On the Scalability of Interdomain Path Computations

Onur Ascigil and Kenneth L. Calvert and James N. Griffioen
Department of Computer Science
University of Kentucky
Lexington, KY 40506

Abstract—**Recent research has considered various architectural approaches in which route determination occurs separately from forwarding. Such offers many advantages, but also brings a number of challenges, not least of which is scalability. In this paper we consider the problem of computing domain-level end-to-end routes in the Internet. We describe a system architecture and a prototype route computation service that provides performance information along with paths. The results of our experiments, which involve updating billions of routes and serving thousands of requests per second, suggest that the resource requirements for a single-domain end-to-end path service (i.e., a service that provides paths from one access domain to all others) are fairly modest.**

I. Introduction

Interdomain routing involves two main challenges: *scaling* and *policy enforcement*. Today's Internet comprises tens of thousands of autonomous systems (AS's), interconnected by hundreds of thousands of channels, resulting in a vast number of possible end-to-end paths. At the same time, the economic viability of the interdomain ecosystem depends on autonomous systems' ability to ensure that they only forward traffic for which they are compensated in some way (i.e., their ability to enforce their policies).

The Border Gateway Protocol (BGP) is the sole mechanism controlling interdomain routing in the Internet today. BGP addresses both challenges in the same way: by controlling information flow. Specifically, each AS filters the set of paths it receives before it propagates them. Thus, the processes of path *discovery* and path *selection* are mingled in BGP. This approach has some well-known disadvantages, including:

- AS policies with non-local dependencies can lead to oscillation and prevent convergence of routes to particular destinations [8].
- Transient topology changes require that unused (non-preferred) paths be re-discovered and knowledge of their existence be re-propagated throughout the routing system. This leads to slow convergence times, even without pathological policies [3].
- The protocol's filtering mechanism (needed for scalability) allows only a *single* path to a destination to be propagated by an AS. This precludes multipath transmission, which offers many advantages. (There are hacks to allow multipath when an endpoint has more than one IP address, but they do not work in the general case.)

In this paper we consider a radically different approach to interdomain routing, in which interdomain paths are computed in a semi-centralized manner and are selected prior to forwarding. More precisely, we consider the design of an interdomain *path service*, which discovers and computes domain-level paths between source and destination domains. Before sending, a source requests a set of paths to the given destination from its local path server. The resulting paths (or some subset thereof) are then used for forwarding. Forwarding could be achieved in various ways, for example by placing a (loose) source route in the packets, or by installing state in the AS border switches (a la SDN). Such a system seems to offer a number of architectural advantages, including (i) native support for multipath forwarding; (ii) the ability to apply different path-selection policies for different applications or user classes; and (iii) guaranteed convergence. Other advantages are described and discussed later in the paper.

Our main focus in this paper is scalability. We show that an interdomain path service can be practically implemented using relatively modest resources, even when the domain-level graph contains on the order of trillions of paths. We emphasize here that the proposed system has been developed as part of a "clean-slate" re-imagining of the Internet architecture [9] and ecosystem [14]; as such, backward compatibility with the existing Internet is not necessarily a goal. While we believe he approach described here *could* eventually be deployed in the Internet, that would necessarily involve many challenges that are beyond the scope of this paper.

Our path service *is* intended for use with a network layer that, like IP, provides a best-effort datagram delivery service. Obviously such an arrangement is only practical when there is substantial opportunity for re-use of paths, so that the cost of obtaining paths from a path service can be amortized over many transmitted datagrams. We observe that this requirement is satisfied in the Internet today, with most packets belonging to *flows* that contain many packets traveling between the same source and destination, and others generally travelling to destinations that change slowly (e.g., DNS servers). Also, as was noted above, refactoring the routing architecture implies a requirement to enforce provider policies and ensure that compensation flows from those who use infrastructure to those who provide it. We briefly describe some alternative mechanisms for this in the next section.

The rest of this paper is organized as follows. In the next section we describe our model of the network ecosystem, which differs in some respects from the current Internet. We give a more precise problem statement, describe the operation of the path service, and describe two approaches to enforcing provider policies—that is, of ensuring that providers are compensated for the traffic they forward. In Section III we describe the design of the path service. Section IV describes the experiments we ran to measure the performance of our

This work was supported by the National Science Foundation under grants CNS-0626918 and CNS-1111040.

ISBN 978-3-901882-58-6 © 2014 IFIP

implementation, and presents the results of those experiments. Section V discussed related work. We present conclusions in Section VI.

II. System Overview

After describing the entities that make up our network ecosystem, we describe its typical operation and state the problem more precisely. We then discuss the economic relationships among the participants and compare them to those in the current Internet. (Economic incentives are a key factor in the viability of any future Internet technology [14].) We then discuss some possible solutions to the problem of enforcing policies; such a mechanism is crucial to the economic model, but depends on the way forwarding is implemented, which is beyond the scope of this paper. Finally, we point out some advantages of our system compared with today's Internet.

A. Players

We consider a network made up of separately-administered *domains*, which correspond roughly to the autonomous systems of today's Internet. Abusing terminology slightly, we will use *provider* and *domain* interchangeably.[1] We avoid the term "AS" because of its connection with BGP and today's routing and forwarding system. We classify domains as either *transit* (also called *relay*), or *access*. *Access* domains exist to provide service to *users*. *Transit* domains exist *only* to interconnect access domains. (Contrast this with today's Internet, in which AS's can play both access and transit roles.) Thus, every packet originates from and is destined for, an access domain. (In reality, packets will travel between finer-grained entities; the architecture can be applied recursively [9]. Also, some means is required to transform endpoint identifiers into destination channel IDs. However, we abstract from that problem for the purposes of this paper.)

Domains are connected via named *channels*. A channel is created between a pair of domains if and only if both deem it mutually beneficial. In general, multiple channels will connect a pair of domains (Fig. 1). In general, one domain connects to another for the same reason AS's connect in the Internet today: to get access to a greater portion of the Internet. However, in our model there are no customer-provider relationships between transit domains; all relationships are peering. Moreover, all transit policies are *strictly local*: a transit provider makes money by relaying packets between its incident channels, period. Thus, transit providers have no say over what happens to traffic once it leaves their domain.

The collection of transit domains forms a switching network that provides at least one path between every pair of access domains. Transit domains form the "switching elements" of this network; they simply relay packets between ingress and egress channels. Our scheme is thus agnostic regarding the internal structure of a transit domain, and the mechanism(s) used to convey packets between ingress and egress points of domains, is opaque to our scheme; a domain might use MPLS [13], SDN [10], IP tunneling, or regular intradomain (IP) routing.

Access domains can peer with any number of transit domains for only the cost of the connecting channel(s)—that is, multihoming comes "for free". Indeed, in our experiments we assume that each access domain connects to at least four transit domains for path diversity and competition. Each packet is forwarded from a source access domain, through zero or more transit domains, to a destination access domain.

One other type of entity is present in the ecosystem. *Path providers* collect information from transit domains about connectivity between their ingress and egress channels, and use that information to compute paths between access domains. To send a packet, a user requests some number of paths to the destination domain from its local path provider. Fig. 1 shows all these entities and the information flow among them.

It is quite possible that *brokers* (which act as intermediaries among users, access domains, and path providers) or other kinds of "middlemen" would arise in this ecosystem. We do not consider them here, although they are quite compatible with our model.

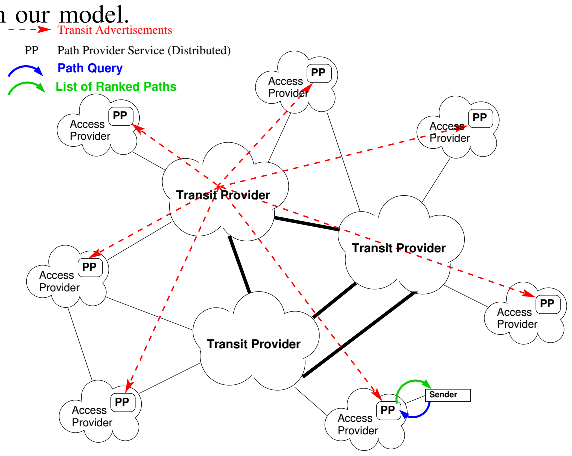


Fig. 1: Entities and Information Flow

B. Model of Operation

Each transit provider periodically emits a set of *relay advertisements*, each of which is an offer to relay packets from one of its channels (the *ingress channel*) to another (the *egress channel*). Each relay advertisement includes the offering provider, the ingress and egress channels, plus performance information and other parameters (e.g., the capacity of each channel, the recent and long-term average utilization of each channel, histogram(s) indicating the distribution of ingress-to-egress delays over some specified period).

Each path provider collects relay advertisements and uses them to construct a database of possible paths from a source domain to other access domains. Each path is an alternating sequence of channels and transit providers. Along with information about the channels making up the path, the path provider also keeps information about the performance of the path, which is derived from the performance information from the individual relay advertisements. The path provider keeps each path's performance updated as it receives new information about relays. It also updates the path database when new relays are advertised or existing ones are withdrawn.

To initiate an inter-domain flow, a source transmits a *path query* to a path provider, which may be local or in another domain. (Each source is configured with paths to one or more path providers at enrollment time, via a DHCP-analogue protocol.) The path query contains: the source and destination domains, the number of paths requested, and (optional) policy "hints", i.e., characteristics of the paths desired. Such hints

[1]The reader is cautioned not to interpret our term "provider" as meaning "current ISP" in this context.

could include, for example, a “white list” of providers to be used if possible, a “black list” of providers to avoid, advice about the application characteristics (e.g., streaming, interactive, or bulk transfer), or desired performance levels. These hints are used to filter the set of paths returned in response to the query. The number of paths returned is the number that satisfied the policy hints, or the number requested (bounded by some system maximum), whichever is smaller.

Although the path service is “centralized” (in the sense that each source sends queries to a specific place), the *global* path service can be distributed, since each access domain only needs paths from itself to other access domains. In our experimental evaluation, we assume that a path service computes paths from one access domain to all others. Note that this does not preclude larger services, which provide paths for multiple source domains, or multiple (possibly competing) services per access domain.

C. Problem Statement

Given the above model, our goal is to build a service that can provide, on demand, multiple paths from a single access domain to any given destination access domain. It must perform this function in a graph at least the scale of today’s Internet AS graph (10^5–10^6 nodes), with latency comparable to an Internet round-trip time (a few tens of milliseconds). The service should be implementable on commodity hardware, and should scale through parallelism.

In addition, we want the service to provide the most recent available information about the *performance characteristics* of any returned paths, to help the application or access domain choose among them. This might include, for example, utilization of the bottleneck channel(s) in recent time intervals, or the distribution of delays recently experienced by packets along that path; such information can be derived from the information for the individual relays making up the path.

D. Economic Considerations

The flow of compensation in our system is as follows: Access domains make money by charging users to connect to the network. Transit domains make money by charging for transit service. Path providers make money by charging users—either directly, or indirectly by charging their access providers—for providing paths, and access to the transit services that implement those paths. Thus, users pay access providers and path providers; path providers pay transit providers for the use of their relay services. Transit providers *must* receive compensation (directly or indirectly) from any users or access domains whose traffic they forward. One thing that makes this feasible is that the path providers acts as a clearinghouse for payments from users to transit providers.

Such payments might occur on various time scales. In the limit, a path provider might charge for each individual request fulfilled. More plausible, however, would be a subscription-type service, in which users (or their access domains) contract with a path provider for path service over a longer time interval; the path provider in turn pays transit providers for service.

One of the features of this system is that *money follows traffic flow*, while traffic flow is only weakly constrained.[2] This is in contrast to the current Internet ecosystem, in which traffic is (mostly) constrained to flow between customer AS’s and their providers. In other words, in the current Internet ecosystem, *traffic flow follows money flow*. Because the money flow changes very slowly, competition based on service quality is effectively inhibited.

[2]Certain path providers might contract with a limited subset of transit providers to get relay information and collect for transit usage for reasons of scalability or business relationships. This is fine, provided the subset ensures adequate path diversity to every destination domain.

E. Policy Enforcement

It remains to show that transit providers can enforce their policies—that is, control access to their resources. If all path services receive all relay advertisements, and a path service constructs many possible paths (up to some maximum path length), in principle any access domain can send packets via any of a large set of transit providers. However, a transit domain should only carry traffic that it knows it has been (or will be) compensated for. The problem is that once a source knows a path it can continue to use it forever (including after it stops paying the path provider), unless some access control mechanism is provided.

The form of such a mechanism will depend on the way forwarding is implemented. Here we discuss two possibilities, which correspond to an SDN-based forwarding approach and a source-routed approach to forwarding.

a) Stateful, SDN-based enforcement.: In this method, when a path provider returns a path to a user, it informs an SDN controller for each transit domain in the path. Each controller then installs state in its domain, which causes packets arriving on the specified ingress channel that match a specified pattern—say, source and destination IP address (or prefix) and/or flow ID—to be forwarded through the network to the specified egress channel. The transaction between the user and the path provider gives the user access to the returned paths for some period of time, after which the switch state is removed from the transit domains, revoking that user’s access. Note that—in contrast to prior work on using GMPLS to establish inter-domain paths [5]—no co-ordination is required between either the switches or the controllers for the different domains. The advantage of this approach is that it can probably be made backward-compatible, using existing protocol headers, thanks to SDN. It has the usual disadvantages of stateful forwarding in networks, and also increases the pretransmission latency.

b) Stateless, in-band enforcement.: In this approach, packets carry an explicit representation of the path in the packet, along with a proof of policy compliance for each domain in the path. Such a proof could take the form of a cryptographically-derived *token*, based on a secret shared between the path provider and the transit domain, and containing an expiration time. Such a scheme for verifying policy-compliance of source-routed packets is described in Platypus, for example [12]. The ingress switch in each transit domain verifies compliance, then simply forwards the packet toward the indicated egress channel. (As with the stateful approach, the intra-domain portion of the path is completely up to the provider.) This approach has the advantage of greatly simplifying the state requirements of transit domains. It has the disadvantage of adding to the computational load on the data plane, and significantly increasing the overhead in the packet. Both of these costs can be amortized over multiple uses, for example by verifying packets at random intervals.

F. Features of the Approach

In this section we point out some advantages and challenges of our proposed approach, compared to the current

Internet.

- **Application-specific path selection.** The ability to select paths with different characteristics for different classes of applications—without building application knowledge into the infrastructure—is a key benefit of our approach.
- **Performance Differentiation.** Our system allows performance information to be computed for paths and conveyed to the application, thus enabling applications to manage their own quality of service.
- **Transparency.** In our system the scope of each provider's (access or transit) responsibility is clear and well-defined. Contrast this with the current Internet, in which customers pay ISPs for Internet access, but no ISP can provide this service by itself, because it controls only a small fraction of end-to-end paths. In particular, a customer cannot hold a provider responsible for downstream service failures.
- **Competition.** Because providers are paid for well-defined services, they have an incentive to innovate and compete based on service quality.
- **Efficiency.** In today's Internet, the routing system amortizes the cost of computing routes to *all* possible destinations over *all* packets; every node has a route to every destination at all times. Our system admits a much wider range of amortization schedules. For example, paths to certain destinations might be computed on demand, if latency is not important for the users. Paths that are frequently used can have their performance information updated more often.

These advantages are not without cost. In particular, our scheme adds latency (to obtain paths) before a first packet can be sent. If paths are unidirectional, a similar delay will be required before any reply packet can be sent. One goal of this paper is to give evidence that a well-engineered system can keep these delays within acceptable range. The policy-enforcement mechanism (see Section II-E) adds additional overhead, in the form of either signaling delay or significant packet header expansion.

Finally, before presenting our system design, we consider the *time scales* of events of interest to the global routing system.

- The *set of possible paths* is defined by the set of access and transit domains and the channels that interconnect them. This set changes only when new providers and channels come into existence or existing ones go out of service permanently. It is therefore reasonable to perform an expensive computation to construct a long-lived database of possible paths up front, and thereafter update it on a relatively slow timescale (hours to days) as the underlying graph changes.
- The *medium-to-long-term performance* of a relay, or of a path, changes on a timescale of seconds to minutes. It makes sense for transit providers to provide periodic updates to relay performance information—say, a histogram of measurements observed over the past minute, and a EWMA-filtered history of such measurements. A path service can then update a *path's* performance information the next time a path using that relay is returned to a user.
- The users of a path are in the best position to observe the *instantaneous status* of a path—whether it is up or down, and its current performance. Given adequate path diversity, the knowledge and ability to select from multiple paths enables a source to quickly detect and react to temporary outages and recoveries—generally on the scale of a round-trip-time, i.e., tens to hundreds of milliseconds. In contrast, today's global routing system converges on the timescale of minutes.

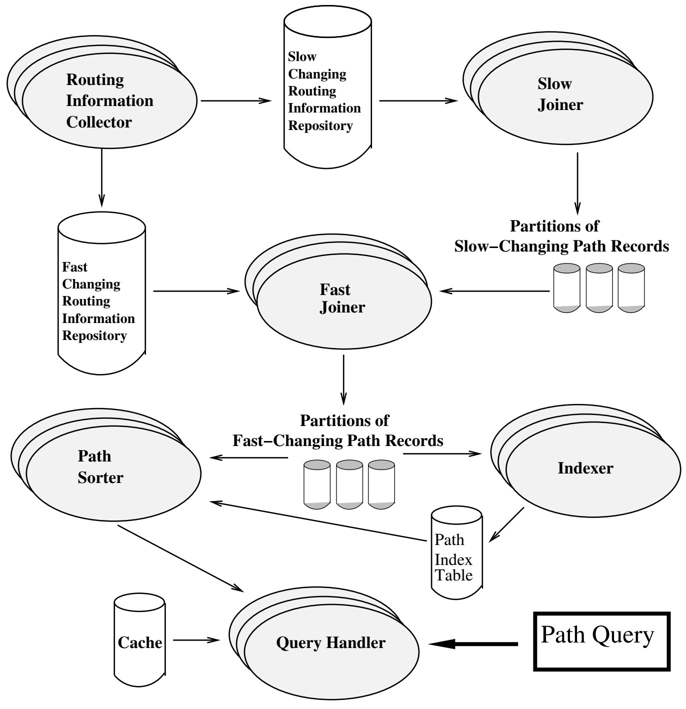

Routing Information Collector
Slow Changing Routing Information Repository
Slow Joiner
Fast Changing Routing Information Repository
Fast Joiner
Partitions of Slow−Changing Path Records
Partitions of Fast−Changing Path Records
Path Sorter
Indexer
Path Index Table
Cache
Query Handler
Path Query

Fig. 2: Path Service Architecture

III. Path Service Design

In this section we describe the conceptual design of the path service described in the foregoing sections. Our goal is to show that the resource requirements to provide adequate service are rather modest. Obviously this is only one point in a large design space; we evaluate its effectiveness in the next section.

Fig. 2 is a high-level overview of the design, which comprises the *routing information collector (RIC), slow-joiner, fast-joiner, sorter, query-handler, and indexer*. Arrows between components in Fig. 2 indicate the flow of information. Ovals represent threads or processes; the number of times each is instantiated can be varied as resources permit.

RIC is a distributed component that receives periodic relay advertisements from transit domains. Each RIC component collects information from a different set of providers in parallel. The collected routing information consists of "slow-changing" and "fast-changing" components. The slow-changing part comprises connectivity information and load-independent properties of relays such as capacity and propagation delay. Fast-changing information includes load-dependent properties like delay distribution and utilization. The received information goes into databases of slow-changing and fast-changing path attributes, respectively. In the implementation evaluated in Section IV, the fast-changing information consists of packet delay and available bandwidth distributions, both

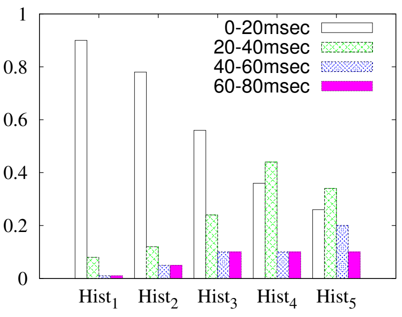


Fig. 3: Latency histograms of five paths sorted from left to right according to their likelihood of satisfying $delay \leq 20$

represented as simple histograms. All relay advertisements are assumed to contain histograms of the same size.

The slow-changing routing information is used by the *slow-joiner* to generate partitions of *slow-changing path records*. A path record contains the sequence of relays that make up the path, along with the aggregated path performance attributes derived from the attributes of the relays in the path.

The *fast-joiner* computes the performance information (delay and available bandwidth distributions) of a path by performing join operations on the relay distributions. In the case of delay distributions, the fast-joiner performs (approximate) *convolution* on the relay distributions. Convolution on two histograms with n bins has a complexity of $O(n^2)$. We use a quantizing approximation described by Nahrstadt et al [15], which treats histograms as if each measurement's value is equal to the midpoint of the bin. Thus, if the bin boundaries are 0 and 20, the operation assumes each observed value (packet delay) counted in the bin was actually 10. This introduces a (bounded) error, but maintains uniformity of the data structure. In the case of available bandwidth distributions, the join operation involves a simple *min* operation on the distributions with a complexity of $O(n)$.

The *sorter* produces a ranking of paths (i.e., path records) according to their likelihood of satisfying the end-system's constraints. For each query that specifies constraints, the sorter first computes each path's probability of satisfying the constraints, ten performs the sorting operation.

Consider a path query that requests k paths from s to d, with delay bound c. The sorter first uses the histogram associated with each s-d path to compute the probability that delay is less than c for that path. That probability is computed as the fraction of the packets in the histogram that are not to the right of the bin containing c; in other words, c is rounded up to the nearest bin boundary, and the fraction of packets in that bin or to the left of it is computed. Fig. 3 shows an example with delay histograms for five paths, where the bin boundaries are at 0, 20, 40, 60, and 80 msecs. The histograms are sorted from left to right in order of decreasing likelihood of satisfying the delay constraint $delay \leq 20$.

Each sorter deals with a separate part of the fast-changing path info database and sorts its own assigned partition. Consider again the above query that requests k paths from domain s to domain d, with delay not greater than c. The *query-handler* receives the query and forwards it to the sorter, which carries out the following steps:

1) Each copy of the sorter retrieves all path records with source domain s and destination domain d from its partition. (A pre-computed path index table is used for fast access to paths by source-destination.)
2) Each sorter computes the likelihood each of its paths satisfies the delay constraint, as described above. This computation requires a single pass through the entire list, which produces the k paths in the partition with the highest likelihood of satisfying the constraint are obtained.
3) Each node in the cluster reports its top k paths to a master node.
4) The master node merges and filters the lists to obtain the k overall best paths.

When each partition of path records contains roughly the same number of paths for a given source-destination pair, each sorter node has roughly the same computational load.

Path queries that include additional hints will, of course, require additional processing steps, for example to filter out paths that use providers on a blacklist or not on a whitelist. The path index table contains pre-computed mappings to locate path records by the providers that they traverse. The *indexer* periodically updates the table only when the list of path records change as a result of changes in slow-changing routing information (infrequently).

IV. Evaluation

In this section we describe an implementation of our path service and present results of our evaluation of its performance. We first describe the input topology that we use in our experiments and also explain how we generate paths; Section IV-B presents details of the implementation and results.

A. Topology

The input to our experiments is an Internet AS-level topology that is constructed using CAIDA's AS relationship dataset [1]. The CAIDA AS relationship dataset consists of AS adjacencies along with annotated relationships (e.g., customer-provider).

The current Internet topology, having domains that act both as access and transit at the same time (e.g. tier-1 domain hosting content), does not map directly to our model, where each domain is transit or access but not both. We therefore transform the CAIDA AS topology to a slightly different domain-level topology that may emerge as a result of separating access and transit roles of domains. The first step is to identify AS's in the CAIDA topology that have mixed roles; the second step is to split them into an access and a transit domain, connected to each other by channels. In order to identify domains with mixed roles, we first classify each domain in the CAIDA topology as either i) primarily transit or ii) primarily access. A method suggested by Dhamdhere et al [4] infers the primary business of each domain (based on the size of its customers, providers and peers) and classifies each domain into one of three categories: (i) transit providers (TPs) (ii) Content/Access/Hosting providers (CAHP) or (iii) Enterprise Customers (ECs), corresponding to various companies, universities and organizations. Applying the above classification method resulted in 35,287 primarily EC, 4,646 primarily TP and 1,755 primarily CAHP domains. Given these results, we further classify EC and CAHP domains as primarily access and the TP domains as primarily transit domains.

We used a day's worth of unsampled NetFlow traces collected at the egress point of a large University campus network to identify domains classified as TP that *also* had secondary hosting or content provider roles. More precisely, 2700 of the 4,646 original transit domains were destinations for a significant number of flows (i.e. 10% of all the flows) in the trace set, and so were identified as "mixed role" TPs. In addition to TPs with mixed roles, a subset of the domains classified as primarily access (ECs and CAHPs) have customers, according to CAIDA data, which means they provide transit service to other domains. In order to reflect the secondary roles of these domains in the topology, we split them into two, one transit and one access domain, connected by multile channels; all channels of the original domain were connected to the transit component. The resulting topology has 37,987 access domains and 6,401 transit domains.

In the Internet today, there are typically multiple channels between large transit providers. To reflect that in our experimental topology, we added 20,000 additional edges between transit providers, placing them randomly between pairs of adjacent transit domains, with higher-degree domains favored. Also, we added a total of 18,694 new edges between access and transit providers in order for access domains to have an average multi-homing degree of 4. When adding these multi-homing edges, we selected access providers uniformly at random. The resulting graph used in our evaluation has 302,148 (symmetric uni-directional) channels.

Using this topology, we compute the following paths from a random source domain s: for each destination (access) domain d, we compute all *shortest* paths from each egress channel s_e of s to each ingress channel d_i of d. We also compute all paths from s_e to d_i of length $L+1$, where L is the length of the shortest path from s_e to d_i. This resulted in approximately 1.65 billion unique paths from a source domain (a university campus network) with four egress channels to all destination access providers.

In generating these paths, we made certain assumptions about the transit routing policies of transit domains. Specifically, we distinguish between those transit providers (TP_{sec}) that resulted from splitting CAHPs or ECs (i.e., for which transit is a *secondary* function), and those (TP_{pri}) that are *primarily* or exclusively a transit provider. For the latter we assumed an unconstrained policy: they offer relays between any two of their channels. This policy makes it more challenging to scale the service. On the other hand, the TP_{sec} transit providers are assumed to have a more constrained policy, viz., they only forward packets between neighbors that CAIDA identifies as their peers and customers and neighbors identified as providers and customers. With these two types of transit policies, the total number of relays offered is around 180 million. However, the computed paths from the random source s to all destinations uses only about 12 million relays.

The topology that we used in our experiments is designed to approximate the structure of the current Internet topology. Future Internet topologies may differ in terms of size and connectivity properties. Because our approach scales through parallelism, future Internet topologies with more potential paths than the current Internet can be accommodated by adding additional instances of components that take part in both the on-demand (search) computations (e.g. *path sorters*) and the pre-computation of paths (e.g. *slow-joiners*).

B. Implementation Details and Results

1) Setup: We implemented a distributed path provider and ran experiments on a single server with 4 Intel Xeon X5650 processors containing a total of 24 cores and 128GB of memory. The path provider implementation consists of S *searchers*, U *updaters* (i.e. *Routing Information Collectors (RICs)*), a *query-handler* and a *cache-controller*. The routing information updates and path queries are fed into the path service system using *update-generator* and *query-generator* processes respectively.

A searcher in the implementation corresponds to a combined *fast-joiner* and *path-sorter* in Fig. 2 and it is responsible for computing the top-k paths within a partition of all path records using the most recent routing information. Path records are partitioned onto searchers so that each searcher is assigned about the same number of paths to each destination; this leads to a more balanced workload among searchers. Updaters are responsible for updating the routing information repository upon receiving new routing information from the update-generator. The unit of routing information is a *relay record* that consists of the relay (i.e., ingress and egress channel identifier pair) and the relay's attributes (i.e. performance characteristics). In our experiments, we used latency and available bandwidth as the two attributes of relays. Both of the attributes are represented in the form of histograms, each containing 5 bins.

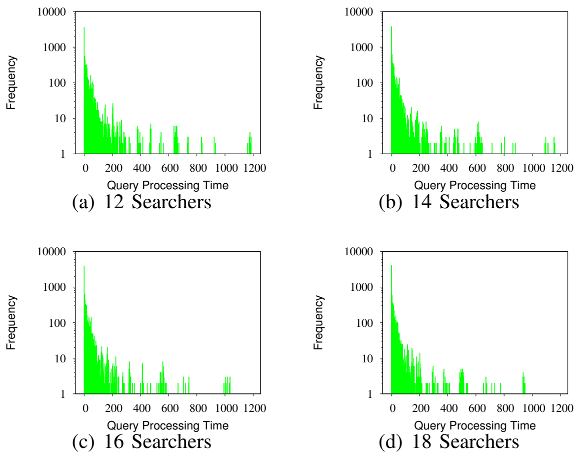
The query-handler is responsible for relaying path queries to the searchers, collecting and merging the individual top-k results obtained from the searchers to obtain the final top-k results. The query-handler is also responsible for sending search and insertion requests to the path cache, which is maintained by a separate process called *cache-controller*. Upon collecting the individual top-k results from the searchers and merging them to get the final top-k results, the query-handler sends a request to the cache-controller to store the final results to the query. In our experiments we use a query-generating process, which is separate from the path service. Query-generator sends path queries to the query-handler at a sufficiently high rate to keep the incoming request queue (with size 10) of the query-handler full at all times. The update-generator periodically broadcasts the changes in routing information to the updaters.

In all of our experiments, we used unsampled Netflow traffic traces collected at the egress point of an access network to generate realistic path queries. The IP flows in the Netflow traces are pre-processed to map each destination IP address to a destination domain. In the Netflow traffic traces, we observed a significant locality of reference in the destination domains contacted by the outgoing flows. This increases the importance of the path caching mechanism, which significantly improves the scalability of path computations. The path-cache is flushed when new updates arrive periodically; thus the effectiveness of caching also depends on the frequency of updates.

A path query consists of a destination domain, a traffic class and k. Each traffic class is statically mapped to a list of constraints on bandwidth and latency. In our experiments, we used five traffic classes: i) Low latency (e.g. interactive traffic) class ii) high bandwidth (e.g. bulk transfer traffic) class, iii) Medium bandwidth and low latency class (e.g. video conference traffic), iv) high bandwidth and low latency class (e.g. real-time, high quality video streaming traffic) and v) best-effort (e.g. email traffic) class. In our experiments, we generated queries with random traffic classes and also with traffic classes that is unique for each destination domain. Upon receiving a query, the path service computes top-k ($k \leq 10$) paths that are *most likely* to satisfy the requirements of the given traffic class.

We are interested in the resources required to serve a stream of path requests with acceptable latency, while concurrently processing state updates. The performance metric of interest is the average query processing time, which we measure as the time from when the query is received until the response is ready to send (In a real system there will also be latency to transmit the query and response, but if the service is located in the source access domain, we expect this to be small—comparable to DNS lookups today). The parameters varied include the number of searchers threads and the frequency of routing information updates.

Upon receiving a query from the query-handler, a searcher first locates the path records in its partition with source s and destination d using its pre-computed *index*. Then, the searcher goes through each path record and determines whether the attributes of the relays that form the path have changed since the path attributes were last computed. The searcher computes the attributes (i.e., latency histogram) of each stale path and the probability of each path's satisfying the constraints that is associated with the traffic class specified in the query. The searcher then sorts the path records according to their probabilities and returns the top-k paths. Because the searchers read from relay records that are periodically written by the updaters, locking mechanisms are implemented to protect the integrity of the relay records.



(a) 12 Searchers (b) 14 Searchers (c) 16 Searchers (d) 18 Searchers

Fig. 4: Distribution of query processing times for different number of searcher threads.

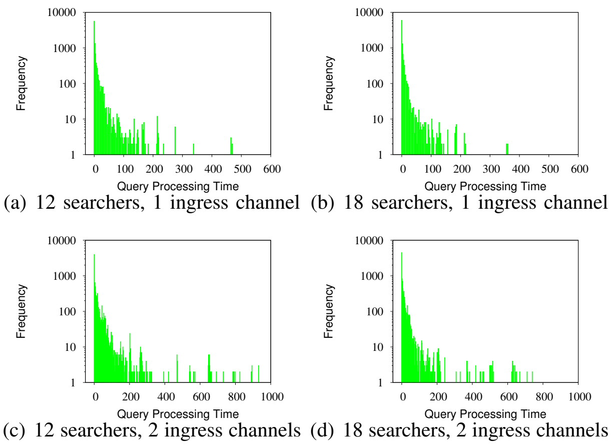
2) Experiments: To evaluate the scalability of our design, we measured the distribution of query processing times with varying number of searchers. The query processing time is the amount of time between the submission of a query to a query-handler until the query-handler collects workers' top-k results and merges them to produce the final top-k result. In Figures 4(a), 4(b), 4(c) and 4(d), the distribution of query processing times with various numbers of searcher threads are shown. As one would expect, the average query processing times decrease as the number of searcher increase. The average processing times corresponding to 12, 14, 16, and 18 searchers are 39.79, 35.12, 31.22 and 28.01 milliseconds per query, respectively. In these experiments the total number of queries that are processed is set to $10,000$, the traffic class values of the queries are selected randomly and updates arrive every 10 seconds each with $100,000$ relay records. We have also tested our approach with larger updates with sizes up to $500,000$ relay records and did not observe significant changes in terms of query processing time. On the other hand, the frequency of query arrivals have significant impact on the query processing times (as shown in Fig. 6(b)) because of their impact on the efficiency of caching as we explain in detail later.

Despite the random selection of traffic classes in the queries, the cache hit rates for the experiments in Figure 4, with 12, 14, 16 and 18 searcher threads are 32.64%, $33,08\%$, 34.02%, and 34.97%, respectively. The high cache hit rates in these experiments is the result of the high locality of reference in the set of destinations contacted by the flows in the Netflow traces. We also performed experiments where we assigned a static traffic class to each destination so that the queries to the same destination have the same traffic class. Space constraints preclude us from presenting the distributions of processing times for those experiments; as expected, however, using static traffic class values in queries leads to even higher cache hit rates of around 60%, which in turn leads to lower average query processing times for the path service. On the other hand, in the absence of caching, the average query processing times for 12, 14, 16, and 18 searchers grows to to 69.74, 64.94, 59.96, and 55.6 milliseconds respectively.

The large processing times in the above results in Figures 4 correspond to queries that request paths to destinations that have very large number of paths. To destinations with large degrees, there are a large number of paths because the number of paths for each source egress and destination ingress combination adds up. We observed a strong correlation between a destination's popularity and its multi-homing degree in the network traces, which is logical because popular destinations tend to receive large volume of traffic and as a result require large multihoming degrees to handle the traffic. Therefore, the path-cache effectively reduces the occurence of expensive calculations for the popular destinations.

In the real deployment scenario, the destination domains are not expected to allow paths through any ingress channel to their network. Instead, destination domains have strong incentive to perform ingress traffic engineering, which involves selecting a set of ingress channels and provide this selection as an additional hint to the path service. Similarly, the source domains are likely to select few of their egress channels to influence the outgoing traffic. In order to test our path service under the traffic engineering scenario, we first limited each destination and source to choose 1 ingress channels. The query processing times for 1 egress and 1 ingress channels are given in Fig. 5(a) and Fig. 5(b) for 12 and 18 searcher threads, respectively.

In a subsequent experiment, we tested with sources and destinations picking 2 incident channels given that they have at least 2 incident channels. The query processing time distributions for up to 2 incident channels are given in Fig. 5(c) and Fig. 5(d) for again 12 and 18 searcher threads, respectively. The average query processing times differ significantly between the tests with 1 and up to 2 incident channels. For the case with 1 incident channels, the average query processing times for 12 and 18 searchers are 8.2 and 6.45 milliseconds, respectively. On the other hand, the average query processing times for the case with 2 egress and up to 2 ingress channels are 26.12 and 19.32 for 12 and 18 searchers, respectively. The difference between the results of the two cases is due to the approximately 4 folds increase in the number of paths considered during the search operations. These results suggest that it may be a good practice to divide the path computation between two domains into individual source and destination channel pairs and perform these computations in parallel in different servers.



(a) 12 searchers, 1 ingress channel (b) 18 searchers, 1 ingress channel
(c) 12 searchers, 2 ingress channels (d) 18 searchers, 2 ingress channels

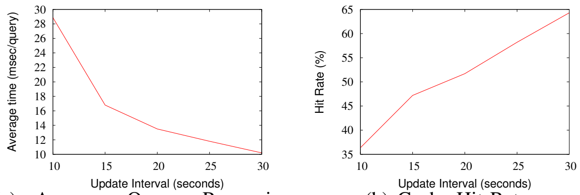
Fig. 5: Distribution of query processing times for different number of searcher threads with ingress traffic engineering.

In the above experiments, we hold the routing information update rate constant at $100,000$ relay records arriving every 10 seconds. In order to understand the impact of update arrival frequency on the query processing times, we measured the performance of the path service as the update arrival interval varies for a fixed number of 18 workers, with random traffic classes in each query. The average query processing times and the caching hit rates are shown in Fig. 6 for update intervals of 10, 15, 20, 25, and 30 seconds. As expected, the average query processing times increase as the update frequency decreases (i.e. increasing intervals), due to the increasing effectiveness of caching. The cache hit rate increases when updates arrive less frequently because the path query results can be reused for a longer time period; between comitting of the updates to the routing repository by the updaters and the arrival of the next update.

While the updaters apply changes to the routing information repository, caching is disabled to prevent stale data from being returned. Once the updaters are done with processing a routing update, an "update-complete" signal is sent to the cache-controller to resume caching of query results until the next update arrival. The rate at which updates are completed is important because it constrains the effectiveness of caching, as well as the quality (i.e. freshness) of the results. Therefore, a number of updaters simultaneously apply the necessary changes to the routing information repository containing the relay records. In all of our experiments, we used 2 updater threads that simultaneously update the relay records. Using 2 updaters, the processing of $100,000$ relay updates takes around 0.025 seconds on average. For larger update sizes, it may be necessary to increase the number of updaters.

It is also worth pointing out that the overhead of relay updates on the network is fairly small especially because the majority of the relay updates consists of only updates on fast-changing attributes of relays, and only a small portion of updates need to carry both the slow-changing and fast-changing attributes (i.e. only upon changes in slow-changing attributes, which happen infrequently). Assuming that the bin boundaries of histograms are static and are not communicated in each update, a single relay update, which carries only fast-changing attributes of a relay is only around 20 bytes and contains a globally unique relay identifier and the value of each individual bin (ranging between 0 and 100) in the two histograms corresponding to delay and available bandwidth. A globally unique relay identifier can be assigned to each relay either based on the relay's ingress and egress channel identifiers or some other method known to both the path service and the domains. The overhead of an update containing $100,000$ relays with only fast-changing attributes that is sent every 10 seconds is only around 1.52 Mbits/sec. Relay updates with slow-changing information contains identifiers of ingress, egress channels, and the various slow changing relay attributes such as bandwidth capacities, propagation delays and so on in addition to fast-changing attributes.

In our results, we emphasized responsiveness of the service rather than the quality (i.e. freshness of routing information used to compute paths) of results. Hardware constraints prevented us from testing other computational strategies that require additional computational components or storage. For instance, we only considered a fully on-demand approach to computing path attributes (e.g. convolutions), while a mix of on-demand and pre-computational approaches may lead to better results; for instance by pre-computing the attributes of a subset of *popular* paths (i.e. using additional processes that work in parallel) before the arrival of queries in addition to query-time path attribute computations. Also, one can use proactive caching mechanisms to update popular cache entries rather than simply flushing all the entries upon the arrival of new updates; however this requires additional computational components that are dedicated to this task. Another strategy to possibly speed-up the path attribute computations significantly is to cache the aggregate attributes of the *subsections* of paths that are common to many paths (e.g. sequence of relays from source domain to a tier-1 domain). Caching subsections of paths; however, require significant amount of additional storage space that needed to be used conservatively on a single server. We leave to future work the implementations of the above strategies possibly in a truly distributed setting such as a high-performance cluster (HPC), where computation and storage constraints are not as limiting.



(a) Average Query Processing Times (b) Cache Hit Rates

Fig. 6: Average query processing times and the cache hit rates for different update intervals

V. Related Work

A variety of emerging future internet architectures have embraced the idea of separating the routing decision from the forwarding decision, thereby enabling a wider range of routing policies than is possible in the current Internet. Typically this manifests itself as a form of source routing in which each packet carries the path to be travelled. Packets in the Platypus [12] overlay source routing system are stamped using capabilities for the paths they traverse. The stamps enable providers to ensure packets follow policy-compliant paths, via a data-plane check. This approach allows routing to be separated from forwarding, although Platypus does not explicitly define how routes should be computed. Another example is Nebula's ICING [11] protocol where packets identify the set of "realms" along a path and also include explicit proofs of consent to traverse those realms. Routes (i.e., consent for paths) are obtained using an up/down approach similar to that used in NIRA [16] where up-paths are combined with down-paths to create end-to-end paths. The SCION [17] routing protocol in the XIA [2] architecture takes a similar approach, using path construction beacons from a trust domain's core to discover up/down paths and their intersections that then form an end-to-end path. The Pathlet architecture [7] uses source routing over a virtual topology and allows providers to use a rich set of routing policies when declaring services. While many of these emerging architectures provide the ability to separate routing from forwarding, they are largely focused on the forwarding mechanisms that make the separation possible, rather than exploring new, efficient, and scalable ways to advertise topology and compute routes. Unlike our approach, these approaches are not designed to offer timely information about current delays, available bandwidth, or other metrics that change frequently on small timescales. Douville etal. [5] proposes an automated method to compose connection-oriented inter-domain transit services that is built on top of the PCE (path computation element) architecture [6], which separates forwarding and routing by introducing a centralized path computation element within a domain. The automated method discovers AS-level paths that satisfy various constraints using a simple breadth-first search like approach and it is not designed with the goal of responding to a stream of queries from users in a timely manner.

VI. Conclusion

We have proposed a system in which end-to-end interdomain paths are computed offline, and investigated the scalability of such "centralized" path computation in a graph resembling the current Internet. The performance results are encouraging: using commodity hardware, with a flow distribution derived from campus traffic traces, we can return multiple paths with associated performance information, while maintaining average latencies comparable to Internet round-trip times (a few tens of milliseconds). Our implementation scales easily through parallelism and admits various additional optimizations; we expect that with modest resources it should be possible to handle almost all queries in at most a few milliseconds. Our scheme does not rely on any particular addressing or naming scheme, and specifically does not rely on hierarchical, topology-based identifiers.

Our factored routing system allows access to a larger portion of the routing/forwarding design space than is possible in today's Internet. We have shown how it can be used with different forwarding schemes, including one based on SDN which we believe can be made backward-compatible, and another based on source routing with in-band policy compliance checks. We believe our proposed system, combined with suitable forwarding and policy enforcement mechanisms offers a number of advantages. The importance of our performance results is that they debunk the conventional wisdom that centralized routing computations do not scale.

References

[1] The CAIDA AS Relationships Dataset, June-August 2013, http://www.caida.org/data/active/as-relationships/.

[2] Ashok Anand, Fahad Dogar, Dongsu Han, Boyan Li, Hyeontaek Lim, Michel Machado, Wenfei Wu, Aditya Akella, David Andersen, John Byers, Srinivasan Seshan, and Peter Steenkiste. XIA: An Architecture for an Evolvable and Trustworthy Internet. In *The Tenth ACM Workshop on Hot Topics in Networks (HotNets-X)*, November 2011.

[3] Danny McPherson Jon Oberheide Craig Labovitz, Scott Iekel-Johnson and Farnam Jahanian. Internet inter-domain traffic. In *Proceedings of SIGCOMM*, pages 75–86, 2010.

[4] A. Dhamdhere and C. Dovrolis. Twelve Years in the Evolution of the Internet Ecosystem. *IEEE/ACM Transactions on Networking*, 19(5):1420–1433, Sep 2011.

[5] Richard Douville, Jean-Louis Le Roux, Jean-Louis Rougier, and Stefano Secci. A Service Plane over the PCE Architecture for Automatic Multidomain Connection-Oriented services. *IEEE Communications Magazine*, 46(6):94–102, June 2008.

[6] A. Farrel, J. P. Vasseur, and J. Ash. A Path Computation Element (PCE)-Based Architecture. RFC 4655, August 2006.

[7] P. Brighten Godfrey, Igor Ganichev, Scott Shenker, and Ion Stoica. Pathlet routing. In *Proceedings of ACM SIGCOMM*, pages 111–122, 2009.

[8] Timothy G. Griffin and Gordon Wilfong. An analysis of bgp convergence properties. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '99, pages 277–288, New York, NY, USA, 1999. ACM.

[9] Onur Ascigil James Griffioen, Kenneth L. Calvert and Song Yuan. Separating routing policy from mechanism in the network layer. In G. Rouskas B. Ramamurthy and K. Sivalingam, editors, *Next-Generation Internet Architectures and Protocols*. Cambridge University Press, 2011.

[10] Hyojoon Kim and Nick Feamster. Improving network management with software defined networking. *IEEE Communications Magazine*, 51(2):114–119, 2013.

[11] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazires, Michael Miller, and Arun Seehra. Verifying and Enforcing Network Paths with ICING. In *Proceedings of the ACM CoNEXT Conference*, December 2011.

[12] B. Raghavan, P. Verkaik, and A. Snoeren. Secure and Policy-Compliant Source Routing. *IEEE/ACM Transactions on Networking*, 17(3):764–777, June 2009.

[13] E. Rosen, A. Viswanathan, and R. Callon. Multiprotocol Label Switching Architecture. RFC 3031, January 2001.

[14] George N Rouskas, Ilia Baldine, Kenneth L Calvert, Rudra Dutta, Jim Griffioen, Anna Nagurney, and Tilman Wolf. Choicenet: Network Innovation through Choice. In *Proceedings of the Optical Network Design and Modeling (ONDM)*, 2013.

[15] Li Xiao, Jun Wang, King-Shan Lui, and Klara Nahrstedt. Advertising interdomain qos routing information. *IEEE Journal on Selected Areas in Communications*, 22(10), 2004.

[16] X. Yang. NIRA: A New Internet Routing Architecture. In *Proceedings of ACM SIGCOMM 2003 Workshop on Future Directions in Network Architecture (FDNA), Karlsruhe, Germany*, pages 301–312, August 2003.

[17] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David Andersen. SCION: Scalability, Control, and Isolation On Next-Generation Networks. In *IEEE Symposium on Security and Privacy*, May 2011.