incrementing the number of overlay nodes N. This allows us to evaluate mesh sparseness as a function of overlay size.

We evaluate two mesh-building algorithms. Algorithm 1 only virtualizes edges (u, v) for which there exists an intermediate overlay node w as shown in Figure 6(a). Algorithm 2 virtualizes these edges as well, and tries to further reduce the mesh sparsity by virtualizing edges (u, v) that resemble Figure 6(b).

Figure 7 shows the reduction in the total number of edges from the fully connected graph for the two algorithms, using a random sequence of sample nodes. As the plot shows, Algorithm 1 reduces the number of edges by 70% for an overlay with a large number of nodes (i.e., our mesh has only 30% of the edges in the fully connected graph). The virtualized paths from Algorithm 1 contain only 2 node-hops on average, and the number of AS-hops is always the same as the original path. As we expected, Algorithm 2 is even more aggressive and achieves over 90% reduction. This additional savings comes at a cost, as Algorithm 2 leads to longer virtualized paths, both in terms of nodes and ASes. On average, the virtualized paths produced by Algorithm 2 (with $N = 1,000$) contain 20 node-hops, and ten times as many AS-hops as the original BGP path, and the maximum number of node-hops and AS-hops along a virtualized path is anomalously high. We should note that the plots vary depending on the sequence of nodes we use, although we always use the same set of 1,000 sample nodes. For instance, when we sort the nodes by ascending degree, the curves rise up sharply and saturate with a small number of nodes. When we sort the nodes by descending degree, the curves do not climb until a large number of nodes are added.

We believe this preliminary result shows that a physically representative mesh can assist overlays like RON to scale by reducing topologically redundant probing. We are currently investigating other mesh-building strategies, as well as additional metrics for evaluating them.

5. DISCUSSION

Based on our experience to-date, we make three observations about how Internet routing (and BGP in particular) might be changed to better support overlays. First, as described in Section 3, BGP speakers need to export their routing tables to overlay networks. Without this coarse-grain connectivity information, bootstrapping the routing underlay is problematic. Second, while ASes that correspond to end-sites are easy to model, transit ASes are much too diverse to be accurately modeled as a single vertex/hop, forcing us to use latency probes rather than depend on AS hop counts. The underlay would benefit from more explicit information about how peers cluster at POPs. Ideally, coarse-grain topology information about the internal structure of long-haul ISPs would also be exposed. Third, our approach argues against pushing any dynamic capability into BGP [7, 8, 3, 4, 5]. Our position is that BGP should continue to provide only connectivity information, with dynamic functionality moved to higher layers of the routing underlay, thereby allowing us to define value-added routing services in a cleaner way, and avoid introducing route instability problems. On the other hand, we actually prototyped our topology probes in Zebra [2], an open-source implementation of BGP, meaning that primitives like GetPath can rightfully be viewed as extensions to BGP.

Our strategy for building a routing underlay is based on the simple observation that the accuracy of a routing mechanism comes at some cost, and hence, we would be well-served by doing a more careful cost/benefit analysis. On the cost side of the equation, one could evaluate an overlay routing mechanism in terms of the number of probes it performs, perhaps reported as the product of the *scope* of its probes (e.g., all N nodes in the overlay or just k neighbors) and the *frequency* of those probes (e.g., at configuration time, on a per-connection basis, or continuously). The benefit side of the equation is much more difficult to quantify since, ultimately,

we would like to compare the route selected for each packet to the route that a global oracle would have selected in order to optimize some metric. When evaluating routing mechanisms, however, we typically assume that one mechanism represents the desired behavior, and are simply trying to find a way to lower the cost without losing too much fidelity. Note that while we focus on the probing costs of routing, there are other potential costs, such as the over-utilization of popular links due to overlays being selfish [23].

Several primitives to support routing in overlay networks have recently been proposed. For example, Jannotti [16] defines two router primitives—*path reflection* and *path painting*—that are used to replicate multicast packets and to create an overlay topology that resembles the underlying network. Jannotti's approach focuses on how routers incrementally (and locally) improve how they map virtual links onto the underlying network topology. In contrast, our approach is to provide a more global picture of the underlying connectivity. Another example, the Internet Indirection Infrastructure (i3) [31], proposes *indirection* as a more flexible communication abstraction than traditional IP forwarding. One could view i3 as a generalized form of source routing overlaid on top of the Internet. Just as source routing often needs to identify way-points that result in the most appropriate path to the destination, i3 also benefits from a topology discovery service. In other words, i3 is designed mainly for the forwarding aspect of overlay routing, while our underlay architecture is designed to enable cost-effective topology discovery. We believe that i3 and our underlay could complement each other as an infrastructure for building routing overlays.

6. CONCLUSIONS

The main thesis of this paper is that allowing overlay networks to independently probe the Internet—with the goal of making informed application-specific routing decisions—is not a tenable strategy in the long run. Instead, we propose a shared routing underlay that overlay networks query. Although we acknowledge that the exact form this underlay takes is not yet well-understood, we posit that it must adhere to two high-level principles. First, it must take cost (in terms of number of network probes) into account. Second, the underlay will most likely be multiple-layered, with lower layers exposing coarse-grain static information at large-scale, and upper layers performing more frequent probes over an increasingly narrow set of nodes.

The paper proposes a set of primitive operations, along with an example library of routing services that can be built on top of the primitives. A preliminary evaluation suggests that a library of low-cost services is feasible, and we are currently deploying the services on PlanetLab. Given ISP and hosting site pressure on PlanetLab to limit the number of traceroutes and pings each node performs, it is likely that we will need to restrict overlay services to using such a shared facility in the near future.

Acknowledgments

We would like to thank the anonymous reviewers and Jon Crowcroft, our shepherd, for helping us improve the clarity and focus of the paper. This work was supported in part by NSF grant ANI-9906704, DARPA contract F30602-00-2-0561, and Intel Corporation.

7. REFERENCES

[1] Fixed Orbit. http://www.fixedorbit.com/.
[2] GNU Zebra. http://www.zebra.org/.
[3] netVmg. http://www.netVmg.com.
[4] Opnix. http://www.opnix.com.
[5] Proficient Networks. http://www.proficientnetworks.com.
[6] Route Views Project. http://antc.uoregon.edu/route-views/.
[7] RouteScience. http://www.routescience.com.
[8] Sockeye Networks. http://www.sockeye.com.
[9] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 131–145, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.
[10] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *Proceedings of Pervasive 2002 - International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.
[11] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang. Enabling Conferencing Applications on the Internet using an Overlay Multicast Architecture. In *Proceedings of the ACM SIGCOMM Conference*, pages 1–12, August 2001.
[12] Y.-H. Chu, S. G. Rao, and H. Zhang. A Case For End System Multicast. In *Proceedings of the ACM SIGCOMM Conference*, pages 1–12, June 2000.
[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.
[14] P. Druschel, M. Castro, A.-M. Kermarrec, and A. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 20, 2002.
[15] L. Gao. On Inferring Autonomous System Relationships in the Internet. In *Proceedings of IEEE Global Internet Symposium*, November 2000.
[16] J. Jannotti. Network Layer Support for Overlay Networks. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.
[17] T. Kernen. Traceroute.org. http://www.traceroute.org.
[18] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000)*, Nov. 2000.
[19] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring Link Weights Using End-to-end Measurements. In *Proceedings of the Internet Measurement Workshop*, pages 231–236, Marseille, France, November 2002.
[20] P. R. McManus. A Passive System for Server Selection within Mirrored Resource Environments Using AS Path Length Heuristics, June 1999. AppliedTheory Communications, Inc.
[21] National Research Council. *Looking Over the Fence at Networks*. National Academy Press, Washington D.C., 2001.
[22] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of the HotNets-I*, 2002.
[23] L. Qiu, R. Y. Yang, Y. Zhang, and S. Shenker. On Selfish Routing in Internet-Like Environments. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.
[24] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-Aware Overlay Construction and Server Selection. In *Proceedings of the IEEE INFOCOM Conference*, New York, NY, June 2002.
[25] Y. Rekhter and T. Li. A Border Gateway Protocol 4, March 1995. RFC 1771.
[26] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, November 2001.
[27] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-Scale Persistent Peer-to-Peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 188–201, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.
[28] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-end Effects of Internet Path Selection. In *Proceedings of the ACM SIGCOMM Conference*, Cambridge, MA, September 1999.
[29] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proceedings of the ACM SIGCOMM Conference*, pages 133–145, August 2002.
[30] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A facility for distributed internet measurement. In *Proceedings of the 4th USITS Symposium*, Seattle, WA, March 2003.
[31] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *Proceedings of the ACM SIGCOMM Conference*, pages 73–85, August 2002.
[32] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of the ACM SIGCOMM Conference*, San Diego, CA, September 2001.
[33] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: Offering Internet QoS Using Overlays. In *Proceedings of HotNets-I*, October 2002.
[34] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Proceedings of the 5th Symposium on Operating System Design and Implementatio (OSDI)*, Boston, MA, December 2002.