2015

Design of a Scalable Path Service for the Internet

Mehmet O. Ascigil
University of Kentucky, onur@netlab.uky.edu

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Mehmet O. Ascigil, Student

Dr. Kenneth L. Calvert, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

DESIGN OF A SCALABLE PATH SERVICE FOR THE INTERNET

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By

Mehmet Onur Ascigil

Lexington, Kentucky

Co-Directors: Dr. Kenneth L. Calvert, Professor of Computer Science
and Dr. James Griffioen, Professor of Computer Science

Lexington, Kentucky

ABSTRACT OF DISSERTATION

DESIGN OF A SCALABLE PATH SERVICE FOR THE INTERNET

Despite the world-changing success of the Internet, shortcomings in its routing and forwarding system have become increasingly apparent. One symptom is an escalating tension between users and providers over the control of routing and forwarding of packets: providers understandably want to control use of their infrastructure, and users understandably want paths with sufficient quality-of-service (QoS) to improve the performance of their applications. As a result, users resort to various "hacks" such as sending traffic through intermediate end-systems, and the providers fight back with mechanisms to inspect and block such traffic.

To enable users and providers to jointly control routing and forwarding policies, recent research has considered various architectural approaches in which provider-level route determination occurs *separately* from forwarding. With this separation, provider-level path computation and selection can be provided as a centralized service: users (or their applications) send *path queries* to a *path service* to obtain provider-level paths that meet their application-specific QoS requirements. At the same time, providers can control the use of their infrastructure by dictating how packets are forwarded across their network. The separation of routing and forwarding offers many advantages, but also brings a number of challenges such as *scalability*. In particular, the path service must respond to path queries in a timely manner and periodically collect topology information containing load-dependent (i.e., performance) routing information.

We present a new design for a path service that makes use of expensive pre-computations, *parallel* on-demand computations on performance information, and caching of recently computed paths to achieve scalability. We demonstrate that, using commodity hardware with a modest amount of resources, the path service can respond to path queries with acceptable latency under a realistic workload. The service can scale to arbitrarily large topologies through parallelism. Finally, we describe how to utilize the path service in the current Internet with existing Internet applications.

KEYWORDS: Future Internet Architecture, Routing, Scalability

Mehmet Onur Ascigil

Student's Signature
Jan. 29, 2015

Date

DESIGN OF A SCALABLE PATH SERVICE FOR THE INTERNET

By

Mehmet Onur Ascigil

Dr. Kenneth L. Calvert

Co-Director of Dissertation

Dr. James Griffioen

Co-Director of Dissertation

Dr. Miroslaw Truszczynski

Director of Graduate Studies

Jan. 29, 2015

ACKNOWLEDGMENTS

I am very thankful to the following people, without whom this thesis would not be possible.

Foremost, I would like to express my deepest gratitude to my advisor Dr. Kenneth Calvert, not only for his strong guidance throughout my doctorate, but for being such a powerful role model for an academic career. His in-depth knowledge, preciseness, high standards in research, and strong work ethics have inspired me to not only be a better researcher, but also to be a better person.

Secondly, I would like to thank my co-advisor Dr. James Griffioen. His enormous patience, immense knowledge and strong encouragement were key motivations throughout my PhD. Our weekly discussions were always very interesting and a great learning experience. I am very grateful for giving me the opportunities to make publications and to attend conferences, which helped me to become a better researcher.

I also would like to thank the staff members of the Laboratory for Advanced Networking. Special thanks to William Marvel and Hussamuddin Nasir, who provided assistance in maintaining the servers and providing technical support with my experiments. I had wonderful memories working in the Hardymon building with fellow colleagues. Thanks for the interesting discussions and collaborations on the projects.

Finally, I would like to thank my parents for their unrelenting support and love. Being so far away from them for so many years, they were still my biggest support. They have sacrificed more than I think I will ever be able to, so that I can be happy and successful. I hope one day I will be able to repay them this incredible debt.

Table of Contents

List of Figures

Chapter 1

Introduction

A *routing architecture* "is a system that includes a way of naming entities such as networks, interfaces, topology aggregates; a way of exchanging information as to where the named entities are; a way of computing and selecting paths between named entities; and a way of causing traffic to take these paths." [21] An *end-system* or *host* is a computer that is connected to the network such as a laptop or a tablet, a smart phone, and so on. *Forwarding* is the relaying of packets from one network component to another via a *channel* that connects the components. *Routing* is the process of selecting paths in a network along which to send network traffic. An *autonomous system* (AS) is a set of routers and channels that are administered by a single entity in the Internet such as a university, company or Internet Service Provider (ISP).

> *This dissertation presents a scalable design for a path service that computes application-specific paths across the network on behalf of end-systems in a (loose) source routing architecture.*

Despite the huge success of the Internet, shortcomings in its routing and forwarding system have become apparent. The determination of a single "best" path between the source and the destination, which has long been considered to be the basic functionality of Internet routing, is no longer sufficient to meet the needs of new and heterogeneous applications that use the Internet. Real-time video streaming

and video conferencing applications are examples of applications that require different (i.e., application-specific) properties than those required by the Internet's older applications such as email and telnet. The current Internet routing and forwarding system is not always able to provide satisfactory quality-of-service (QoS) for real-time video applications, especially during the times of heavy Internet usage. Video is the dominant component of the Internet traffic during the peak hours of usage [54], and the performance of video streaming applications depends on the QoS provided by the Internet routing and forwarding system during these times of heavy load.

Interactive (e.g., online multi-player games) and real-time (e.g., video conference, VoIP) applications require the routing and forwarding system to provide robust end-to-end communication even in the face of failures (e.g., channel or node failure). While network failures are common in the Internet, the current Internet inter-domain (i.e., between ASs) routing system is known to suffer from slow convergence:[1] The average convergence time of the Internet inter-domain routing system upon changes in the topology was measured to be 3 minutes in past studies [52, 53]. Slow convergence of the Internet inter-domain routing system has been shown to hinder the performance of interactive applications like VoIP [51], multi-player games, and online financial transactions [78, 65].

While there are thousands to millions of unique communication paths between any two end-systems located in different parts of the Internet (see Section 6.2.1), end-systems have little to no influence on path selection in the current Internet architecture. Even the failure of a single network component (e.g., channel) on the "best" path between two end-systems can disrupt communication for long periods due to slow convergence of the routing system. Ideally, an application (or the operating system) running on an end-system could simply switch to a (pre-computed) alternate

[1] The current inter-domain routing and forwarding system requires all the network providers to synchronize (i.e., converge) on the same set of "best" paths to all destinations in the Internet; otherwise, packets can loop and will never reach their destinations.

path upon detecting a failure. In general, an end system can rapidly—that is, on the order of a few round trip times (RTT)—detect performance problems based on the feedback (or lack thereof) from the other end-system.

An important design factor that limits the influence end-systems have on the path selection process is the *conflation of routing and forwarding* [38, 17]. In the current Internet routing architecture, the routing functionality is performed in a fully-distributed way: each node along the communication path makes an independent decision as to where to forward packets next in a process called *hop-by-hop* routing. This means that both routing and forwarding are performed at each hop that a packet visits.

As opposed to the Internet's fully-distributed routing system, in a fully-centralized routing system (also known as *strict source routing*) one entity (not necessarily the sender) computes and selects the entire path (i.e., source route) that the packets are to follow using a topology map of the global network. Senders of traffic place these paths in their packets and each network hop simply forwards those packets according to the paths that they carry. However, ISPs usually have no desire to reveal topological details of their internal network. More importantly, ISPs have great incentive to control the routing of packets across their network in order to optimize the usage of their infrastructure.

A possible hybrid between the fully-distributed routing and the fully-centralized routing is *loose source routing*, where one entity picks the whole path, but in terms of high-level abstractions. Such a routing mechanism gives stake-holders (e.g., ISPs) local control over the routing within their infrastructure. For example, in a provider-level (i.e., AS-level in the current Internet architecture) loose source routing system, the paths carried in the packets determine the sequence of network providers (e.g., ASs) that the packets are to traverse, while the providers control the forwarding paths of the packets that transit their network.

Provider-level source routes can be specified at different granularities. One possibility is for packets to specify a sequence of providers (i.e., node-level path), and another possibility is for packets to specify the locations or interfaces where packets enter and leave each provider (i.e., channel-level path). In the following discussion, we assume the latter case. Channel-level routes result in a larger number of possible paths compared to node-level routing, as the large providers in the Internet tend to connect to one another at multiple locations.

A major challenge in deploying a loose source routing architecture is the computation and the selection of provider-level paths in a centralized way. Considering that there are over 35,000 inter-connected ASs in the current Internet, the resources required in collection and maintenance of provider-level information and the path computations can be nontrivial even if it is only based on connectivity. More importantly, computing application-specific paths (i.e., paths that satisfy multiple QoS constraints) requires NP-hard [79] computations when selecting a small number of paths that satisfy constraints on multiple additive routing metrics[2] from possibly millions of unique (provider-level) paths. In addition to the computational cost, computing application-specific provider-level paths requires periodically obtaining topology information including performance (i.e., QoS) information such as latency and available bandwidth.

In this thesis, we propose that the burden of application-specific path computation/selection and collection of topology information be placed on specialized servers that provide routing as a service, and we refer to this service as a *path service*. In a loose source routing architecture with a path service, end-systems (before they can initiate data communications with destinations) send *path queries* to the path service to obtain a set of paths that meet application-specific constraints. Upon receiving a query, the path service computes and returns a set of provider-level paths that satisfy

[2]A routing metric is *additive* if the join operation on the metric requires an addition operator (e.g., the latency metric).

the constraints specified in the query based on the most recently obtained topology information.

Because a (loose) source routing architecture with a path service brings several important advantages[3], especially with respect to fostering competition between providers to attract user traffic, we believe it is worth examining in detail. The architecture is not without challenges. In particular, obtaining paths from the path service adds additional latency before end-systems can send their first data packets. The additional latency cost is more significant for short-lived flows that last for only few round-trip times (RTTs). Because short-lived flows are common in the Internet communications, it is important to minimize the additional latency to obtain paths. Domain name service (DNS) [60] lookups, which happen prior to most data communications in the current Internet architecture, add an additional latency of an RTT to existing Internet communications. To be comparable with the latency of DNS lookups in the current Internet architecture, the path service must limit the latency in computing and returning paths for a "typical" request to at most an RTT. For example, 10 milliseconds is a reasonable bound, because it is less than most RTTs in the Internet. Our goal is to limit the latency incurred by the vast majority (i.e., 95%) of requests to be less than a few tens of milliseconds.

In addition to the time limit in computing query results, another requirement for a path service is being aware of changes in the topology information including performance information such as latency and available bandwidth across the forwarding paths of the network providers. However, awareness to changes in the performance information at very short time-scales (i.e., milliseconds) is neither scalable nor useful, because frequent collection of topology information leads to a large overhead, and the collected information is likely to become stale by the time the path is computed using the information. We propose that the path service use statistical performance infor-

[3]A detailed description of the advantages of a loose source routing architecture are given in Section 1.1.3.

mation that is likely to change more slowly (i.e., on the time-scales of minutes). Such information can be considered advisory as opposed to information that is guaranteed to be accurate.

The main challenge for the design of a globally-deployed path service that provides paths for all Internet communications is *scalability*. A scalable path service is one that computes and returns paths on the rough time-scale of an RTT, in a topology that is at least the size of today's Internet with at least as many possible provider-level paths. Although loose source routing systems have been proposed in the literature [18, 36, 14], the scalability of path computation and selection in these architectures is largely ignored.

In this thesis, we propose a design for a scalable path service that discovers the topology and computes application-specific provider-level paths for end-systems in a loose source routing architecture, while considering ways to deploy the service in the current Internet.

1.1 A Loose Source Routing Architecture

Our work assumes a clean-slate routing architecture where routing and forwarding are separated: the path service performs inter-domain (i.e., provider-level) path computation/selection for end-systems, and the network performs forwarding of packets according to the paths they carry. End-systems query the path service expressing their policies, such as path selection policy, as "hints", and the path service computes and returns a list of inter-domain paths according to the supplied policies. The policy hints include a traffic class, selected from a set of known classes, and other hints such as a blacklist of domains to be avoided in the paths returned.

1.1.1 Assumptions

We envision an inter-network consisting of independently-administered domains (i.e., providers) that are connected by bidirectional *channels*, each having a globally-unique identifier. An inter-domain path consists of a sequence of (inter-domain) channels starting with the egress channel of the source domain and ending with the ingress channel of the destination domain. A separate lookup system (similar to DNS) maps end-system identifiers to domains[4]. We assume that domains and end-systems have unique identifiers, but the structure of the identifiers is irrelevant to our scheme; they can be IP addresses or flat (i.e., unstructured) identifiers.

Deploying a source routing architecture in the real world would require payment mechanisms by which users could compensate individual (both local and non-local) network providers for their usage, as the providers would have no incentive to forward source-routed traffic without compensation. A scalable payment mechanism is outside the scope of this thesis but is on-going work that can be found in the ChoiceNet Project [83]. In addition to payment mechanisms, the architecture must enable transit providers to enforce their policies, so that they can control access to their resources. For example, it is in the interest of all transit providers to forward source-routed traffic only if it originated from, or destined to a paying customer. We briefly describe two possible policy-enforcement mechanisms that can be used with our system in Section 3.4.2.

In the proposed architecture, network providers reveal performance information in addition to their connectivity with their neighboring domains. Today, network providers reveal connectivity through the inter-domain routing advertisements they send. By collecting the routing advertisements from multiple vantage points, it is possible to (largely) infer the inter-domain topology [19]. However, the providers in

[4]The mapping of end-system identifiers to domain identifiers can be performed together (i.e., via the same protocol) with the path computation, to avoid adding an additional RTT to the latency.

the current Internet do not reveal performance information to users. Providers do not gain benefit from doing so because end-systems lack the ability to choose provider-level paths. In our proposed architecture, revealing (good) performance information benefits providers by attracting user traffic.

1.1.2 Topology Information

The path service periodically collects topology (i.e., routing) information from transit providers in the form of *relay advertisements*, which are offers to relay (i.e., forward) packets from an ingress channel to an egress channel. A relay advertisement contains the advertising domain, the ingress and the egress channels, performance information, and possibly other attributes. The actual set of routers used to relay packets across a domain are not revealed in the relay advertisements, in order to keep the internal topology of the domain hidden. Selection of the routers used to create a forwarding path for the packets using the relay is under the control of the domain. However, a domain can advertise multiple classes of forwarding services from a particular ingress channel to a particular egress channel, by advertising each service as a separate relay advertisement across the same domain. In that case, the end-systems can influence the path selection across a domain.

When multiple relays are offered between a pair of ingress and egress channels, each such relay is advertised with a unique traffic class label in addition to the physical ingress, egress channels to uniquely identify the forwarding service in the paths. The performance information in a relay advertisement provides a (statistical) summary of measurements collected from the forwarding path associated with the relay. When a domain utilizes multiple forwarding paths for a single advertised relay, it is responsible for maintaining similar performances across the individual paths.

1.1.3 Advantages

Application-specific path selection is an important advantage of the proposed loose source routing architecture over the current Internet architecture. The architecture gives users the ability to discover and choose provider-level paths, which creates competition between providers based on service quality assuming that providers get compensation for usage.

In a loose source routing system with appropriate payment mechanisms, users can reward providers that offer good service quality by choosing to route packets through their network (and paying for usage), and "punish" (i.e., by blacklisting or simply not using) providers that offer poor service quality. Such a system gives incentives for providers to innovate and compete based on service quality, in contrast to the current Internet routing architecture where there is no way to reward providers for innovation and quality.

Unlike routers in a hop-by-hop routing system that are required to pre-compute shortest paths to all destinations, in a loose source routing system the path service can deploy more flexible computational strategies such as computing paths when they are needed or a combination of on-demand and pre-computations. Another advantage of our scheme is evolvability: testing and deployment of new path selection algorithms only require modifications to the path service. In comparison, the fully distributed nature of the Internet routing is very difficult to change or evolve, as testing and deployment of new path selection techniques require global deployment. These advantages come with new challenges, especially with respect to the computation and the selection of paths, which we describe next.

1.2 Path Service Design Problem

Our study on the design of a scalable path service focuses on the two main "pinch points" that we believe are the primary challenges: i) cost of path computations, and

ii) bandwidth cost of periodically collecting routing information from the network. The overhead of path computation is directly related to the responsiveness of the service (i.e., latency in returning query results). Next, we discuss these two cost components.

1.2.1 Path Computation Cost

Even though the path computation/selection is performed in a centralized manner, the global path service is not necessarily a single centralized service. Because the end-systems in an access domain only require paths that originate from their local domain, a separate path service can be deployed at each access domain to compute inter-domain paths from one access domain to all other end-systems. Although deploying a path service in each access domain improves the scalability of the path computations, it leads to increased bandwidth overhead of collecting routing information from the providers to the path service, as we explain in Section 1.2.2.

In order to improve the responsiveness of the service, it is important to carry out the majority of the path computations before the arrival of queries as *pre-computations*. However, pre-computations must be repeated when the routing information used in the computation changes. Pre-computations can only be done if the rate of change in the input is smaller than the rate at which the pre-computations can be completed. Moreover, if the results of the pre-computation are not requested before the routing information changes, then the pre-computation was wasted. Therefore, an important factor in the design of a scalable path service is the *time-scales of change* in the routing (i.e., topology) information and the rate of requests for the pre-computed paths.

Routing information in the relay advertisements can be broadly categorized as *slow-changing* and *fast-changing* information. An important observation is that the connectivity of the domains and load-independent information such as a channel's

bandwidth capacity and propagation delay changes relatively slowly when compared to the load-dependent routing (i.e., performance) information such as latency and available bandwidth in the Internet. The inter-domain connectivity and the load-independent information change only with addition, upgrade, or removal of channels, which happen on the time-scales of days or months. Transient changes in connectivity can happen, however, with short-term (i.e., seconds to minutes) failures in the network components (e.g., channel or node failure). The end-systems are in the best position to rapidly detect and react to transient failures by simply switching to alternate (e.g., disjoint) pre-computed paths that they obtain from the path service. Therefore, it is not necessary for the path service to keep track of transient changes in the connectivity.

In addition to time-scales of change, another important consideration in the design of the path service is the *caching* of recently computed paths. Caching can potentially improve the responsiveness of the service, if there is a significant locality of reference in the destinations contacted by the end-systems. Our preliminary results, based on collected network traffic traces consisting of flows that originate from an access domain, indicate that there is a significant locality of reference in the destinations contacted by end-systems within an access domain. In particular, our measurements found that around 90% of flows leaving the University of Kentucky (UK) campus during an entire day were destined to fewer than 2000 destination domains (see Chapter 5 for details). Another important factor determining the effectiveness of caching is the time-scales of change in the performance information, because cached paths are not guaranteed to be valid (i.e., may no longer satisfy the application-specific constraints) upon changes in the information. As we pointed out earlier, the path service only keeps track of slower-changing performance information such as statistical performance information that is expected to change in the time-scales of minutes. As we will show, caching is effective in reducing the path computation cost even when its contents expire on the

time-scales of minutes or less.

1.2.2 Routing Information Collection Cost

In the routing architecture that we consider, transit providers perform measurements and periodically send the summaries of their measurements (i.e., performance information) for each of their relays (directly) to all the path services as part of the relay advertisements. Even though the path service uses slow-changing, statistical performance information, the overhead of periodically sending relay advertisements with performance information can be significant on the network. The overhead is greater when the path service is deployed at each access domain to reduce the computational cost, because the transit providers must periodically send relay advertisements to all access domains. However, our scheme does not mandate that each access domain have a path service; a path service can serve multiple access domains.

There are possible ways to reduce the bandwidth overhead of the relay advertisements. One possibility is to build specialized routes to distribute the advertisements. For example, one can construct multicast trees for each transit provider to distribute its information to all path services using the smallest possible number of inter-domain channels, assuming that the architecture supports forwarding of packets with paths containing trees and duplicate packets when necessary. Also, the providers can use compression mechanisms to reduce the size of the relay advertisements that they send to the path service.

A more effective approach to reduce the overhead is to reduce the frequency of relay advertisements. An important observation is that the path service can use a *pull model* (i.e., on-demand) to collect relay advertisements from the transit providers. With an on-demand approach, the frequency of collecting the performance information from different parts of the network is one of the "knobs" that can be controlled in the design. The path service can implement various on-demand collection strategies to reduce

the overhead of advertisements. For example, the service can collect performance information from unpopular regions of the inter-network less frequently than from the popular regions. Because the majority of the inter-domain traffic originating from one access domain is destined to a small number of (popular) destination domains, parts of the inter-network topology (i.e., closer to popular destination domains) are likely to be more relevant to the path service than some other parts. Furthermore, frequent collection of relay advertisements from backbone providers where the largest number of flows are aggregated may be unnecessary, because the change in the statistical performance information is expected to be slow at such locations. A comprehensive solution to minimizing the overhead of performance information collection on the inter-network is provided by Wu [84], and it is outside the scope of this thesis.

1.3 Our Solution

Our approach is to perform the most expensive computations *in advance* on the slow-changing information (i.e., connectivity). For a network the size of the current Internet this produces over a billion paths from a single source domain to all destination domains which we sort and then select paths using fast-changing information during the query-time (i.e., on-demand). The pre-computation stage is performed only once, and only incremental changes are made upon changes in the slow-changing routing information. In order to improve the scalability of on-demand computations, our approach makes heavy use of *parallel* computations. In particular, the set of pre-computed paths are partitioned and assigned to a set of threads or processes[5], where each performs a part of the path computation in parallel.

Our hypothesis is that through path caching, a combination of pre-computations and on-demand computations, parallel computations, and the use of slow-changing (i.e., statistical) performance information, a scalable path service with acceptable

[5]Our approach does not rely on shared-memory for inter-process communication.

latency (i.e., comparable to a RTT) can be designed and implemented. We test our hypothesis in a setting where a path service computes paths for end-systems in a single access domain. Our workload model includes a set of queries that are generated using real Internet traffic traces collected at a university campus and relay advertisements generated using an accurate model of the Internet inter-domain topology.

1.4 Summary of Our Results

Our main result is that using commodity hardware with a modest amount of resources, the path service can achieve acceptable latency under a realistic workload. The service can support topologies with an arbitrarily large number of possible paths by scaling the computations through parallelism. Also, the caching of recently computed query results effectively reduces the computational cost under realistic workloads because of the high locality of reference in the set of destinations contacted by the end-systems.

The path service can also compute disjoint paths (paths that share no channels in common) with negligible additional overhead. In particular, the path service processes "disjoint path queries" that provide an input path and request application-specific paths that have the fewest channels in common (i.e., maximally disjoint) from the input path. Such queries are useful, for example, when end-systems need to request a disjoint path from a recently failed path in order to recover from failures. Our results show that the path service can provide application-specific, maximally disjoint paths with negligible overhead. The service minimizes the overhead of disjoint paths by using compact data structures, i.e., Bloom Filters [15], to represent paths and by using efficient comparison methods on the filters.

In this thesis, we also describe a loose source routing system with a path service that can be deployed alongside the current Internet architecture, and we describe a prototype implementation. The implementation of the system includes a "wrapper" library and packet forwarding elements. The wrapper library enables legacy (unmod-

ified) Internet applications to use the loose source routing system and select paths according to application-specific constraints. The forwarding element is a software router implementation that forwards source-routed packets produced by the wrapper. The implementation of the system uses a data-plane policy-enforcement mechanism for transit providers that is inspired by Platypus [66], where policy-compliant packets (e.g., packets that are paid for) carry tokens that are verified by the forwarding elements of the providers. We also briefly describe how the loose source routing system can work with partial deployment—that is, when a subset of the domains in the Internet forward source routed packets and send relay advertisements to the path service—in order to demonstrate that the system is incrementally deployable.

We begin by describing Internet routing and forwarding in Chapter 2. The loose source routing architecture and the challenges of designing a scalable path service are described in Chapter 3. Chapter 4 describes relevant prior work and its relationship to the thesis. Chapter 5 describes the path service design in detail and Chapter 6 presents the evaluation of the design. In Chapter 7, we describe a loose source routing architecture with the path service, which can be deployed alongside the current Internet routing architecture. Finally, in Chapter 8, we present our conclusions and discuss possible future work.

Chapter 2

Background

To better understand the limitations of the current Internet, we start by explaining routing and forwarding in the current Internet. First, we briefly describe basic concepts related to routing such as the Internet Protocol (IP) and the Autonomous System (AS). Then, we discuss intra-domain routing and forwarding approaches in the Internet. Finally, we describe the inter-domain routing and forwarding system of the Internet.

2.1 Internet Protocol

Internet protocol version 4 (IPv4) and version 6 (IPv6) are connectionless protocols used in packet-switched computer networks for transmitting blocks of data called datagrams[1] between hosts. The Internet Protocol (IP) provides a best-effort communication facility—no acknowledgement of successful receipt of datagram messages is defined by the protocol. In addition, there is no assurance of in-order delivery of packets, and there is no avoidance of duplicate deliveries. IP addresses identify network interfaces. Each network interface participating in IP is assigned a fixed-length identifier: a 32-bit address in IPv4 and a 128-bit address in IPv6. Datagrams carry source and destination IP addresses.

As the successor of IPv4, IPv6 is proposed as a long-term solution to cope with

[1] The term *packet* is a more general name for blocks of data that are delivered either reliably (using mechanisms that are not part of IP) or not.

the exhaustion of the 32-bit address space in IPv4. IPv6 addresses are written using eight groups of four hexadecimal digits (16-bits) separated by colons. To simplify IPv6 address representation, the Internet Engineering Task Force (IETF) proposed two conventions: (i) "::" is used (at most once in an address) to shorten one or more 16-bit groups of hexadecimal digits containing only 0's, and (ii) leading zeros in a 16-bit groups are suppressed [46]. Using these conventions, the IPv6 address "0004:0dfb:85a3:0000:0000:0000:0000:0001" can be represented in a compact way as "4:dfb:85a3::1".

Some IPv4 header fields that are rarely used have been made optional in IPv6 to reduce the processing cost of packets. These options are moved to *extension headers*, which allow flexibility in the length of options and greater extensibility by allowing for new options to be added in the future. One example is a fragment extension header, which contains information to support breaking datagrams into smaller fragments and then reassembling them. We use the IPv6 extension headers to carry source routes in IPv6 packets in Chapter 7.

2.2 Autonomous System

An *Autonomous System* (AS) is a set of routers and channels that are administered by a single entity such as a university, a company or an Internet Service Provider (ISP). An AS represents a single, coherent routing policy to the Internet. As we explain later in Section 2.3.2, AS routing policies determine whether an AS shares its routes with neighboring ASs. These policies are influenced by the business relationships between ASs. The business relationships between neighboring ASs can be classified into two broad categories: (i) a provider-to-customer relationship, where a customer AS pays a provider AS for sending and receiving traffic and (ii) a peer-to-peer (i.e., peering) relationship, where both ASs find it mutually beneficial to exchange traffic without exchanging money. Based on their traffic patterns, ASs can be broadly classified into

stub and *transit* ASs. A stub AS carries traffic that either originates or terminates within that AS. On the other hand, a transit AS carries traffic that does not originate or end within the AS.

The customer-provider relationships in the Internet form a hierarchy among the ASs. Based on their location in the hierarchy, ASs can be broadly classified into tier-1, tier-2, and tier-3 ASs. Tier-1 ASs form the root of the hierarchy and typically have peering relationships with almost all other tier-1 ASs to form a highly-connected *core* region. Tier-1 ASs typically span a large geographical area including multiple continents. Tier-2 ASs have regional customers and have one or more tier-1 providers. Tier-3 ASs have small number of local customers and have one or more tier-2 providers.

One or more blocks of IP addresses called *IP prefixes* are assigned to autonomous systems. An IP prefix is a block of addresses with the first N bits uniquely identifying the network and the rest of the bits identifying the locations within the network. The first N bits are called the prefix and "/N" denotes a prefix of N bits long. An example of a IPv6 address prefix is "3001:bd8::/32", where the first 32 bits are represented with two 16-bit groups of hexadecimal numbers.

The assignment of IP addresses to ASs is made in one of the two ways: (i) the customer AS obtains a block of IP addresses from its provider or (ii) the AS obtains a block of IP addresses from a Regional Internet Registry (RIR). In addition to IP prefixes, each AS has a unique 32-bit autonomous system number (ASN) which uniquely identifies the AS in the Internet. Autonomous systems create a two-level hierarchy for routing in the Internet. Routing between ASs is called *inter-domain routing* and routing within the ASs is called *intra-domain routing*. Next, we describe routing and forwarding in the Internet.

2.3 Internet Routing and Forwarding

Routing in traditional IP networks is performed in a *fully-distributed*, *hop-by-hop* manner. Each router makes an independent routing decision to determine which "next-hop" neighbor the packet should be forwarded. ASs use an intra-domain routing protocol to determine how to route packets within the AS and an inter-domain routing protocol to determine how to route packets from and to other ASs.

2.3.1 Intra-domain Routing and Forwarding

Intra-domain routing protocols can be divided into two categories: (i) distance-vector and (ii) link-state. With link-state protocols, each router advertises its neighbors to all other routers in the domain. In the distance-vector routing protocols, each router shares its routing table—a data table that contains the *next-hop* to forward packets with the "cost" of doing so for each destination—only with its directly connected neighbors. Initially, the routing tables of nodes contain only their direct neighbors and the cost of reaching them. The cost is a single value that can be computed using various metrics such as hop count, latency, and available bandwidth. In link-state protocols, each node composes the global map of the topology using the link-state advertisements from all other nodes and uses a shortest path routing algorithm such as Dijkstra [28] to compute paths.

An important task of intra-domain routing and forwarding systems is to select paths with the goals of efficient use of network resources and good network performance. Traditional intra-domain protocols route packets along the shortest (i.e., minimum hops) paths, which can cause congestion on the channels where multiple shortest paths overlap. Equal-Cost Multi-Path (ECMP) algorithm [40] enables hop-by-hop routing protocols to support sharing of traffic among multiple shortest paths. Even though ECMP allows load-balancing traffic across multiple shortest paths, it does not support load-balancing among paths with different costs.

Because of the limitations of the hop-by-hop routing, centralized routing approaches are becoming popular. One such approach is Multi-Protocol Label Switching (MPLS) [69]. In MPLS, packets are encapsulated with a stack of labels which determines their paths across the network. Labels are pushed on the incoming packets at an entry point of the network and popped off the packets at an exit point. Unlike ECMP, MPLS supports load-balancing across paths with different lengths. Another important advantage of path-oriented approaches over traditional hop-by-hop routing is fast recovery from failures. In particular, an MPLS network can rapidly recover from failures by diverting the traffic to pre-computed backup paths. On the other hand, hop-by-hop routing requires a period of convergence, which can last for seconds to minutes, to recover from failures.

Existing approaches to centralized routing in the Internet is mostly limited to individual ASs, where all the network devices are conveniently under a single management. At the inter-domain level, routing is performed in a fully-distributed manner, as we explain next.

2.3.2 Inter-domain Routing and Forwarding

While the routing decisions within a single domain are based mostly on efficiency and performance, inter-domain routing decisions are mostly based on *routing policies*. A main challenge of inter-domain routing is to ensure that domains forward traffic that is policy-compliant. In particular, providers have no incentive to transmit traffic which does not generate revenue. An ISP, call it S, usually provides full transit of incoming and outgoing packets for its own customers and provides some transit for the customers of the other ASs involved in a peering relationship with S.

The only inter-domain routing protocol in use today is the Border Gateway Protocol (BGP), version 4 [68]. Neighboring ASs use BGP route advertisements to exchange routing information. A BGP route advertisement carries a particular IP

prefix. Consider Figure 2.1 where the AS P advertises its prefix "1280:1630::/32" to AS A. AS A propagates (i.e., exports) the advertisement to AS B, which implies that AS A is willing to forward packets destined to P through its network.

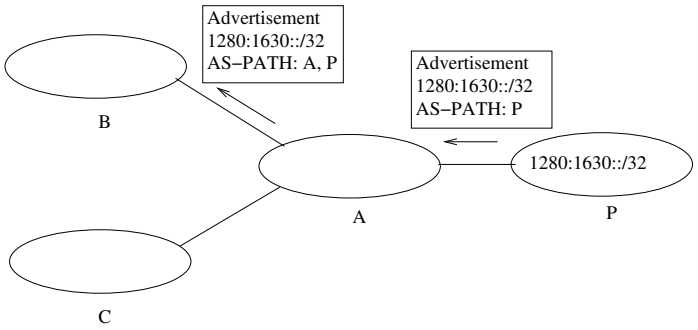

Figure 2.1: Provider P advertising its prefix "1280:1630::/32" to provider A, and A exports the advertisement to provider B.

Unlike intra-domain routing protocols, the BGP advertisements do not provide cost metrics such as the number of hops. Instead, the BGP advertisements contain *reachability* information in the form of AS-level paths. The AS-level paths in the advertisements are placed in the AS-PATH fields. Originally used only for loop-detection in the paths, AS-PATH is used to make policy-based path selection decisions by the ASs. In particular, when multiple routes to reach a particular prefix P are received, the AS selects only one of the routes as the "best" route to P and exports only the best route to its neighbors. The selection can be based on the length of the AS-PATH or other considerations based on the ASs on the AS-PATH as specified by the local AS policy.

BGP converges when the entire inter-network is *synchronized* to the same set of "best" paths to all prefixes. Lack of synchronization in parts of the Internet can lead to serious problems such as routing loops. The global synchronization is known to take on the order of minutes, during which the end-systems can be disconnected [52]. Because BGP limits ASs to select a single "best" path for each destination, multi-path routing has very limited support in the Internet at the inter-domain level.

The inter-domain topology of the Internet can be largely inferred from the BGP advertisements. From the advertisements, one can construct an AS-level graph in the following way: an edge exists between AS_i and AS_j if and only if AS_i advertises some prefix to AS_j or vice versa. Various organizations such as the Route Views project [75] collect BGP advertisements from multiple locations in the Internet and then make their data available for use by researchers.

In summary, the BGP protocol provides mechanisms for ISPs to perform a distributed computation to determine a single "best" path to all destination IP prefixes. BGP requires global synchronization to work properly, and multi-path routing usage is very limited. Because the vast majority of the paths that end-systems use in the Internet traverse multiple domains, the limitations of the inter-domain routing and forwarding system impacts almost all the communications in the Internet.

2.4 IP Routing Compared to a Path Service

While IP routing has worked well, it has several limitations that are not present in a loose source routing approach that uses a (logically) centralized path service to compute paths. The centralized path computation approach separates routing from forwarding. By separating routing and forwarding, one can achieve faster convergence time and faster recovery from failures. Also, the end-systems in a loose source routing architecture can use multiple paths with different characteristics instead of a single "best" path as in BGP. The current Internet architecture has embedded business models with no incentives for ISPs to innovate (i.e., improve service quality) in order to attract traffic from end-systems. In a loose source routing architecture, end-systems can select a path and (given appropriate mechanisms) pay for the route, which provides incentives for ISPs to offer better service quality. However, a loose source routing architecture with the path service brings new challenges. In particular, the path service must periodically collect routing information containing performance

information and compute/select paths in a timely manner. In this thesis, we mostly focus on the latter challenge.

Chapter 3

A Clean-slate Approach to Inter-domain Routing

We consider a clean-slate approach to inter-domain routing in which inter-domain paths are computed in a centralized manner and are selected prior to forwarding. In this approach, a *path service* discovers topology and computes domain-level[1] paths between source and destination domains. Applications, before sending packets to destinations, request a set of paths from a path service. The resulting paths returned from the path service (or a subset) are then used by the application to send data packets.

Routing and forwarding are separated: the path service performs routing (i.e., selection of domain-level paths), and the domains perform forwarding of packets according to the domain-level paths carried in the packets. Such a system offers a number of architectural advantages over the current Internet routing system, including (i) native support for multipath forwarding; (ii) the ability to apply different path-selection policies for different applications or user classes that can evolve over time; and (iii) the ability to react to changes (in topology, etc.) without waiting for the network to converge.

More precisely, we consider the design of an inter-domain path service, which

[1]We use the term domain or provider instead of "AS" because of the latter's connection with BGP in today's routing and forwarding system.

discovers and computes domain-level paths between source and destination domains. Applications send a query to the service to obtain paths from the service and the paths returned from the path service (or some subset of them) are then used by the network for *forwarding*.

3.1 Overview

We consider a network made up of separately-administered domains, with no central administration (similar to today's Internet). We classify domains as either transit or access (i.e., stub), as it leads to a simpler and cleaner model. Access domains exist to provide service to users. Transit domains exist only to interconnect access domains. Thus, every packet originates from and is destined to, an access domain. This is different from today's Internet, where ASs can play both access and transit roles.

The separation of roles requires dividing the domains with mixed roles into a transit domain and one or more access domains, where the transit domain interconnects the access domains to each other and to the rest of the Internet. The provider networks in the current Internet are commonly organized as a backbone area that interconnects one or more stub areas. Dividing providers with such an organization can be straightforward since the backbone area performs the functionality of a transit domain, and each stub area (or each group of directly-connected stub areas) can become an access domain. Additional details on the access and transit separation in our model and its implications on the Internet inter-domain topology are provided in Chapter 6.

Domains are connected via named *channels*. The basic channel abstraction is a point-to-point, bidirectional, best-effort packet transmission facility. Channels are named with globally unique channel identifiers (CIDs) whose structure is irrelevant to our scheme; they can be the IP addresses of the two endpoints or flat (i.e., unstructured) identifiers. A channel is created between a pair of domains if and only

if both deem it mutually beneficial. In our model, one domain connects to another for the same reason ASs connect in the Internet today: to get access to a greater portion of the Internet. However, our model has no customer-provider relationships between domains; all relationships are *settlement-free*, similar to peering. Because a settlement-free relationship only requires two domains to pay the cost of connecting a channel, access domains can peer with any number of transit domains—that is, multi-homing comes "for free". A possible implication of low-cost peering is the increase in the multi-homing degrees of providers. In our experiments, we consider access domains with larger multi-homing degrees than those in the current Internet.

All transit policies are strictly local—that is, transit providers have no say over what happens to traffic after 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. The mechanism used to convey packets between ingress and egress is opaque to our scheme; a domain might use MPLS, Software-Defined Networking (SDN) [31], IP tunneling, or regular intra-domain (IP) routing.

The *path service* is a crucial entity in our architecture. One or more (possibly competing) instances of path services serve the users in the global Internet. The path services collect routing 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 a path service. The resulting information flow in our model between all these entities is given in Figure 3.1.

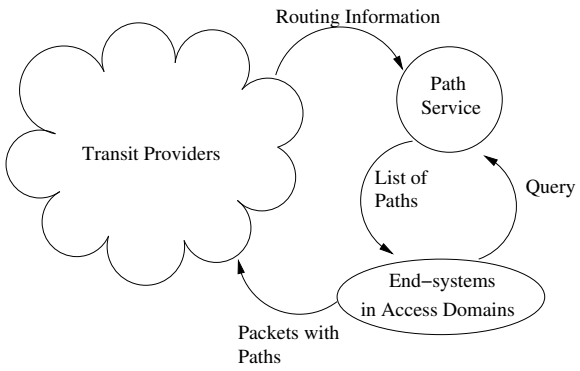

Figure 3.1: Entities and information flow in our model.

3.2 Model of Operation

The path services periodically collect topology information and use them to construct paths between access domains. Paths consist of a sequence of channels starting with the egress channel of the source domain and ending with the ingress channel of the destination domain.

The topology information from each domain must contain enough description of the domain's topology for the path service to be able to compute paths and must not reveal the inner details of the domain's topology. This is accomplished by advertising an abstraction of the domain's topology using a set of edge-to-edge (i.e., ingress to egress) *relaying services* described next.

3.2.1 Relay Advertisements

The path service *periodically* collects *relay advertisements* from transit domains. A relay advertisement is an offer to relay packets from an ingress channel to an egress channel, and includes the *offering provider*, the *Channel Ids (CIDs)* of the ingress and the egress channels, plus *performance information* and other attributes. The performance information of a relay advertisement contains a statistical summary of the recent measurements of load-dependent quality-of-service (QoS) attributes such as latency and available bandwidth. We describe the performance measurements in

Section 3.2.2.

Transit domains can provide multiple "virtual" relaying services across a pair of physical ingress and egress channels. In that case, each virtual relay is advertised separately with a *traffic class* label, along with its ingress, egress channels, performance information, and other attributes. When sending data through a relay, the packet must carry the traffic class in order to identify which virtual relay is being requested. The inter-domain paths returned by the path service contain a traffic class label for each virtual relay that the path contains.

Unlike the Differentiated Services (DS) [62] field in IP packets, a traffic class label is associated with a relay not the entire path. Therefore, different combinations of traffic classes can be selected on a path. Also, a domain which is not involved in providing a particular traffic class does not need to support it or even be aware of the traffic classes supported by other domains. The label values are not necessarily tied to a small number of predetermined traffic classes as in the DS field. Instead, each domain can use arbitrary traffic class labels for its services that do not necessarily match the label for the same service provided by another domain.

Relay advertisements reflect local policies of the domain. A domain that does not wish to forward packets from an ingress channel i to an egress channel e, simply does not advertise a relay from i to e. The number of relays advertised by a domain with n channels, offering C traffic classes per relay is at most $n \times (n-1) \times C$. The number of traffic classes depends on the variety of applications using the inter-network. Existing research in Internet traffic classification methods to map the traffic from different applications to different (QoS) classes uses four main classes: throughput-intensive, low-latency (interactive), web, and real-time [70]. In our experiments, we use a similar number of traffic classes (see Section 6.1.1) for applications to choose from.

3.2.2 Measuring Performance Information

Transit domains perform measurements to generate performance information reported in the relay advertisements, such as latency and available bandwidth information. The performance information includes the performance on the forwarding paths across the domain and on the ingress and egress inter-domain channels. Measuring the performance information on inter-domain channels requires cooperation of the neighboring domains. We assume that such cooperations can be established by joint interests of the domains. The neighboring domains must make the necessary arrangements to advertise the performance of their common inter-domain channels (e.g., the domain with the smaller identifier reports the inter-domain channel measurements in its relay advertisements).

It is the provider's responsibility to ensure that the performance information is accurate for the advertised relays; otherwise, the provider may lose business. If multiple forwarding paths are utilized across a domain for a relaying service, the provider must ensure that the advertised performance information for the relaying service accurately reflects all the forwarding paths. In order to achieve that, the provider may need to employ traffic engineering (TE) mechanisms to maintain similar performances across all the forwarding paths for a relaying service. For example, a flow-based traffic distribution—using hashing on source and destination addresses and possibly other fields of the header in the incoming packets—is commonly used to distribute load across equal cost paths in IP networks, and it is effective in networks where the number of flows is large and heterogeneous [12]. If the performances of paths differ significantly, despite traffic distribution and other TE mechanisms, the domains can alternatively report a lower-bound of the performances of paths—that is, the measurements on the path with the worst performance among all the forwarding paths of the relay.

The latency on a path from an ingress channel i to an egress channel e can be

obtained by active measurements, where packets are periodically sent from i to e containing the exact time—preferably in microsecond-level granularity—at which the packet was sent (i.e., time-stamp value) from i. Once a measurement packet reaches e, the difference between arrival time of the packet to e and the time-stamp in the packet gives the latency of the packet from i to e. This type of active measurements involving timestamps requires that the clocks of the measurement nodes to be synchronized. We assume that microsecond-level synchronization is possible across nodes in a network using existing techniques such as reference broadcasts [30].

Measuring available bandwidth is more difficult than measuring latency because it requires computing the unused bandwidth—total capacity minus total volume of traffic—on the intra-domain channels of the forwarding paths. There are existing active measurement approaches; one that is most popular in the literature is packet probing, sometimes referred to as *packet pairs* [41]. In packet probing techniques, the ingress node sends a pair of packets (one immediately after the other) to the egress node, and the egress node observes the changes in the spacing between the two packets to estimate the volume of the cross (i.e., competing) traffic. The accuracy of these techniques at any given point in time is questionable. However, taken on longer time scales, the accuracy improves and gives reasonable estimation of the available bandwidth [72].

The back-to-back probing packets, when time-stamped at the ingress point, can also provide latency measurements in addition to available bandwidth estimation. The difference between the latency measurements of the back-to-back packet pairs provides a fine-grained latency variation on the forwarding path in addition to available bandwidth estimation. Transit domains can collect latency variation measurements —also called jitter [26]—to be reported to the path service in addition to the latency and the available bandwidth measurements.

The path service only maintains performance information for inter-domain paths

consisting of the inter-domain channels and the forwarding paths across the transit domains. Figure 3.2 shows the "access paths" and the inter-domain path between two end-systems. Our scheme ignores the performance information of the access paths between the end-systems and the border nodes of their local access domain. However, it is possible to deploy an intra-domain path service that computes access paths based on performance information (of intra-domain paths) for end-systems.

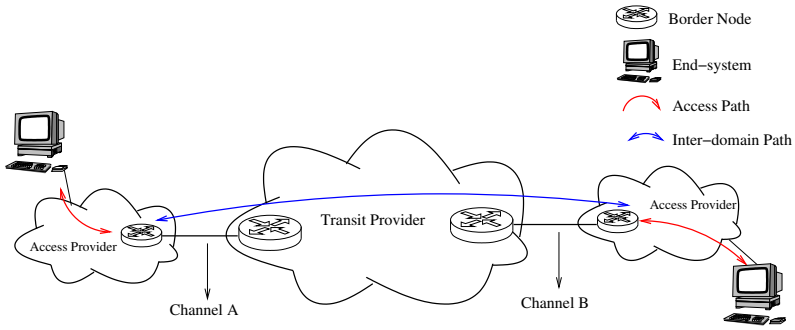

Figure 3.2: Two access domains connected through a transit domain.

3.2.3 Obtaining Paths from the Path Service

Each source is configured with paths to one or more path services through a bootstrapping protocol similar to DHCP [29]. A loose source routing architecture can also use mechanisms that enable automatic discovery (i.e., without manual configuration) of the path services by broadcasting their existence. A scalable broadcasting mechanism to bootstrap distributed services in a source routing architecture is described in [10], and the details of the mechanism are outside the scope of this thesis. End-systems obtain path service-specific arguments through the bootstrapping protocol such as a *time limit* for the service to return query results upon receiving a query. End-systems use the time limit to determine when to resubmit a query to the path service in case of no response from the service.

To initiate an inter-domain flow, a source transmits a path query to a path service, which may be local or located in another domain. The path query contains the source

and the destination domain identifiers (similar to an AS number), the number of paths requested, and (optional) policy "hints", i.e., characteristics of the paths desired. Such hints 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.

In order for a sender to obtain its own source domain's number and the domain identifier of endpoints, additional mechanisms are necessary. Next, we discuss a directory service that maps the endpoint (sender and the final destination of packets) identifiers to domain identifiers.

3.2.4 Mapping Endpoints to Domain Identifiers

In reality, packets will travel between finer-grained entities, and some means is required to transform endpoint identifiers into (access) domain identifiers. The endpoints can have either hierarchical (e.g., IP addresses) or flat identifiers. In the case of hierarchical endpoint identifiers where the domain identifier has no relation to the IP prefixes in the domain, a distributed directory system similar to DNS can be used to store endpoint identifier to domain identifier mappings. If the domain identifier is based on the prefixes then there is no need for a DNS-like system. In case of non-hierarchical names, a distributed hash table can be used. The directory system must allow mobile endpoints to register their requests when moving from one access domain to another. There are existing proposals [76, 57, 32] for a directory system that maps identities to (dynamic) locations (e.g., domain identifier) and can handle mobility in an efficient and scalable way. Our system can use any of these existing solutions.

3.3 Problem Statement

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 to improve computational scalability. In particular, since each access domain only needs paths from itself to other access domains (and possibly paths from other domains to itself), the computation of paths from one access domain can be handled by a local path service as shown in Figure 3.3. Moreover the local path service needs to compute paths from the local access domain to all other domains, not all-to-all. This greatly reduces the computational complexity.

Deploying a path service at each access domain improves the computational scalability of the global path service, but at the same time increases the bandwidth overhead. In particular, path services at each access domain must periodically collect relay advertisements from each transit domain. There are possible ways of reducing the bandwidth overhead of the advertisements. Our approach is to reduce the overhead by controlling the frequency of advertisement collection by the path service. An important observation is that the path service can use a pull (i.e., on-demand) model to collect relay advertisements. By using an on-demand collection mechanism, the path service can control the frequency at which the relay advertisements are collected from the transit domains. For example, it is expected that the large (e.g., backbone) transit domains, where a large number of flows are aggregated, have relatively stable performance measurements compared to smaller transit providers. The path service can also select the frequency of relay advertisement collection from different transit domains according to their "popularity". For example, if a set of transit domains are heavily used (i.e., popular) by the end-systems to relay packets compared to the other transit domains, then the path service can collect relay advertisements from those domains more frequently than from the others. A comprehensive solution to collecting performance information with minimal bandwidth overhead is described

by Wu in [84]. In this thesis, we mostly focus on the computational overhead of computing paths rather than the bandwidth overhead.

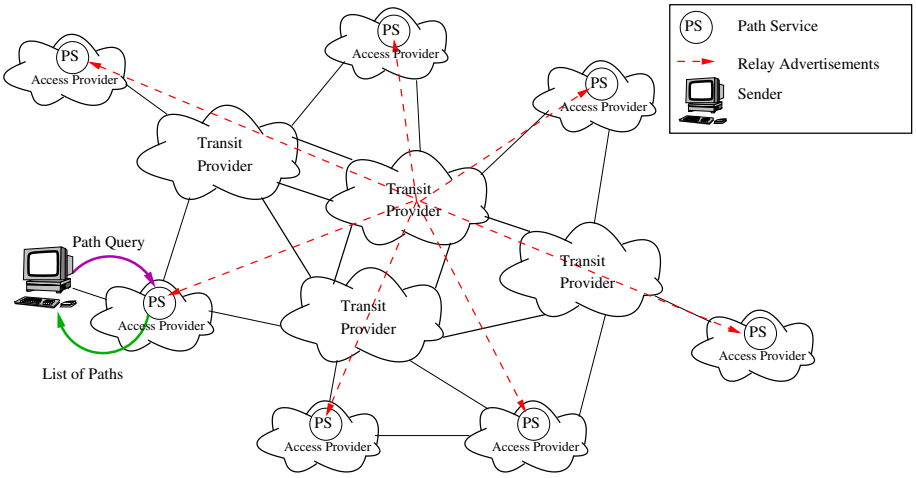

Figure 3.3: Inter-network with transit providers and access providers with local path services.

The conventional wisdom on source routing mandates that accurate topology maps of the global Internet be used to compute end-to-end paths, and accurate topology maps require awareness to changes in the routing state that happen at very short time-scales (in milliseconds). Based on this conventional wisdom, source routing is very hard to scale because of (i) the high frequency of route updates, and (ii) path services repeating path computations with the frequent arrival of new routing information rather than being able to use pre-computed paths.

Our hypothesis is that awareness of changes at very short time-scales is not necessary because end systems can observe the state of the relevant parts of the network by performing end-to-end measurements. The measurements can either be made in-band during a data transfer or by simply sending probe packets. Instead of notifying the path service of short-term changes in the network, transit providers can provide performance information that contains information such as a statistical summary of measurements over a longer (e.g., 10 seconds to minutes) period.

The design of a scalable path service is challenging due to computational com-

plexities and having to return paths to end systems, upon path requests, in *a timely manner*. The path service must respond to queries (i.e., compute and return paths) with latency comparable to the time it takes it takes for users in the current Internet to resolve a DNS query, prior to sending their first data packets. In particular, the time to obtain paths from the path service should not exceed an Internet round-trip time of up to a few tens of milliseconds for the majority (say, 95%) of the queries.

Periodically collecting large amounts of routing information containing dynamic performance information is also a challenge. However, we only consider domain-level routing information that is aggregated to hide the inner details of the domains. Therefore, the amount of routing information is on the order of the number of transit domains in the global Internet and not the number of routers. However, each transit domain can produce routing information (i.e., relay advertisement) that is quadratic in terms of the number of channels that the domain has (in theory, a domain can advertise multiple relays for each of its ingress, egress channel pairs). The path service must keep up with the advertisements that are periodically obtained from the transit domains. More importantly, the service must provide the most recent available information about the performance characteristics of any returned paths, to help the application—the path characteristics are derived from the information for the individual relays making up the path.

Given the above model, the challenge is to design a path service that can provide, on demand, multiple paths that satisfy a set of application-specific constraints from a set of source domains (one or many) to any given destination access domain in a timely manner. It must perform this function in a graph at least the scale of today's Internet AS graph with 10^5 – 10^6 nodes. On-demand computation of paths that satisfy multiple constraints require the path service to consider thousands to millions of possible paths between access providers in a limited amount of time. The computation should use the most recently collected performance information. The

second challenge is to scale to large networks with arbitrarily large number of paths between access domains.

3.4 Assumptions

In this section, we list our assumptions about the routing architecture that the path service requires. In Section 3.4.1, we describe a possible money flow between the stakeholders—that is, the path services, the domains, and the users. Then, we explain mechanisms by which transit domains can enforce their transit policies in Section 3.4.2.

3.4.1 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 services 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 services; path services pay transit providers for the use of their relay services. Transit providers should 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 service acts as a clearinghouse for payments from users to transit providers. Such payments might occur on various time scales. In the limit, a path service might charge for each individual path query it fulfills. More plausible, however, would be a subscription type service, in which users (or their access domains) contract with a path service for a longer time (e.g., over a month period) interval.

One of the features of this system is that money follows traffic flow, while traffic flow is only weakly constrained. 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.

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

3.4.2 Policy Enforcement

It remains to be shown 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 service), 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 that correspond to an SDN-based forwarding approach and a source-routed approach to forwarding.

- *Stateful, Software Defined Networking (SDN) based enforcement*: In this method, when a path service 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 service 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 no co-ordination is required between the SDN controllers for the different domains. The advantage of this approach is that it can be made backward-compatible, using existing protocol headers. It has the usual disadvantages of stateful forwarding in networks, and also increases the pre-transmission latency. We discuss SDN in Section 4.1.

- *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 service and the transit domain, and containing an expiration time. Such a scheme for verifying policy-compliance of source-routed packets is described in Platypus [66], for example. The ingress switch in each transit domain verifies compliance, then simply forwards the packet to the indicated egress channel. 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.

3.5 Advantages and Challenges

Below are some of the advantages of our proposed approach, compared to the current Internet routing architecture:

- *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 costs. In particular, our scheme adds additional latency to obtain paths before a first data 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 thesis is to give evidence that a well-engineered system can keep these delays within acceptable range.

Chapter 4

Related Work

In this chapter we discuss several important research directions related to our work. First, we briefly describe centralized routing approaches in the current Internet architecture in Section 4.1. In Section 4.2, we provide a summary of several clean-slate source routing architecture proposals. Section 4.3 describes several solutions to the multi-constrained path computation problem. Finally, in Section 4.4, we describe a protocol to advertise performance (i.e., QoS) information in the Internet.

4.1 Centralized and Multi-path Routing Approaches

In this section, we first discuss the IP source routing mechanisms which are generally not available in the current Internet architecture due to security issues and violation of ISP policies. Then, we discuss centralized routing approaches within provider networks and within bridged local area networks. An application-level centralized routing approach is described in Section 4.1.4. We conclude this section with the description of an on-going effort to support reliable multi-path communications in the Internet.

4.1.1 Source Routing in the Internet Protocols

The Internet protocol specification defines two IPv4 options called *"loose source and record route (LSRR)"* and *"strict source and record route (SSRR)"* [42]. Both the

LSRR option and the SSRR option provide a mechanism for the sources to insert paths (i.e., source routes consisting of a sequence of IPv4 addresses) into packet headers. SSRR differs from LSRR in that successive IPv4 addresses in the SSRR option must be directly adjacent neighbors, whereas in LSRR they may be separated in the router-level topology by several hops. The headers of both options contain a variable-length path field to store the path, and a 1-byte pointer field which indicates the next hop to be traversed in the source route.

The destination address of LSRR and SSRR packets is initially set to the address of the first hop node in the source route by the sender. In the case of a packet with LSRR option, the network simply forwards the packet using hop-by-hop routing until it reaches its destination. If the destination is not the final hop on the LSRR path, the intermediate (destination) node replaces the IPv4 destination address of the packet with the address of the next hop in the source route and increments the pointer field in the header. In the case of a packet with SSRR option, each hop along the path replaces the destination with the next hop address and increments the pointer.

The processing of IPv4 packets with source routing option at the intermediate destinations also involves recording the reverse path in the packets for the receivers to use in their return traffic back to the sender. The recorded reverse path replaces the route in the path field placed by the sender. In particular, each intermediate destination on the path replaces the address of its ingress interface in the path field with the address of its egress interface, where the packet is to be forwarded. The resulting recorded path is in the reverse order of the path from the destination back to the source. Similar to LSRR option in IPv4, the Internet protocol version 6 (IPv6) specification defines an optional routing extension header (type 0) that can be used by an IPv6 host to insert a sequence of IPv6 addresses [25]. Unfortunately, both the IPv6 and the IPv4 source routing options are *deprecated* [7, 67]. As a consequence, most network routers and hosts (by default) drop IP packets that contain source

routing options.

One of the main reasons for the deprecation of source routing is the ability of the senders to specify arbitrary paths that violate the policies of ISPs, such as paths that bypass firewalls or other middleboxes that ISPs strategically place in their networks to perform various checks on the traffic [37]. As opposed to IP source routing, in our source routing approach, packets can only contain ingress and egress channels of the domains, and the domains control the forwarding paths across their infrastructure. Also, the providers use policy-enforcement mechanisms to only allow packets that contain policy-compliant inter-domain paths.

Another important reason for the deprecation of IP source routing is the potential for bandwidth exhaustion attacks. In such attacks, the sender specifies a route containing loops, for example one that is routed between two victim nodes repeatedly [7]. In our approach, a path service provides simple paths to end-systems, and additional mechanisms can be used to prevent end-systems from modifying (e.g., inserting channels) those paths provided by the service. In particular, the in-band policy enforcement mechanism (described briefly in Section 3.4.2) can be modified to generate proof-of-compliance tokens that are cryptographically bound to the entire path. Such tokens are verified with computations on each channel that form the path, and modifications to the path would make the token invalid with very high probability. However, using tokens that are tied to the entire path also makes the verification of tokens more expensive.

4.1.2 Source Route Bridging

The 802.2 standard defines source route bridging that is used in token-ring networks [5]. A routing information field (RIF), is used to place source routes consisting of a series of bridges and Local Area Networks (LANs) in the frames.

End-systems discover source routes to destinations (within the LANs inter-connected

by source routing bridges) by sending a special type of broadcast frame. A broadcast frame records the path it traverses as it travels through bridges. In particular, each bridge receiving a broadcast frame records its bridge number (i.e., unique to each bridge) and drops arriving broadcast frames that already has its own bridge number to prevent loops. Each broadcast frame also has a maximum hop count, which is decremented by each bridge receiving the frame. A frame is dropped if the hop count reaches zero.

The destination uses the recorded source routes in each of the received broadcast frames to send unicast frames along the reversed paths to notify the sender of the available paths to the destination. Therefore, for each broadcast frame sent, the sender discovers a set of paths to the destination. The sender selects one of the discovered paths and places the path in the outgoing frame to communicate with the destination. The discovery of paths using broadcast messages is effective within small bridged networks, but the approach is not scalable for the discovery of paths at the inter-domain level.

4.1.3 Software-Defined Networking (SDN)

Software-Defined Networking (SDN) [31] is a centralized routing approach that separates the routing functionality from routers and performs it in a centralized *controller*. SDN achieves this separation by decoupling *the "control plane" from the "data (forwarding) plane"*. The control plane is the part of the router architecture that performs the routing functionality (i.e., participating in routing protocols to make routing decisions). The data plane is the part of the router architecture that performs the forwarding function based on the routing decisions of the control plane. The controller processes all the packets that belong to the control plane, makes routing decisions, and pushes the decisions onto the network elements for the forwarding plane.

SDN requires new protocols for the network switches to communicate with the

controller to forward control plane messages and obtain forwarding instructions. One such protocol is OpenFlow [58]. OpenFlow provides an interface for remotely controlling the forwarding tables of network devices to add, remove or modify packet matching rules and actions. The use of SDN in the current Internet is largely limited to single-provider networks such as data center networks. An exception is Google's deployment of SDN in a wide-area network which connects multiple Google data centers [45]. In addition, recent research has proposed the use of SDN-capable switches at the Internet Exchange Points (IXPs) where multiple providers peer with each other [39]. The deployment of an SDN-capable switch at an IXP would enable application-specific peering policies. For example, two domains can establish peering to carry only the packets belonging to certain applications by inserting appropriate forwarding rules in the switch.

Our path service approach is compatible with the use of SDN to forward packets across the domains. In particular, the path service, upon selecting a path, can inform an SDN controller for each transit domain on the inter-domain path to install forwarding state in its domain. Once the state is installed in a transit domain, the packets arriving on the specific ingress channel (that match a specific pattern such as source and destination addresses in the packet) can be forwarded through the network to the specified egress channel. However, the deployment of SDN is currently not common in transit providers and is more common in datacenter networks.

4.1.4 Application-level Routing

Routing overlays have emerged as a result of the users' desire to change the default (BGP) path of data through the network. Routing overlays have been proposed for defining paths based on application-specific criteria (e.g., lower latency) and for improving resiliency through multi-path routing [9, 8, 90, 16, 73].

In routing overlays, a set of end-systems participate in routing and forwarding

of the packets. Using a routing overlay with participating end-systems, applications concatenate multiple default paths through multiple end-systems to reach their destination. Even though the routing between end-systems still uses the default paths chosen by the BGP protocol, overlay routing has been shown to improve the quality and resilience of communications between end-systems [9].

On the other hand, overlay traffic tends to violate the routing policies of the ISPs by circumventing their routes determined according to cost and operational efficiency criteria [56]. By circumventing the routes determined by the ISPS, overlay traffic can cause load imbalances in the ISP networks that would otherwise have been avoided. More importantly, overlay traffic routed over multiple intermediate end-systems often consumes the resources of ISPs that do not obtain any compensation from the sources or the destinations of the traffic. This leads to an on-going "tussle" [22] between ISPs and users in the Internet. As a result, ISPs deploy packet filtering mechanisms (i.e., deep packet inspection) to drop overlay traffic.

There are also other technical problems with overlays. For example, overlays are inefficient because they involve routing through end systems (which should not be used to transit packets). In particular, the in/out problem of traffic going "in" to an end system just to be turned around and come "out" of the end system is known to be a source of congestion in overlays. In summary, the routing overlays are not a viable long-term solution to obtain paths with better end-to-end quality and resilience because of policy and technical problems.

4.1.5 Multi-Path Transmission Control Protocol (MPTCP)

It is increasingly common for end-systems to be multi-homed in the Internet. For example, mobile systems often have simultaneous access to a Wi-Fi channel and to a cellular data network. Such systems can use multiple simultaneous paths to reach destinations by sending the traffic through all the available interfaces. *Multi-path*

transmission control protocol (MPTCP) [33] is an on-going work at the IETF whose aim is to allow reliable transport-level connections through multiple simultaneous paths. MPTCP provides the same interface as TCP to applications by abstracting the individual resources on the paths into an aggregate pool of resources [81]. MPTCP sends the data on multiple paths as individual flows (one flow per path) of traffic and controls the rate of each flow.

An important challenge with MPTCP is stability. In particular, MPTCP needs to be responsive to congestion and move traffic from flows with congested paths to flows with less congested paths in order to take advantage of the pool of available bandwidth across the available paths. However, an over-responsive traffic distribution mechanism can lead to oscillations in the rates of the individual flows. MPTCP uses the theoretical foundation of the work by Kelly et al. [47] to stabilize the rate of change in the flows by coupling the congestion control with the load-distribution mechanism that moves traffic across the flows [82]. MPTCP moves the traffic from a congested path to uncongested paths at a rate permissible by the additive increase multiplicative decrease (AIMD) of the congestion control mechanism. In particular, the change in the rate of a flow is only constrained by the RTT of the flow's path and not the RTTs of the other flows' paths.

The performance of MPTCP can suffer with the use of overlapping paths when overlapping links have congestion. Because paths are selected by BGP in the current Internet architecture, MPTCP is not guaranteed to obtain disjoint paths when sending traffic through multiple interfaces. Our path service, on the other hand, can return disjoint paths that can improve the performance of MPTCP. Also, the applications can replace paths with new ones (obtained from the path service) when the current set of paths used for the MPTCP connection are not satisfactory.

4.2 Clean-slate Source Routing Approaches

In this section, we start by briefly describing a policy enforcement mechanism that we use in our source routing implementation in Chapter 7. Then, we will describe several clean-slate source routing architectures.

4.2.1 Platypus

Raghavan et al. [66] identify the lack of policy enforcement mechanism as the main barrier towards the deployment of source routing and propose a data plane (i.e., in-band) policy enforcement mechanism for provider-level loose source routing architectures. In the data plane mechanism, policy-compliant packets carry valid proof-of-compliance (capability) tokens that are verified during forwarding of the packets.

An end-system computes a proof-of-compliance token using a one-way function—keyed hash-based authentication code (HMAC)—on various invariant fields of the packet such as the path information. The shared secret (i.e., key) used in the HMAC computation for a token is securely transmitted from either the waypoint owner or a "delegated entity". Platypus provides an efficient delegation mechanism to distribute shared secrets from the waypoints to customers and each customer in turn can further delegate to its own customers and so on.

Token delegations and verifications involve HMAC computations, but require no state to be maintained by the network. We do not provide further details of Platypus in this thesis, as our focus is on routing. The source routing system, which we describe in Chapter 7, uses an in-band policy enforcement mechanism, very similar to Platypus. However, our routing scheme is compatible with other (stateful) policy-enforcement mechanisms including SDN.

4.2.2 Routing as a Service

Lakshminarayanan et al. [55] propose centralized computation of paths with guaranteed quality-of-service (QoS). A centralized *Routing Service Provider (RSP)* acts as a broker between providers that are willing to sell quality-of-service routing and users who desire customized routes. Providers sell virtual links, which are paths across the providers (similar to relay paths), to the RSP. The RSP stitches together virtual links to form inter-domain paths and sells the end-to-end paths to users. ISPs provide the RSPs with a service-level agreement (SLA), which defines the QoS guarantees for each virtual link sold by the providers. However the RSP is designed to be used by a small number of users for only customized routes that demand guaranteed QoS.

4.2.3 NIRA: New Inter-domain Routing Architecture

In *NIRA* [88], end-systems discover paths to a set of well-connected providers that form the "Internet core region". The paths to the core region are discovered through the *Topology Information Propagation Protocol (TIPP)* that distributes "path information" downwards in a customer-provider hierarchy. The core region, being the root of the hierarchy, initiates the TIPP protocol by sending messages that eventually reach the edge (i.e., access) domains. As the protocol messages propagate downwards, they accumulate the paths to reach the core region.

An example hierarchy of domains with a core region consisting of the domains $B1$, $B2$, $B3$, and $B4$ is shown in Figure 4.1. Because the discovery protocol messages propagate only downwards in the customer-provider hierarchy, all the discovered paths from access domains to the core strictly follow customer-to-provider (inter-domain) links that are shown with lines with arrows pointing to the provider domain in the Figure 4.1. In NIRA, an end-to-end path between two hosts contains an up-path and a down-path portion. The up-paths consist of a sequence of customer-to-provider links and the down-paths consist of a sequence of provider-to-customer links.

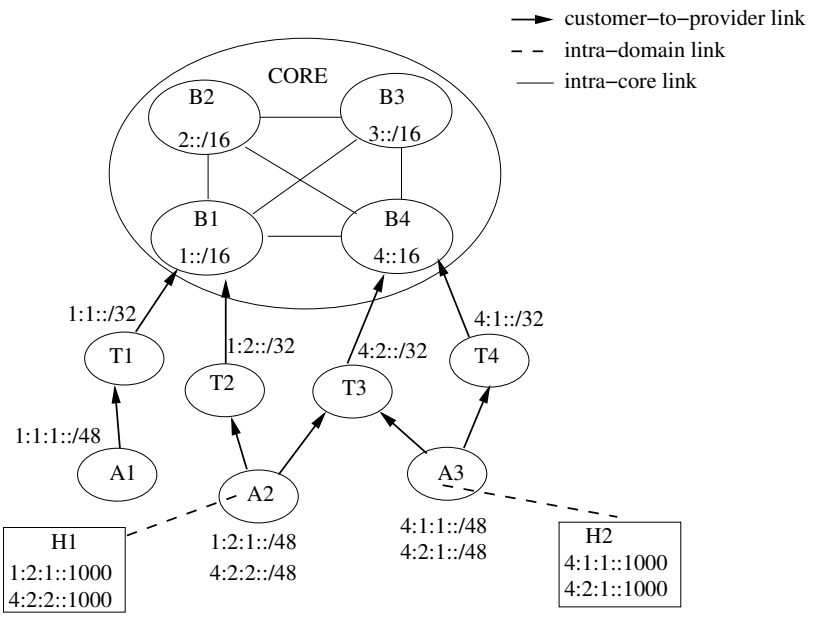

Figure 4.1: A customer-provider hierarchy of domains with a core component that contains B1, B2, B3, and B4.

NIRA architecture encodes both up-paths and down-paths in IPv6 addresses that are 128 bits long, where the first 96 bits are used as an inter-domain address prefix that is hierarchically allocated from a provider to a customer, and the remaining 32 bits are used as an intra-domain address that identifies a network location within a domain. In Figure 4.1, the top-level providers are shown with their globally unique IPv6 prefixes that are obtained from a Regional Internet Registry. Using the TIPP protocol messages, $B4$ allocates portions of its prefix 4::/16 to its customers $T3$ and $T4$. $T3$ and $T4$ obtain 4:2::/32, 4:1::/32, respectively. $T3$ then allocates prefix 4:2:1::/48, and similarly, $T4$ allocates 4:1:1::/48 to the access domain $A3$. The end-system $H1$ in $A2$ is assigned 1:2:1::1000 and 4:2:2:1000 from the two prefixes assigned to $A2$, while the end-system $H2$ in $A3$ is assigned 4:1:1::1000 and 4:2:1::1000.

In NIRA, packets contain a source address corresponding to an up-path and a destination address corresponding to a down-path. Packets are first forwarded along the sequence of domains that allocated the source address, and then forwarded along

the sequence of domains that allocated the destination address. By selecting a source and a destination address, end-systems determine the sequence of providers along the up-paths and down-paths that the packets are to follow.

As an example, consider the scenario where $H1$ selects the source address 1:2:1:1000 and places the destination address 4:2:1::1000 in its packets destined to $H2$. This means that the packets are to follow the up-path $A2 \rightarrow T2 \rightarrow B1$ and then the down-path $B4 \rightarrow T3 \rightarrow A3$. Because the root of the up-path ($B1$) and the down-path ($B4$) are different providers, there needs to be a mechanism to route the packets between the providers in the core region. In NIRA, routing within the core is performed using a hop-by-hop mechanism and the hosts have no control over the portion of the paths that move packets across the core. Even though the inter-connections between the domains within the core region are shown in the Figure 4.1, in reality, the domains or the end-systems outside the core have no visibility into the core region. Also, the core region is not necessarily a fully-connected network of domains as depicted in Figure 4.1.

An up-path and a down-path may intersect before reaching the core. In that case, the transition in the forwarding path from the up-path to down-path happens at the intersecting provider. As an example, if $H1$ selects the source address 4:2:2::1000 to reach $H2$ using 4:2:1::1000, then the packets are forwarded along the path: $A2 \rightarrow T3 \rightarrow A3$.

In order to compose an entire path to a destination, the sender host needs to discover the down-paths (i.e., IPv6 addresses) of a destination host. In NIRA, a lookup service provides down-paths of destination hosts. Hosts, who are willing to be reachable, register their addresses to the lookup service. To send a packet to a destination host, a sender first contacts the lookup service and obtains the set of addresses of the destination host. Then, the sender chooses a source address and a destination address (from the addresses obtained from the lookup service) to form an

end-to-end path.

The TIPP protocol has a separate component to distribute performance information in addition to address allocation messages that enables end-hosts to discover their connectivity to the core. However, the performance information only includes the performance on the inter-domain links (outside the core region) and does not include the performance of the forwarding paths across the providers in the up-paths and the down-paths. Because the discovery protocol messages propagate only downwards in the customer-provider hierarchy, all (discoverable) inter-domain paths in NIRA have the "valley-free" property [35]. The valley-free property means that after traversing a down-path, the inter-domain paths do not traverse an up-path again. Our source routing approach does not impose such restrictions on the structure of the paths. In addition, our scheme does not require the existence of a core region in which end-systems have no control over the paths. More specifically, our scheme enables domain-level path selection using the connectivity of all the domains and does not require the abstraction of a region of domains.

4.2.4 SCION

Scalability, Control and Isolation On Next-Generation Networks (SCION) is a source routing architecture design [89]. In SCION, a set of contigious ASs form a *trust domain (TD)*, which is an abstraction to isolate ASs that have a "trust" relationship with one another from other ASs. ASs in a TD only exchange control plane messages with each other. Within a TD, the ASs form a customer-provider hierarchy with a trusted core component consisting of fully-connected ASs. Routing within a TD is very similar to NIRA and involves discovering up-paths and down-paths. The up-paths and down-paths of an AS A only consist of other ASs that are in the same TD with A.

The inter-TD routing takes place using pre-negotiated and configured paths, as

the SCION architecture assumes at most 100s of TDs to emerge in the Internet. An inter-TD route first follows an up-path within the source TD followed by a pre-configured inter-TD path and a down-path within the destination TD. The main difference between NIRA and SCION is the concept of trust domains and addressing. In SCION, the end-systems have flat identifiers and path information is not embedded in hierarchical addresses as in NIRA. The SCION architecture achieves scalability through TD abstraction. In our approach, we do not rely on TD abstraction to achieve scalability.

4.2.5 Nimrod

Nimrod [18] is a routing architecture that represents an inter-network topology as a hierarchy of *clusters*. Clustering begins by aggregating network elements (i.e., routers and switches) and can be applied recursively to form cluster of clusters. The only restriction with clustering is that the elements within a cluster are connected by a path that is entirely within the cluster. A universal cluster contains all other clusters in the inter-network at the topmost level of the cluster hierarchy. A topology map of a cluster representing an inter-network at multiple levels of abstraction is shown in Figure 4.2.

Routers and hosts discover the topology through a map-distribution mechanism. Maps express the connectivity and services between entities in different regions of the Internet. Maps can represent a physical network at different levels of details. In Figure 4.2, Node B reveals more details of its topology than A. Also, node D reveals the routers within its network, while node E does not reveal its inner details. By repeated clustering and abstracting the connectivity and services provided in the maps, the Nimrod architecture reduces the amount of information that is visible to the routing process, thus achieving scalability.

The maps are distributed using a link-state protocol that allows providers to

restrict the distribution (i.e., which neighbors to share) of the maps. In addition to the link-state protocol, the Nimrod architecture provides mechanisms for entities to request detailed maps of a region by sending queries.

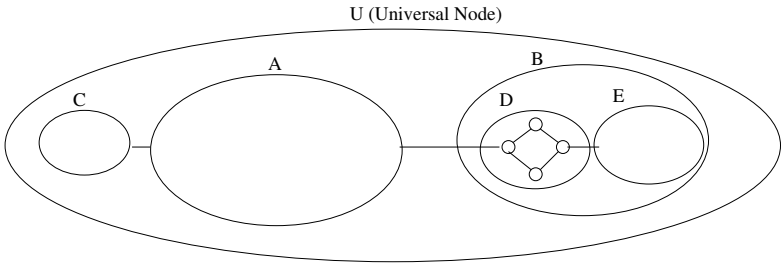

Figure 4.2: The Nimrod architecture represents the Internet topology as clusters of clusters of routers and switches.

Locations of end-systems and nodes are represented by *locators*, which are sequences of identifiers that describe the levels of clustering the entities. For example in Figure 4.2, the locator of node D is U:B:D (the universal cluster U can be omitted). Maps represent entities by their locators. A lookup mechanism maps endpoint identifiers to locators.

The architecture supports routes with different levels of details: strict and loose source routing as well as hop-by-hop routing are supported for end-systems with different computational power. In particular, end-systems with sufficient computational power can request detailed topology maps and compute strict source routes, while other (less powerful) end-systems can use less detailed maps to compute loose source routes or simply place the destination address in the packets to use hop-by-hop routing. In Nimrod, forwarding information in routers is established on-demand rather than pre-computed as in the current Internet routing architecture. Therefore, forwarding of packets by the routers may be delayed if the routing information (i.e., topology map) of the router is insufficient to compute paths. In that case, the router requests detailed routing information from the neighbors. Routers cache forwarding information to reduce the amount of resources consumed and delay incurred in

obtaining the information in the future.

Nimrod architecture does not provide the necessary mechanisms to support transit policies for providers and does not include scalable mechanisms to distribute topology maps and to compute end-to-end paths. As a result, the only instance of Nimrod is an intra-domain routing protocol for ATM networks called the Private Network-to-Network Interface [11] protocol.

4.2.6 Pathlet

The *Pathlet* [36] architecture uses MPLS-like label switching as the main mechanism to forward packets. In the Pathlet architecture, the packets contain a sequence of labels as their source routes. The nodes in the architecture advertise sequences of labels—corresponding to segments of paths—which are called *pathlets*. What distinguishes the Pathlet architecture from other label-switching architectures is the concept of *virtual nodes (vnodes)*. A vnode is an abstraction that can represent an entire domain, a group of routers, or an interface to a router. In essence, pathlet routing is label-switching over a virtual topology whose nodes are vnodes and whose directed edges are pathlets. Vnodes are assigned globally unique identifiers. Each domain in the Pathlet architecture must generate at least one vnode and advertise a pathlet to reach its vnode in order to receive traffic.

An important property of the Pathlet architecture is that it supports a variety of transit routing policies. One extreme case is the local transit (LT) policy. A domain, say N, uses the LT policy if N's willingness to relay packets along some route depends only on the portion of the route that crosses N. In order to use the LT policy, N designates a vnode for each of its ingress and egress interfaces. If N is willing to transit packets between its neighbors A and B, then N constructs and advertises a pathlet from its vnode corresponding to its ingress interface from A to its vnode corresponding to its egress interface to B. A domain with n neighbors can advertise

up to $n \times (n-1)$ pathlets to provide transit service between all its ingress and egress links.

The other extreme case is the BGP routing policies, where a domain N's willingness to relay packets along some route depends on the portion of the route from N to the destination of the route. The Pathlet architecture can emulate BGP policies by using a path-vector protocol (similar to BGP) to distribute pathlet advertisements. In this case, each access domain creates a vnode for each of its prefixes and advertises pathlets to reach its vnodes to their neighboring domains. As the pathlet advertisements propagate in the network, each domain constructs a multi-hop (inter-domain) pathlet to each destination prefix along its most preferred domain-level path. Multi-hop pathlets are advertised with a single label along with the AS-PATH corresponding to the pathlet. A packet containing a single label corresponding to a multi-hop inter-domain path is forwarded as follows: each domain receiving the packet along the path swaps the (topmost) label in the source route with a new label that is associated with the forwarding path to reach the next hop domain.

The downside of using BGP-style policies is that a domain needs to maintain pathlets for each destination prefix in order to send the packet along the desired paths. In the case of LT, the pathlets are advertised using a link-state protocol and can be stitched together (i.e., sequence of labels) in exponentially many ways to form larger end-to-end inter-domain pathlets, similar to our source routing approach. In our scheme, a relay advertisement corresponds to one or more pathlets between an ingress channel and an egress channels.

The novelty of the Pathlet architecture is its use of vnodes. The architecture represents the policy-compliant data plane with a set of pathlets that inter-connect vnodes. Being able to represent routing policies with topology maps consisting of vnodes, the architecture can emulate a variety of routing policies. The Pathlet architecture does not specify how the end-to-end paths are computed in the case of LT

policies. Our path service can be used to complement the Pathlet architecture with the LT policies to compute inter-domain paths for the end-systems.

4.2.7 ICING

ICING [61] is a loose source routing proposal which focuses on policy-compliance of source-routed packets. Different from existing proposals, ICING uses *path-based policies* where the domains' willingness to forward packets depends on the entire inter-domain path (i.e., sequence of domains) of the packets. In particular, a sender, before it can send traffic along an inter-domain path, must obtain *consent* from each domain along the path to use the path. A possible motivation for domains to enforce policies that concern the entire inter-domain paths is security. For example, if a domain, say S, does not trust another (not necessarily a neighbor) domain, say M (for malicious), S can refuse to provide consent for paths that contain M. By refusing to provide consent for paths with M, S can prevent packets that are forwarded through M to reach its network and also prevent packets being sent to M through its network.

However, the path consent mechanism is not sufficient to enforce path-based policies without additional mechanisms to verify that packets actually take the paths for which they have consent for. In order to address that problem, ICING provides mechanisms for each provider N_i to verify that an arriving packet with path $P = N_0, N_1, N_2, ...$ actually followed the path $N_0, N_1, ...N_{i-1}$ on its way to N_i. ICING uses in-band mechanisms to provide consents and to verify that the packets actually follow their paths.

Because ICING is mostly concerned with the challenges of forwarding, it is orthogonal to our path service that computes and selects paths. The path service design, which we explain in this thesis, assumes local transit policies. However, our path service can be extended to work with domains that use path-based policies. For example, the path service can collect path-based policies from the providers (possibly in

the form of regular expressions) to compute and select paths that are compliant with the path-based policies of the domains. Because such policies are expected to change slowly, the path service can compute policy-compliant paths in advance similar to computing paths with local transit policies. We leave it to future work to extend our path service to support path-based policies.

4.2.8 A Network Path Advising Service for the Internet

Wu et al. propose *a network path advising service (NPAS)* that monitors the performance of a set of links and advises paths for the end-systems [86, 85]. End-systems obtain performance measurements from NPAS to select paths among a small subset of possible paths to a destination. The goal of NPAS is to monitor the minimum number of links that cover the majority of the paths that the end-systems use in the Internet.

NPAS takes advantage of locality of reference in the set of destinations contacted by the end-systems to achieve a high coverage percentage [86]. In particular, by covering the majority of the links that are close to popular destination domains, NPAS achieves a high coverage of the paths. The results of the work by Wu et al. is relevant to the design of the path service, because the path service also collects performance measurements from the network. The path service can use the strategy of selectively collecting performance information from the popular regions of the inter-network to reduce the bandwidth overhead of collecting measurements.

4.3 Multi-constrained Path Problem

The multi-constrained path problem (MCP) [74, 20] is the problem of computing a feasible end-to-end path from a given source to a given destination that satisfies a given set of constraints. A constraint is a desired upper or lower bound value on a specific link quality measure (routing metric) such as bandwidth, delay and so on.

The MCP problem is formally stated as: Given a graph representation G where each link in the graph is associated with a vector of m (instantaneous) measurements in various routing metrics (e.g., delay), a source node S, a destination node D and a vector of constraints C, find a single path from S to D that satisfies the constraints.

Routing metrics are classified as either *additive* or *max-min* based on their composition. For example, routing metrics such as number-of-hops, propagation delay, jitter are additive metrics, because the path metric is defined as the sum of the metrics of the links in the path. Available bandwidth is an example of a max-min metric, because the available bandwidth of the path is the minimum over the available bandwidths of the links in the path. The problem of finding a feasible path in a network that satisfies multiple constraints on multiple additive routing metrics has been proved to be NP-hard, and the proof involves reducing the problem to the well-known NP-hard Partition problem [79]. The main reason for the complexity is the exponential growth of *incomparable paths* that an exact solution to the MCP problem must consider during the computation. Paths P_1 and P_2 (with the same source and the destination) are incomparable if both of them satisfy the constraints C and P_1 is better than P_2 in at least one metric, and at the same time P_2 is better than P_1 in at least one other metric. P_1 is said to be *dominated* by the path P_2 if P_2 is better than P_1 in all the routing metrics. When a path is dominated by another (with the same source and destination), it can be removed from the set of feasible paths that are considered during the computation of a feasible path.

Samcra [74] is an exact solution to the MCP problem, which uses a nonlinear path length function: $len = max_{1 \leq i \leq m} \frac{w_i(P)}{C_i}$ where $w_i(P)$ is an additive attribute of a path P and C_i is the constraint on the additive attribute. An important property of a nonlinear path length is that the subsections of shortest paths in multiple dimensions (i.e., attributes) are not necessarily shortest paths. This suggests that one should consider the k-shortest paths in the computation as opposed to the shortest

paths. The Samcra algorithm is essentially Dijkstra's algorithm [28], but it does not terminate when the destination is reached. Instead, it continues until k paths are found to the destination. If k is not limited, the k-shortest path algorithm returns all the possible paths between the source and the destination. Samcra does not have a pre-determined limit on the value of k.

As the Samcra algorithm extends the k-shortest paths from the source node towards the destination, the number of incomparable paths can grow exponentially and in the worst case scenario, the algorithm considers all possible paths between the source and the destination. Samcra uses various techniques to eliminate paths as they are generated. One example is the elimination of dominated paths.

Next, we discuss several heuristic solutions that reduce the search space to obtain a polynomial-time algorithm.

4.3.1 Heuristic Approaches to Multi-constrained Routing

One heuristic approach is to compute the length of paths as the linear combination of the individual routing metrics to avoid incomparability of paths [44]. Such an approach reduces the m dimensional problem, where m is the number of routing metrics, to a single dimensional problem. After reducing the problem to a single dimension, any shortest-path algorithm (e.g., Dijkstra) can be used to compute the shortest path. However, the shortest path does not necessarily satisfy the individual constraints when the path length is a linear combination of the routing metrics.

Korkmaz et al. proposed a randomized heuristic algorithm that consists of a pre-computation step followed by a randomized breadth-first search [50]. The algorithm starts by computing the shortest path (using Dijkstra's algorithm) from each node u to the destination node D with respect to a linear combination of the link weight components (called length) on the lines of Jaffe's approach [44]. After this pre-computation step, the algorithm starts from the source node S and discovers

neighboring nodes that are estimated as "feasible". A node t is feasible if the length of the pre-computed shortest path from t to D plus the length of the discovered path from S to t is less than the linear length value of the constraints. Once all the feasible neighbors of S are discovered and placed in a queue Q, the algorithm randomly picks a node n from Q and discovers n's neighbors similarly. The algorithm terminates when D is discovered. However, the path to D may not be a feasible path as the prediction of feasibility on pre-computed linear lengths can direct the algorithm towards infeasible paths.

Several heuristic methods take actions to limit the number of incomplete (i.e., a path from source to an intermediate node) paths considered by the algorithm so that the complexity becomes polynomial. One such algorithm by De Neve et al. [23] prevents the exponential growth of paths by limiting the number of (incomparable) incomplete paths to intermediate nodes that are computed.

To summarize this section, the main drawback of the exact approach to the MCP problem is that in the worst case, all possible paths are considered. Heuristic solutions cut down the number of paths, but they do not guarantee finding a satisfactory path. Our approach to computing paths also uses a heuristic approach that limits the number of paths considered to a subset of all possible paths. However, our approach also makes use of pre-computations and parallel computations by considering the time-scales of change in the routing information.

4.4 Advertising QoS Information

In general, the QoS routing algorithms assume that the most recent performance measurements from the networks are available for computation. The current BGP protocol does not include performance measurements as part of the advertisements.

In a work by Xiao et al. [87], histograms of measurements are advertised periodically in route advertisements of ASs in a modified BGP protocol. As the route

advertisements are propagated between ASs, the histograms of measurements of different ASs are joined to obtain aggregate measurements of paths. For latency, the histograms are joined using convolutions. However, this approach has the same drawbacks as the BGP in terms of limiting the number of paths known to ASs since each AS advertises one path to each destination. In our approach, we also use histograms to represent a set of measurements.

Chapter 5

The Design of a Path Service

In this chapter, we discuss several important design considerations needed to achieve a scalable path service, including caching and the time-scales at which routing information changes. We then describe the system architecture in Section 5.3.

An important consideration in the design of the path service is the *transferability* of paths between end-systems in the same access domain. In particular, inter-domain paths from the access domain S to all other domains are usable by many of, but only, the end-systems in S. Reuse of recently computed paths by end-systems in the same access domain can be achieved by storing the paths in a local *path cache* in each access domain. A local path cache can improve the scalability of the path service if there is significant locality of reference in the set of destinations contacted by the end-systems. We will discuss the path cache in Section 5.2.

From a system design perspective, our path service and existing web search systems have similar properties. Both systems deal with dynamic information, even though the time-scales of change in the dynamic information used by the two systems can be different[1]. In the case of web search systems, the dynamic information is the content of the web pages and in case of the path service, it is the routing information. Also, both systems must respond to queries in a timely manner. Web search

[1]In general web page information changes at a substantially slower rate than routing information; however, web search systems for social networking sites must deal with fast-changing content because of the high rate of updates in such sites.

engines use parallelism to speed up the processing of on-demand computations and also perform expensive computations, such as page rank [64], offline in advance. The on-demand computations are parallelized by dividing the web content into smaller partitions. In particular, a set of workers computes results on the individual partitions in parallel, followed by a relatively inexpensive merging step, which produces the final results [13]. We follow a similar strategy of using expensive pre-computations and parallelizing the on-demand query computations.

In general, pre-computations are useful only if the rate of change in the input is smaller than the rate at which the pre-computations can be repeated. Therefore, an important consideration in the design of the path service is the time-scales of change in the routing information.

5.1 Time-Scales of Routing Information Changes

We consider the time-scales of change in the following routing information:

- The *set of possible paths* is defined by the topology, namely the set of access and transit domains and the channels that interconnect them. This set changes *slowly*: only when new providers and channels come into existence or existing ones go out of service permanently.

- The *medium-to-long-term performance information* of a relay, or of a path, collected and averaged/analyzed over timescales of seconds to minutes—e.g., a histogram of measurements observed over the past minute. It is expected that such information is a reasonably good predictor of the near future.

- *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 [53].

Because the connectivity of the domains in the Internet changes slowly, *it is reasonable to perform an expensive computation to construct a long-lived database of possible paths in advance, and thereafter update it on a relatively slow timescale (hours to days) as the underlying graph changes.* The constructing of paths can include the computation of slow-changing (i.e., load-independent) attributes of the paths such as bandwidth capacity and propagation delay. The path service can update the performance information of the pre-computed paths in an on-demand fashion.

Generating a subset of all possible paths from an access domain to all other domains has an exponential cost of $O(T^{L-2})$, where T is the number of transit domains and L is the length (in terms of number of domains, including source and destination) of the longest path allowed between any two access domains. This cost approximation assumes the transit domains are fully-connected, thus there are T alternatives for each hop along the path. A better approximate upper-bound on the cost is $O(D^{L-2})$, where D is the highest degree among the degrees of the transit providers. Because of the computational cost involved in constructing all possible paths, we use a heuristic approach to rule out paths that are much longer (more than 2 hops) than the shortest paths.

5.2 Path Cache

In order to get an idea of how effective a path caching mechanism would be if it were to be deployed at an access provider, we performed a set of experiments using a flow-level trace of the outgoing traffic collected at the edge routers of a campus network for the duration of one day.

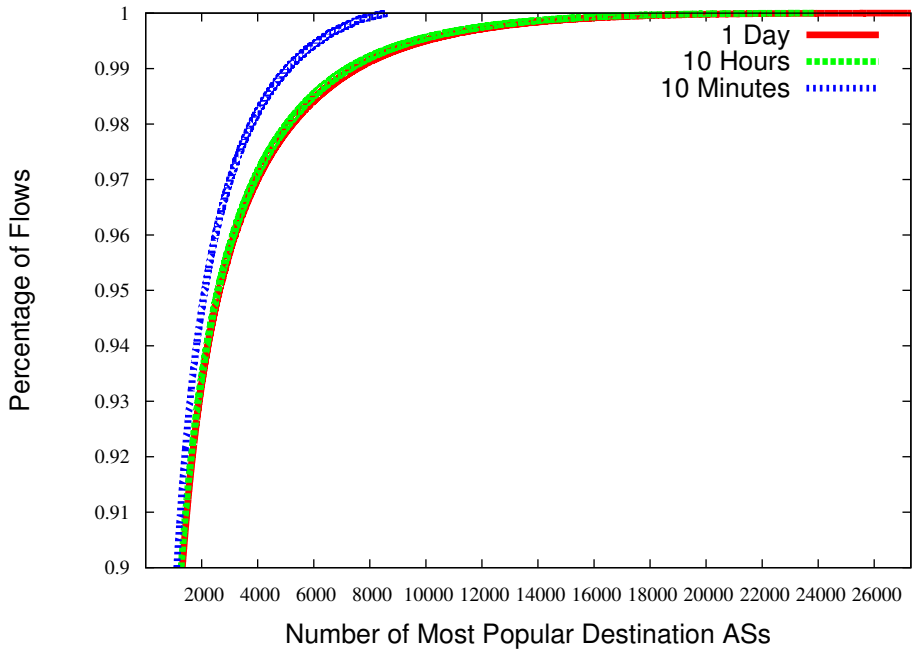

Figure 5.1: CDF of flows to each of the most popular destination ASs for the time-scales of 10 minutes, 10 hours, and 1 day.

These measurements can give us an idea about the effectiveness of caching, because each flow with destination domain D leaving the campus network corresponds to a request to the path service for paths to domain D. We make the assumption that the paths returned from the path service to end-systems in the campus network are cached at some location close to the end-systems.

Our initial results indicate that there is a significant locality of reference in the set of destination ASs contacted by flows originating from the campus network. The cumulative distributions of flows going to each of the most popular destinations—that is, the destinations that attract the largest number of flows in the traces—is shown in Figure 5.1 for the time-scales of 10 minutes, 10 hours and 1 day. As can be seen from the figure, the majority (over 90%) of flows, in all the time-scales, are destined to less than 2000 ASs. The flow distribution for the 10 minute data in Figure 5.1 corresponds to the first 10 minutes during the peak hour (i.e., most flows leaving the campus) traffic. The peak hour happened between 11 am and 12 noon on a

business day. We used the peak hour traffic because the effectiveness of caching is most important under the most intense load. Also, we used the 10 hour data in the plot corresponding to the busiest 10 hour period during the business hours of 8 am to 6 pm.

In our preliminary analysis, we did not classify flows according to classes of traffic. Instead, we simply assumed that all flows used the same paths for the same destination domain. Our analysis without traffic classes suggests that a path cache with a capacity of few thousand entries would cover around 98 percent (i.e., miss rates around 2 percent) of all path requests at all time-scales.

One concern with path caching is the lifetime of the paths. With the arrival of new routing information through relay advertisements, a cached path may become *stale* if it is not computed with the most recently obtained routing information. We explore the effectiveness of caching paths with dynamic attributes in the next chapter.

5.3 Path Service Architecture

This section presents the conceptual design of the path service described in the foregoing sections. The path service design takes advantage of the time-scales of change in the routing information. In particular, the service distinguishes between *fast-changing* and *slow-changing* routing information. The slow-changing routing information comprises the connectivity and load-independent information. Examples of load-independent information include bandwidth capacity and propagation delay. On the other hand, the fast fast-changing routing information comprises load-dependent performance information such as latency and available bandwidth.

The path service performs an expensive computation to construct paths and their slow-changing information. We refer to this operation as *slow-join* since the constructing of paths require an iterated operation that is the equivalent of a database "join" operation on slow-changing relay information (an example is given below).

Computing the performance information of paths involves join operations using fast-changing relay information. Similarly, we refer to the join operation performed using fast-changing information as *fast-join*.

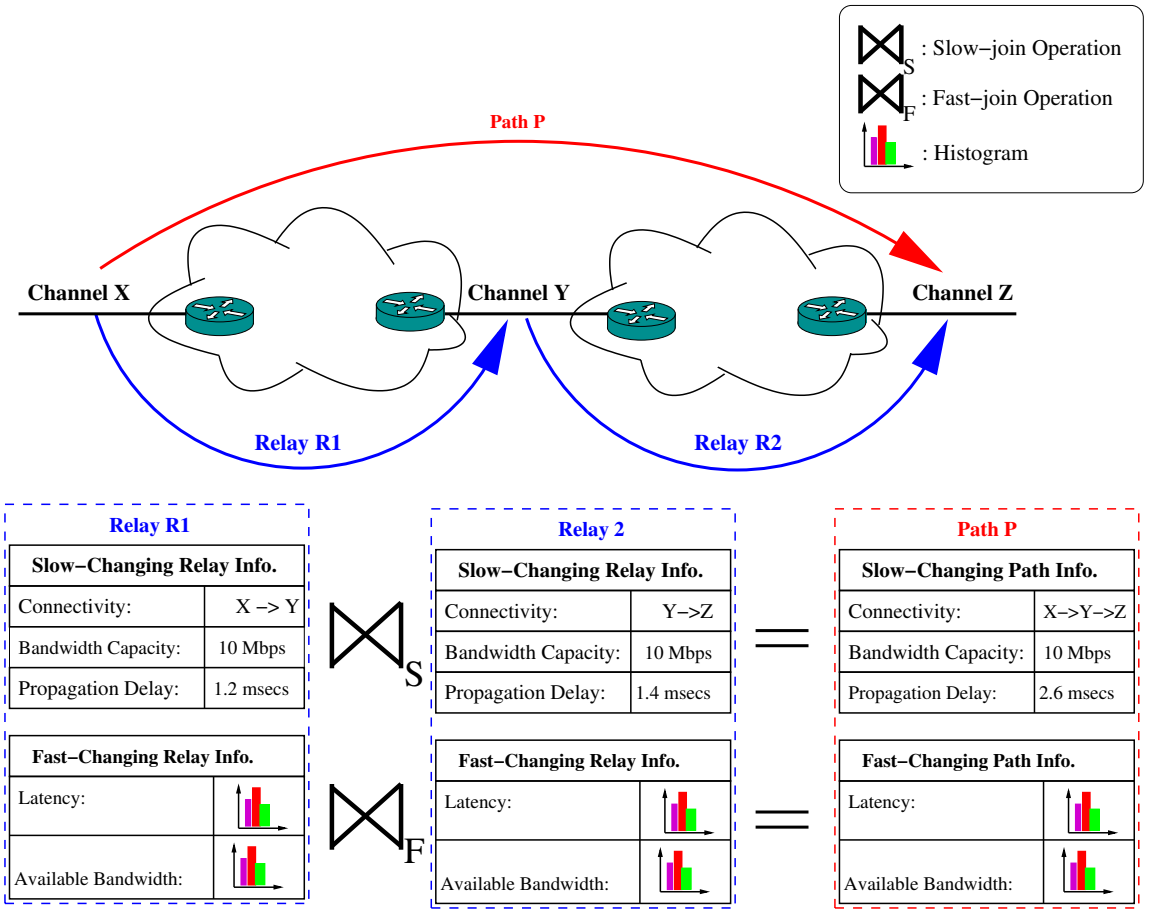

Figure 5.2: An example of slow-join and fast-join operations on two relays: R1 and R2 that form a path P.

An example for slow-join and fast-join operations is shown in Figure 5.2. In this example, slow-join and fast-join operations are performed on the relay information of the two relays: $R1$ and $R2$. Using the slow-changing information of the relays, the slow-join operations compute the slow-changing information of the path P. The connectivity of a relay comprises the ingress channel i and the egress channel e of the relay shown as $i \rightarrow e$ in the figure. The slow-join on the connectivity of the two adjacent relays (the egress channel of $R1$ matched the ingress channel of $R2$) forms

a sequence of channels that make up the path. A slow-join on the load-independent information of $R1$ and $R2$ are performed to compute the rest of the slow-changing attributes of P. In particular, slow-joins on the propagation delay and available bandwidth information are performed using addition and minimum operations, respectively. The fast-changing information is represented in the form of histograms. The join operations on the fast-changing information involves histogram operations such as convolution on the latency histograms and minimum operation on the available bandwidth histograms.

Unlike the QoS routing algorithms [74, 23] that perform the slow-join (i.e., constructing paths) and fast-join operations together (at the same time), our approach separates the two joins and performs them in different stages of the computation. In particular, the slow-join operations are performed during the pre-computation stage and the fast-join operations are performed during the on-demand stage of the path computation. An important observation is that when the two join operations are separated, both the slow-join and the fast-join operations are *parallelizable*.

Parallelizing the on-demand fast-join operations improves the responsiveness of the system—that is, how fast it responds to queries. A path query consists of i) a source, ii) a destination, iii) a traffic class selection from a set of available classes, and iv) the number of paths (i.e., k). The path service maps the traffic class to a set of constrains such as an upper-bound on the latency of the desired path.

Figure 5.3 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 Figure 5.3 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 which 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 the slow-changing and the fast-changing components that we discussed in the example above. The received information goes into databases of slow-changing and fast-changing routing information as shown in the figure.

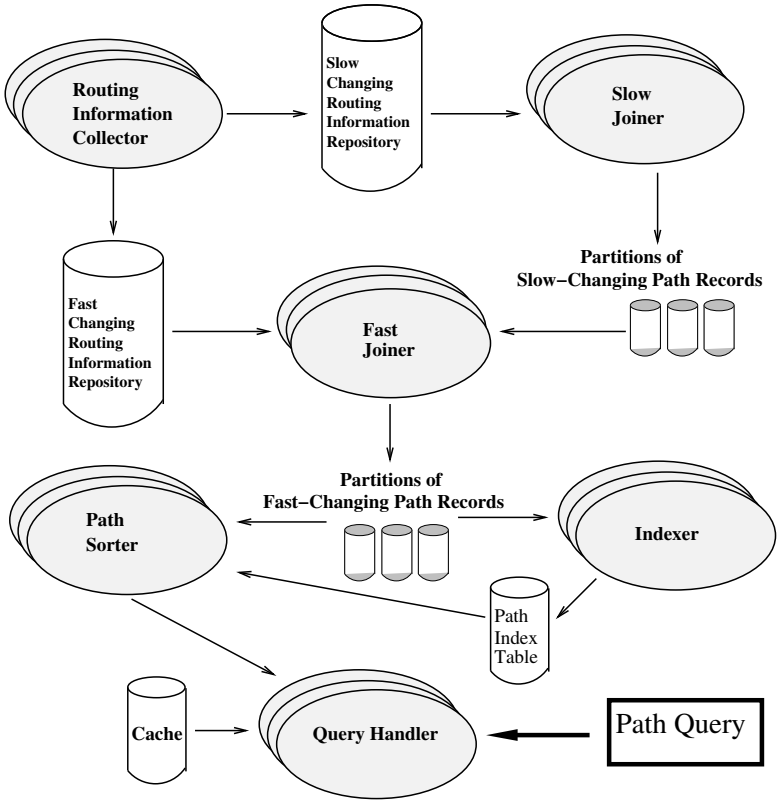

Figure 5.3: Path service architecture.

The slow-changing routing information is used by the slow-joiner to *pre-compute* partitions of *slow-changing path records*. A slow-changing path record contains the sequence of relays that make up the path, along with the aggregate slow-changing attributes derived from the slow-changing attributes of the relays in the path. Because the computation of a large number (e.g., over a billion) of possible paths is data-intensive (i.e., I/O bound), the MapReduce distributed computation model can be used for the pre-computation stage [24]. We will explain the details of the slow-joiner implementation in Section 6.2.1.

The fast-joiner computes the performance information (delay and available bandwidth distributions) of the pre-computed partitions of (i.e., slow-changing) paths by performing join operations on the histograms (i.e., fast-changing attributes). In the case of delay distributions, the fast-joiner performs approximate convolution on the histograms because exact convolution is computationally expensive. In particular, we use a quantizing approximation described by Xiao et al [87], 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. Convolution on two histograms with n bins has a time complexity of $O(n^2)$. 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 fast-joiners work in parallel—one fast-joiner per partition of slow-changing path records—to generate partitions of fast-changing path records.

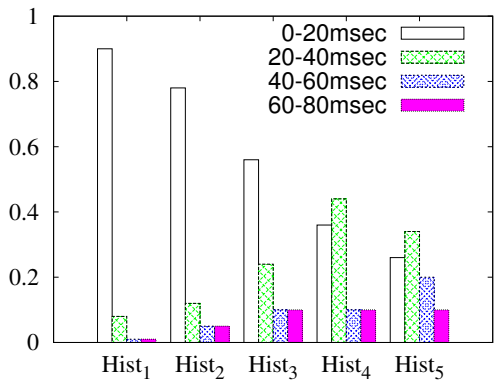

Figure 5.4: Latency histograms of five paths sorted from left to right according to their likelihood of satisfying delay $\leq$ 20 milliseconds.

The path sorter produces a ranking of paths according to their likelihood of satisfying the end-system's performance constraints. For each query that specifies performance constraints, the path sorter first computes each path's probability of satisfying the constraints, then performs the sorting operation. With one path sorter working

on each partition of fast-changing path records, the sorting operation is computed in parallel.

Consider a path query that requests k paths from S to D, with the delay upper-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. Figure 5.4 shows an example with delay histograms for five paths, where the bin boundaries are at 0, 20, 40, 60, and 80 milliseconds. The histograms are sorted from left to right in order of decreasing likelihood of satisfying the delay constraint: delay $\leq$ 20 milliseconds. Each sorter deals with a separate partition 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:

- Each path sorter retrieves the fast-changing path records with the source domain S and the destination domain D from its partition. Indexers pre-compute a path index table which is used by the sorters to locate paths by source-destination domains.

- Each path sorter computes the likelihood of satisfying the delay constraint for each path. This computation requires a single pass through the entire list of S to D paths. The linear search through the paths produces the k paths in the partition with the highest likelihood of satisfying the constraint.

- The individual results of the sorters are merged to obtain the k overall best paths.

Path queries that include additional hints will 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 updates the table only when the list of path records changes as a result of changes in slow-changing routing information (i.e., infrequently).

Having provided a high-level overview of the system architecture, a detailed description of an implementation of the path service architecture is presented in the next chapter.

Chapter 6

Evaluation

This chapter describes an implementation of the previously described path service architecture and the experiments conducted using the implementation. We first describe the experiment methodology in Section 6.1. In Section 6.2, we describe the process of generating paths that we use in our experiments. We provide a detailed description of the path service implementation in Section 6.3. Section 6.4 presents the performance results of the path service implementation on a single server with multiple cores. In Section 6.5, we use the performance results of the single server implementation to simulate a path service consisting of multiple instances of the single server implementation working in parallel, under a realistic workload. Finally, in Section 6.6, we describe how the path service can return disjoint paths for resiliency.

6.1 Experiment Methodology

In this section, we describe the workload and the network models used in the experiments. We discuss the workload of the system consisting of the path queries and the relay advertisements in Section 6.1.1. Later in Section 6.1.2, we will describe the network model.

6.1.1 Workload Model

We used unsampled[1] NetFlow [43] traffic traces to obtain a realistic set of path queries originating from an access domain. The traces were collected at an egress point of a campus network in June 2012 for the duration of 24 hours. The traces consist of traffic flows leaving the campus network, with each flow represented by its source IP address, destination IP address, source port, destination port, and the protocol information.

A Netflow collector is a router which collects flow information from the passing traffic. The collector keeps active flows in its flow cache with a finite size. When a flow terminates with the exchange of FIN flags (in the case of TCP), or when a flow becomes inactive (i.e., no packets belonging to the flow are captured for more than 120 seconds), the flow information in the cache is written to a secondary storage. The expiration of inactive flows is necessary to prevent the flow cache from overflowing but possibly introduces errors—e.g., flows that send packets infrequently with more than 120 seconds gaps. However, the resulting workload is an over-estimation of the number of flows, and therefore it does not invalidate our results.

An outgoing flow leaving the campus network is considered a query to the path service. We mapped each destination IP prefix in the NetFlow data to a destination Autonomous System (AS) number using the CAIDA IP prefix-to-AS-mappings data set [3]. By doing so, we obtained the domain-level traffic matrix from a single access domain to all other domains. In addition to the sequence of flows, the NetFlow data also contains the flow initiation times that are the arrival times of the first packets of flows to the NetFlow collector.

In our experiments in Section 6.5, we use the flow initiation times as the query arrival times to the Path Service. We only used the peak hour traffic in the experi-

[1] "Unsampled" means the captured traffic traces include all the flows that were passing through the traffic collector.

ments in order to measure the performance of the path service under the most intense load. The peak hour happened on a business day between 11 am and 12 noon. The rate of the outgoing flows in terms of flows per second during the peak hour is shown in Figure 6.1. The average number of outgoing flows per second during the peak hour is measured as 30,200 flows per second. Because we map each flow to a query for the path service, the plot in Figure 6.1 presents the workload in terms of query arrivals per second.

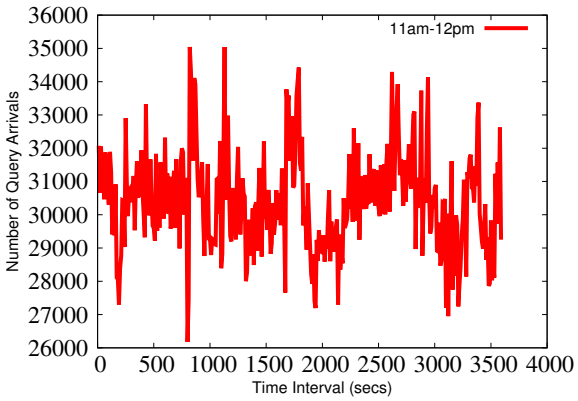

Figure 6.1: Rate of query arrivals during the busiest hour.

A path query consists of i) a source AS, ii) a destination AS, iii) a traffic class selection from a set of available classes, and iv) the number of paths (i.e., k). The traffic class selections are mapped to a list if performance constraints on latency and available bandwidth by the path service. In the case of latency, the constraint is an upper-bound, whereas in the case of available bandwidth, the constraint is a lower-bound. In the experiments, the traffic class in each query is either picked randomly or statically—the same traffic class is selected for all traffic to a destination domain. The list of available traffic classes that users can choose from are listed below. The list is based on the existing work on IP traffic classification such as the work described in [70]:

- Low latency (interactive traffic).

- High bandwidth (bulk transfer).
- Medium Bandwidth and Low Latency (video conference).
- High Bandwidth and Low Latency (real-time applications).
- Best Effort (web, email traffic).

Both queries and the relay advertisements that the path service periodically receives are generated by external components named *query-generator* and *update-generator*, respectively. Both of the external components communicate with the service through the Message Passing Interface (MPI) [2]. The query-generator can use either random queries or queries based on the NetFlow traces. The relay advertisement generator sends R relay advertisements every T_u seconds. We test the performance of the implementation for varying rate R and frequency $\frac{1}{T_u}$ of relay advertisements.

6.1.2 Network Model

The input to our experiments is an Internet AS-level topology which is constructed using CAIDA's AS relationship dataset [1]. The CAIDA AS relationship dataset consists of AS adjacencies along with annotated relationships such as customer-provider and peer-to-peer. We used the CAIDA dataset to build an initial topology by adding a single channel between the domains that have business relationships according to the dataset. As we discuss in Sections 6.1.2.1 and 6.1.2.2, we made modifications to the initial topology to obtain an equivalent AS-level topology that met our requirement that domains be either access or transit, but not both.

The current Internet topology, having domains that act both as access and as transit at the same time (e.g., a tier-1 domain hosting content), does not map directly to our model where each domain is transit or access, but not both. We describe the separation of access and transit roles in Section 6.1.2.1. We also added additional

channels to the topology to reflect the existence of multiple channels between the domains in the Internet and to reflect the possible implication of settlement-free relationships. We describe the addition of extra channels to the CAIDA topology in Section 6.1.2.2.

6.1.2.1 Separating Access and Transit Roles

We began by transforming the CAIDA AS topology into a domain-level topology by separating the roles of access and transit. The two steps of the transformation are: i) identify ASs in the CAIDA topology with mixed-roles, and ii) split the ASs with mixed-roles into an access and a transit domain, connected to each other by a channel.

In order to identify domains with mixed-roles, we first classified each domain in the CAIDA topology as either i) *primarily transit* or ii) *primarily access*. A method suggested by Dhamdhere et al [27] infers the primary business of each domain based on the size of its customers, providers and peers. In particular, the method classifies each domain into one of the three categories: i) *Transit Providers* (TPs), ii) *Content/Access/Hosting providers* (CAHP), and iii) *Enterprise Customers* (ECs). The EC domains correspond to various companies, universities and organizations. Applying the above classification method to the CAIDA data resulted in 35,287 domains that are primarily EC, 4,646 primarily TP, and 1,755 primarily CAHP. Using these results, we formed the set of primarily access domains from the domains classified as either EC or CAHP and formed the set of primarily transit domains from the domains classified as TP.

We then used the NetFlow traces to identify domains with mixed-roles among the primarily transit domains. Our approach to identify mixed-role transit domains is based on determining whether the domain is the destination in a large percentage of the flows in the NetFlow traces. Based on the traces, 2,700 of the 4,646 primarily transit domains were the destinations in a significant number of flows (i.e., over 10% of all the collected flows). Therefore, we identified those 2,700 domains as mixed-role

transit domains. We also identified mixed-role primarily access domains using the CAIDA AS relationships data. In particular, a primarily access domain is identified as mixed-role if the domain is the transit provider of at least one other domain according to the CAIDA data. Finally, we split the mixed-role domains into a transit and an access domain, connected with a channel. After splitting the domain into two, we assign all the existing channels of the original domain to the transit component.

6.1.2.2 Adding Extra Channels

In the Internet today, there are typically multiple channels between domains, especially between large ISPs. However, the CAIDA AS relationship topology dataset does not provide the number of channels between the ASs. In order to reflect the multiple channels in our network model, we added 60,000 additional channels[2] between the transit providers, placing them between pairs of adjacent transit domains. We performed a biased random selection of channel endpoints by favoring high degree[3] domains in the selection. In particular, we followed the three step procedure to add a new channel: i) pick one domain with a probability proportional to its degree, ii) pick a second domain from the existing neighbors of the first domain with a probability proportional to its degree, and iii) add a channel between the two domains.

After adding extra channels between the transit domains, we also added new channels from access domains to transit domains in order to reflect the possible impact of settlement-free relationships between the domains (as described in Section 3.1). In particular, 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 (as opposed to 2.8 in the original topology). When adding the extra channels to access domains, we selected access providers uniformly at random in order to create a topology where

[2]60,000 additional channels equate to around 10 extra channels per transit domain on average. However, we performed a biased random selection favoring high degree domains when choosing the channel endpoints. As a result, the largest transit domains obtained hundreds of additional channels.

[3]We made the assumption that the number of channels of a domain (i.e., degree) is a good indication of its size.

all access providers were equally likely to be multi-homed. The resulting topology contains 37,986 access domains and 6,402 transit domains. In the topology, there are 171,074 bidirectional channels.

The final topology approximates the structure of the current Internet topology. Because our approach scales through parallelism, future Internet topologies with larger number of paths than the current Internet can be accommodated by adding additional instances of parallel components to our path service. Next, we describe how the paths are generated in the pre-computation stage.

6.2 Slow-Joiner: Generating Paths

The challenge in computing paths in a large topology with around 44K nodes is both the resource requirements in terms of memory usage and performing the computation in a timely manner. We describe a method which uses an acceptable amount of memory (less than 64 GB) and can complete the computation in five hours on our (modified) Internet topology of roughly 44K nodes. Note that the slow-joiner computation is by design intended to be performed infrequently (on the order of days rather than hours), so a completion time of five hours is more than acceptable.

Using the topology described in Section 6.1.2, we computed the following paths from a random access domain S in two steps: i) for each destination access domain D, we computed the shortest paths from each egress channel S_e of S to each ingress channel D_i of D using Dijkstra [28] to obtain the length of the shortest paths between source egress and destination ingress channels, ii) for each S_e of S and D_i of D, we computed all the paths from S_e to D_i with length up to $L + 2$, where L is the length of the shortest path from S_e to D_i. This resulted in approximately 1.65 billion unique paths to all destination access domains from source domain S.

In generating these paths, we made certain assumptions about the routing policies of the transit domains. Specifically, we distinguished between the transit providers

(TP_{sec}) that resulted from splitting primarily access domains (with mixed-roles) and the transit domains that are classified as primarily (TP_{pri}) transit provider. For the latter, we assumed an unconstrained policy: they offer relays between any two of their channels, since their business is primarily transit. The use of such an unconstrained policy increases the number of paths between the access domains, and thus makes it more challenging to scale the path service. On the other hand, we assumed a more constrained policy for the transit domains in TP_{sec}. Because the TP_{sec} domains exist mainly to connect their access domain neighbors to the rest of the Internet, we assumed that the TP_{sec} domains only transit traffic between their access domain neighbors and their transit domain neighbors. Using the two types of transit policies, we obtained a total of 157 million relays offered by the transit domains. However, the computed paths from the random source access domain S to all destination domains uses only about 11 million relays out of 157 million—mainly because of the constraint on the length of the paths.

The computed paths are kept in the main memory as a list of *path records*. As shown in Listing 6.1, a path record structure contains the source (line 3), the destination (line 4), the slow-changing attributes (line 6), the fast-changing attributes (line 7) and the list of references (line 5) to *relay records* that form the path. Similarly, each relay is stored as a relay record in the main memory. A relay record structure, shown in Listing 6.2, is identical to the path record structure with the exceptions of the references to relay records and a readers-writer lock. We discuss the relay records and the required locking mechanisms on the relay records in Section 6.3.5.

The slow-join operation computes and writes all members of the path records except the fast-changing attributes. We refer to these path records that are precomputed by the slow-joiners as *slow-changing path records*. The fast-changing attributes of path records are computed and written during the computation of the query results—paths that are most likely to satisfy the performance constraints of

the application. A path record becomes *stale* when the fast-changing or the slow-changing attributes of one or more relays that form the path change with the arrival of fresh relay information. We describe how the implementation deals with staleness in Section 6.3.3

```
1 struct PathRecord
2 {
3   Channel source;
4   Channel destination;
5   RelayRecord *relays[];
6   SlowChangingAttr s_attr;
7   FastChangingAttr f_attr;
8   Time lastWrite;
9 };
```

Listing 6.1: Path Record Structure.

```
1 struct RelayRecord
2 {
3   Channel ingress;
4   Channel egress;
5   SlowChangingAttr s_attr;
6   FastChangingAttr f_attr;
7   ReadWriteLock lock;
8   Time lastWrite;
9 };
```

Listing 6.2: Relay Record Structure.

6.2.1 Slow-Joiner Implementation

The pre-computation of paths is a time-consuming operation in large graphs such as the Internet AS-level graph. Even though changes to the slow-changing path records happen infrequently (as a result of permanent changes in the connectivity), it is still important to complete the pre-computations in a timely manner (i.e., at most a few hours). To that end, we use a *parallel* breadth-first search method which is effective in reducing the computation time. However, parallel breadth-first search requires a significant amount of memory when computing paths, because the frontier (i.e., incomplete) paths are kept in the main memory. We use two strategies to

significantly reduce the memory usage: i) divide the problem of computing paths from a single source domain to all destination access domains into subproblems of computing paths from an egress channel S_e of the source domain to an ingress channel of a destination access domain D_i, and ii) pre-compute distances from all the channels to the destination ingress channel in order to identify and filter frontier paths that are to exceed the permissible length (i.e., L+2) of the path from S_e to D_i, as they are generated.

To further reduce the memory usage, the path generation method first generates paths on an "abstract" AS-level topology that contains a single *"virtual channel"* between all the neighboring domains that are connected by one or more physical channels in the actual topology. In the second step, the set of paths generated using the abstract topology are duplicated using the virtual-to-physical channel mappings. Generating paths using the two-step approach does not miss any paths that a breadth-first search method would compute on the actual topology.

The procedure to compute paths from a single source domain S to all destination domains is given in Listing 6.3. In this approach, we first generate an initial set of paths (Lines 4–11) using the abstract topology, named $T_{abstract}$. The path generation procedure is divided into smaller sub-problems of computing paths from an egress channel S_e of the source domain to an ingress channel D_i of the destination domain. The procedure computes all the paths from S_e to D_i with length up to $L + 2$, where L is the length of the shortest path (i.e., distance) from S_e to D_i. Then, the paths are duplicated with the multiple channels of the actual topology (Line 12).

The parallel breadth-first search method (Line 8) computes paths from an egress channel S_e of the source domain to an ingress channel D_i of the destination domain D. The method uses the pre-computed distances from the channel D_i to all other channels in the topology which are computed using the Dijkstra algorithm (Line 7). In particular, the pre-computed distances are used to eliminate frontier paths, whose

current length plus the pre-computed distance of its last channel to D_i exceeds the length constraint (i.e., distance between S_e and D_i plus two), as they are generated.

```
1  def Path_Generation(Topology_Graph T,Domain S)
2    T_abstract = Extract_Connectivity_Graph(T)
3    Paths = {}
4    for each egress channel S_e of S
5      for each Stub Domain D in T
6        for each ingress channel D_i of D
7          Dist_Di = Dijkstra(D_i,T_abstract)
8          Paths += BFS_Parallel(S_e,D_i,T_abstract,Dist_Di)
9        end
10     end
11   end
12   Final_Paths = Duplicate_Paths_Parallel(Paths,T_abstract,T)
13 end
```

Listing 6.3: Procedure to compute paths from domain S to all access domains.

Using the topology described in Section 6.1.2, we computed the paths from a randomly selected access domain to all other access domains using the above procedure. The computation resulted in 1.65 billion paths. The path computation procedure requires around five hours to complete on a single server with four Intel Xeon X5650 processors (six cores per processor) with 128 MB RAM, using 24 threads working in parallel.

The main bottleneck in the computation is the input and output since the computation is data intensive. Because the MapReduce programming model is a good fit for data intensive computations, we developed and used a parallel breadth-first search implementation in Hadoop [80]—an open source MapReduce implementation—to generate paths. The MapReduce implementation generates the same set of paths from a single source to all destinations with the length constraint in just an hour using 8 quad-core computers each with 500 GB storage.

The distribution of the number of paths to each destination domain is shown in Figure 6.2. It is important to note that a large percentage of the destinations (32,067 of 37,986 domains or 86% of all domains), have 60,000 or less paths. Also, a very small

percentage of domains, (517 domains or around 1% of all domains) have 300,000 or more paths. The domains with large number of paths have high connectivity and as a result, our channel-oriented path generation method computes more paths to such highly connected access domains. These domains include popular destinations such as content distribution networks (e.g., Akamai) and content producers like Google. The popularity of the domains with large number of paths makes it more challenging to scale the path computations. However, by caching recently computed paths, the path service can respond to queries that request paths to popular destinations directly from its cache.

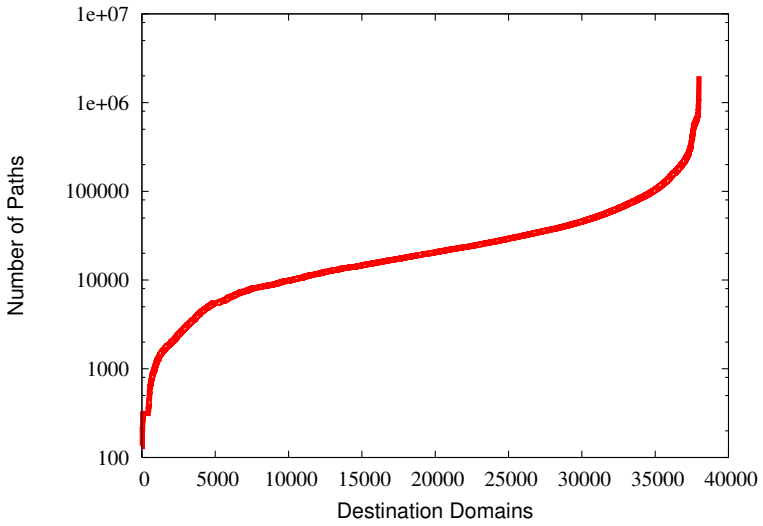

Figure 6.2: Distribution of paths to destination domains.

In the experiments, we assume that the slow-changing routing information is static. Therefore, the slow-join operation is only performed once and no incremental changes are made thereafter. Instead, we focus on the more challenging problem of scaling path computations with periodic changes in the fast-changing routing information. Next, we describe the path service implementation which only deals with the fast-changing routing information.

6.3 Path Service Implementation

The path service implementation follows the conceptual design depicted in Figure 5.3. The major components of the implementation are *master*, *workers* and *updaters* as shown in Figure 6.3.

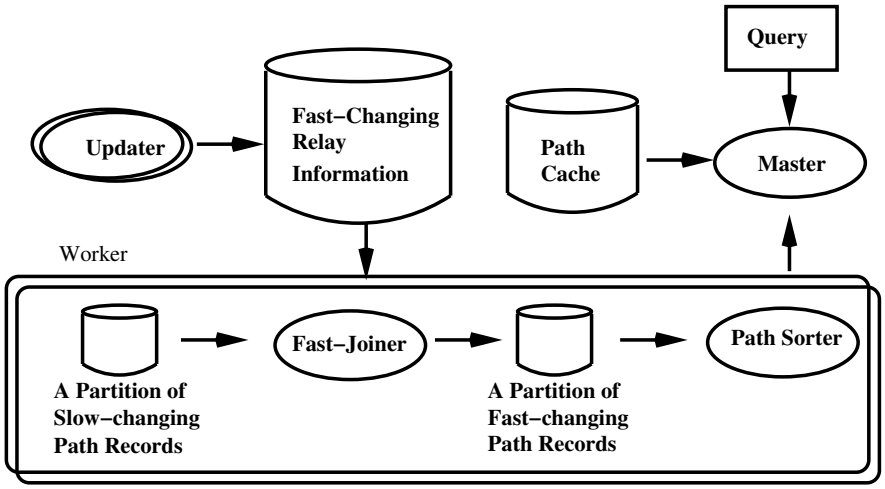

Figure 6.3: The main components of the path service implementation.

6.3.1 Master

The master consists of a communication component and a computation component. The communication component is responsible for sending queries to and collecting results from both the cache-controller (described in Section 6.3.2) and the workers. As explained in Section 6.3.3, the path service implementation distributes the work of computing query results among all the workers. The computation component of the master merges the individual results from the workers to obtain the final query results. The master is responsible for the following tasks:

1. Receive queries and forward them to the workers.

2. Collect results from all the workers.

3. Merge the individual results to produce the final query results.

4. Forward the final results to the cache-controller and the user.

The master performs three communication tasks (Tasks 1, 2, and 4) and a computation task (Task 3). It is important for the master to communicate with the workers in a timely manner for the path service to be work-conserving—that is, workers are busy computing results as long as there are queries that arrived to the system. In particular, if the master gets blocked with a computation task and does not forward the arriving queries to the workers immediately, the workers could stay idle instead of computing the query results. In order to prevent the computation task from blocking the communication tasks of the master, the computation component and the communication component of the master are separated, they work in parallel. The computation component consists of a single *merger* thread which computes the results to incoming merge requests and returns the results.

The master and workers communicate through a task queue and a results queue (i.e., completed tasks). The master inserts a query in the form of a task into the task queue, from which the available workers obtain the tasks. Once a worker completes a task, it inserts the result (i.e., paths with performance information) into the results queue for the master to fetch. Arrival of a query at the master results in exactly W tasks being inserted in the task queue when there are a total of W workers. This results in W query results from the workers to be merged. Another task of the master is to make cache-insert and cache-search requests to the path cache, which is described next.

6.3.2 Path Cache

The most recently computed results to path queries are stored in an in-memory *path cache* which is implemented as a hash table. The cache is managed by a *cache-controller* process that handles insert and search requests from the master. A cached entry in the path cache is shown in Listing 6.4. The cache maps a triple consisting

of a source domain, a destination domain and a traffic class to a set of top k paths (i.e., query results).

```
1  struct CachedEntry
2  {
3    int sourceDomain;
4    int destinationDomain;
5    int trafficClass;
6    PathRecord *topk[k_max];
7    int timestamp;
8  };
```

Listing 6.4: A Path Cache entry containing information about the top k paths resulting from a query.

One concern over the caching of query results is the lifetime of the paths. With the arrival of new routing information (in the form of relay advertisements), some of the cached paths to a destination domain D may become stale. However, it is not possible to determine whether a cached path to D is stale after the arrival of new routing information without repeating the path computation to D using the most up-to-date routing information. Although one could imagine a more efficient algorithm to determine whether a cached path needs to be recomputed or not, the cache-controller simply treats all the existing cached entries as stale entries (and does not use them) once new routing information arrives.

It is important for the cache-controller to respond to search and insert requests in a timely-manner. Therefore, even if fresh routing information arrives, it is not desirable for the cache-controller to flush all the existing cached entries. Instead, the cache-controller updates entries only when a search or insert operation finds a stale entry. To determine staleness, the cache controller timestamps each entry in the cache as it is inserted (in the timestamp variable in Listing 6.4). A comparison of the timestamp in a cached entry with the arrival time of the most recent routing information determines if the entry is stale.

6.3.3 Worker

A worker is a combined *fast-joiner* and *path sorter* as shown in Figure 6.3. The main task of the worker is to compute results for a particular query. The workers compute the results of a query in parallel. In order to distribute the work of computing query results, each worker is assigned a single partition of slow-changing path records, and the workers compute the results using only the partition that is assigned to them. After the master forwards the query to the workers, each worker computes and returns a list of path records. The master then merges the individual results from each worker to produce the final results, as discussed before.

The task of the fast-joiner component is to compute the fast-changing attributes of the path records in the assigned partition of path records using the fast-changing relay (routing) information. The fast-changing information consists of a list of relay records, which contains both slow- and fast-changing attributes of relays. Similar to path records, relay records are kept in the main memory for fast access by the workers. Once the fast-changing attributes of paths are computed, they are stored as part of the path record to use for future queries. However, the fast-changing attributes of a path record need to be re-computed when the path record becomes stale. As mentioned before, a path record becomes stale when the fast-changing attributes of one or more relays used in the path change. In order to check for staleness, both path records and relay records contain a member to record the time of the most recent write to the fast-changing attribute. We refer to path records whose fast-changing attributes are computed as *fast-changing path records* in Figure 6.3. The actual worker implementation uses a single copy of each path record; the two partitions—slow-changing and fast-changing—are used in the Figure 6.3 for presentation purposes.

In our experiments, we use latency and available bandwidth as the two fast-changing attributes. Both of the attributes are represented in the form of histograms with five bins that represent probability distributions. The value of each bin is an

integer between zero and 100, and the sum of the bin values in a histogram is 100. As mentioned before, an external (simulation) component named update-generator periodically sends relay (i.e., routing) information updates to the path service. These updates contain fast-changing attributes for a list of relays, with randomly generated histograms of probability distributions. The fast-joiner component computes convolutions on the latency histograms and minimum operations on the available bandwidth histograms to produce path histograms.

The path sorter component performs a linear search on path records to obtain the top k_{max} paths, where k_{max} is the largest k value that the system allows for queries. We set k_{max} to five in our experiments as this provides senders with a reasonable number of alternatives. As the path sorter goes through the list of path records, it maintains a list of the current top five path records. Once the search is complete, the list of the top five path records is returned to the master. In order to obtain the top five paths, the path sorter computes the likelihood of each path record satisfying the constraints on latency and available bandwidth. The constraint on latency is an upper-bound, while the constraint on available bandwidth is a lower-bound value. In the implementation, the fast-join and path sorter components operate together in a sequence—they are implemented as part of the worker thread. After the fast-joiner computes the attributes of a path, the path sorter computes its likelihood of satisfying the constraints.

Each worker maintains an *index* to locate the path records for a specific source and destination channel in its path record partition. Upon receiving a query, a worker first looks up in its index to locate paths whose source and destination matches the source and the destination requested in the path query. *Our hypothesis is that when each partition of path records contains roughly the same number of paths with similar lengths for a given source-destination pair, each worker has roughly the same*

computational load.[4] We test our hypothesis in Section 6.4.4. The pre-computed paths by the slow-joiner are partitioned in such a way as to ensure that each worker obtains a similar load, as follows:

1. Pre-computed paths are grouped by source and destination domains.
2. Within each group, paths are sorted by their length (i.e., number of relays).
3. Grouped and sorted paths are assigned to workers in a round-robin fashion.

The majority of the time that a worker spends computing query results is during the fast-join operation. During this operation, the worker iterates over the paths in its partition and computes the path attributes through a sequence of histogram operations (i.e., convolution and minimum). In order to improve the performance of this step, we perform an additional optimization step to reduce the repeated computations for paths that have common portions. This optimization step is explained next.

6.3.4 Grouping Together Paths with Common Prefixes

The pre-computed paths from one source domain to a destination domain are likely to contain common prefixes. As an example, consider two paths that traverse the same relays to reach a particular tier-1 domain T and then follow different relays from T to the destination. By grouping paths with similar prefixes in each partition, the fast-joiner component can avoid the repeated computation when two subsequent paths have a common prefix. To group paths with similar prefixes together, each sub-partition containing paths from one particular source domain to a particular destination domain is sorted lexicographically using the relay identifiers that form the path.

[4]The processing cost of a path is dominated by the histogram (i.e., join) operations to compute the fast-changing path information. The number of histogram operations performed on a path is based on the length (i.e., number of relays) of the path.

We also considered other potential solutions to avoid repeating join computations across different paths. One solution is to identify portions of paths (not necessarily prefixes of paths) that are most common across different paths and reuse the result of the join operation on the common sub-paths during the fast-joins. Identifying the most common sub-paths across a large number of paths is a challenging task which involves counting the occurrences of all possible sub-paths (across over a billion paths) and sorting them. This task can possibly be performed using a MapReduce approach. However, computing popular sub-paths and using them to avoid repeated join computations requires additional overhead in terms of (in-memory) storage, because one needs to store all the popular sub-portions and represent the paths in terms of both relays and popular path portions.

We use the simpler path grouping approach, because it requires no extra overhead except the grouping stage. Yet another possible solution is to group paths with similar suffixes rather than prefixes together. However, we found that grouping paths by common prefix or suffix makes a negligible difference in terms of their impact on the fast-join performance.

6.3.5 Updaters

While workers read fast-changing relay records from the local (in-memory) storage during fast-join computations, the system periodically receives fresh fast-changing relay information. The task of the updaters is to update the in-memory storage of relay records with the arriving fast-changing routing information. It is important to update the local storage with the new information as soon as possible in order to compute query results with the most up-to-date information. Therefore, one or more updaters perform the writing of new relay information to the local storage in parallel. Because the workers read from the relay records that are periodically written by the updaters, locking mechanisms are implemented to protect the integrity of the relay

records.

Another task of the updater is to inform the cache-controller of the arrival of fresh routing information in order for the cache-controller to identify stale cached entries. The updater and the cache-controller use MPI to communicate. In our experiments, updaters receive routing information updates periodically every T_u seconds, and each update contains the fast-changing attributes for R distinct relays. We use different T_u and R values in the experiments. The update-generator picks the relays to be updated randomly.

In the next few sections, we present our experiments with the path service implementation. In Section 6.4, we provide results using a single server implementation with four Intel Xeon X5650 processors containing a total of 24 cores and 128GB of memory. In this setting, the server has no input queue to store the incoming queries. Therefore, the server accepts one query at a time: it receives a query, computes the results and then obtains the next query. Later, we consider a distributed system with multiple identical instances of the single server implementation which has an input queue. We experiment with the distributed system using a realistic query arrival schedule in Section 6.5.

6.4 Experiments without Queuing

In this section, we measure the performance of the path service on a single server, where the service obtains one query at a time. The master, each worker, and each updater component is implemented as threads. In addition to these path service threads, we use two simulation processes: an update-generator and a query-generator to send relay information updates to the updater thread and to send queries to the master thread, respectively.

The performance metric of interest is the query processing (i.e., service) time, which we measure as the time from when the query is received by the path service

until the results are ready to be sent by the path service (In a real system there will also be latency to transmit the query and response, but if the service is located in the local access domain, we expect this to be small, comparable to DNS lookups today). The parameters varied include the number of worker threads and the frequency of routing information updates.

In the first experiments, presented in Section 6.4.1, we used path queries with random destinations and disabled the path caching in the path service. The goal of the first experiments is to collect the service times for each destination access domain when the computation is done in real-time (no caching). The collected service times are used in the simulation of the distributed system in Section 6.5. In the second set of experiments, presented in Section 6.4.2, we present the results of the experiments with the path caching enabled and compare the caching-enabled path service to the caching-disabled path service. In all the experiments that we present in this chapter, the traffic classes in the path queries are selected randomly, unless stated otherwise.

In the experiments that we present in the next two sections, the arrival rate of the relay information is set to one million relays every 10 seconds. The pre-computed paths from the source domain to all destination domains use about 11 million relays. At the rate of 6 million relays per minute, all the 11 million relays can be updated within 2 minutes. The update-generator picks the relays in the advertisements uniformly at random, and each relay advertisement contains random performance information containing randomly generated histograms of latency and available bandwidth.

6.4.1 Experiments without Caching

The cumulative distribution functions (CDF) of query service times for 6, 12, and 18 workers are presented in the three plots in Figure 6.4. We present the distributions of the query processing times using three plots by dividing the percentage of queries (x

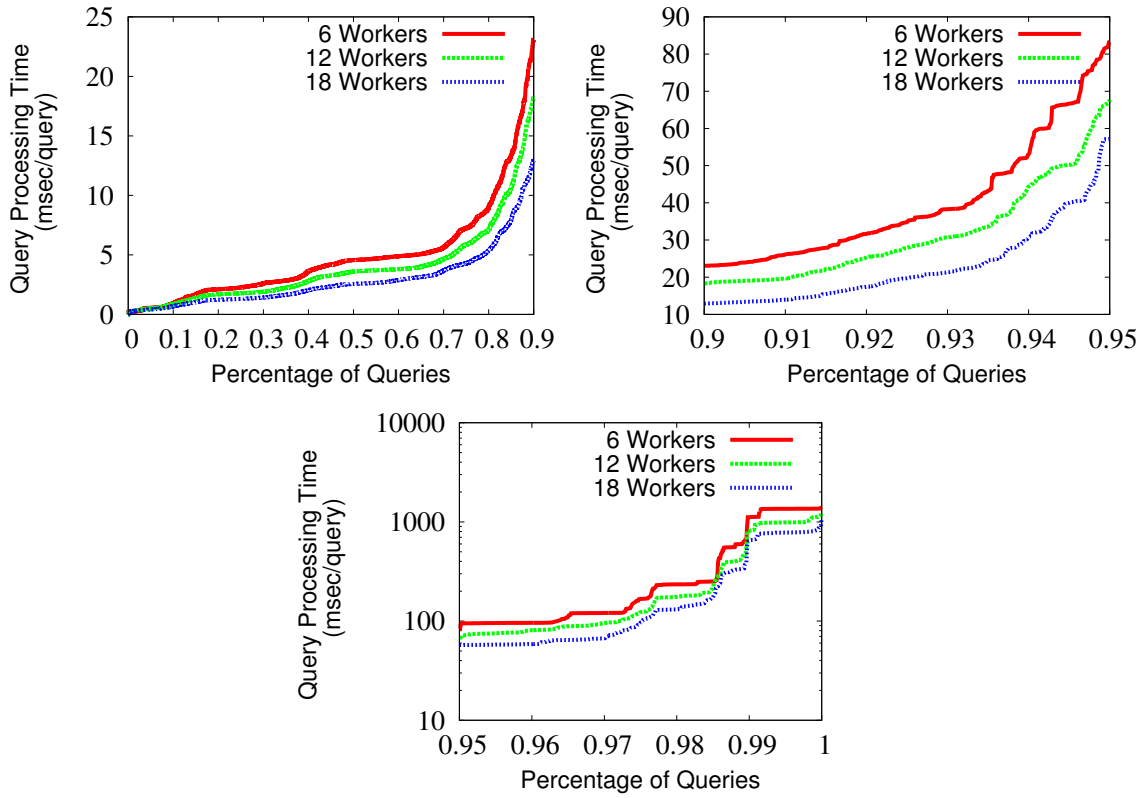

Figure 6.4: Cumulative distribution of query processing times for 6, 12, and 18 workers.

axis of the plots) into three ranges: i) 0–0.9, ii) 0.9–0.95, and iii) 0.95–1.0. Note that the y axis of the plot for the range of 0.95–1.0 is log-scaled. As expected, we achieved a (close to linear) decrease in the service times decrease as the number of workers that perform computations in parallel increases. The average of query processing times for 6, 12, and 18 workers are 27.36, 20.92, and 16.05 milliseconds, respectively.

6.4.2 Experiments with Caching

In the experiments we present in this section, we used path queries that we generated using NetFlow traffic traces collected from a campus network to real-world destinations. In particular, we selected a 1-minute interval which was representative of the other parts of the traces based on its locality of the reference—around 90% of the queries destined to less than 2000 destinations.

Figure 6.5 presents the results of the experiments. The plot on the left hand side of the Figure 6.5 presents the difference in the cumulative distribution of the service times for the path services with 12 and 18 workers. The initial flat portions of the curves correspond to the queries that are satisfied from the path cache. The cache hit rate of the path service with 18 workers is higher than the path service with 12 workers. The reason is that the path service with 18 workers has a smaller service time, and as a result it can cache larger numbers of paths between the arrivals of fresh relay information compared to the path service with 12 workers. The cache hit rate of the path service with 12 workers was 34%, while the cache hit rate of the path service with 18 workers was 52% in the experiments. The plot on the right hand side of the Figure 6.5 presents the difference between the path services with caching enabled and caching disabled.

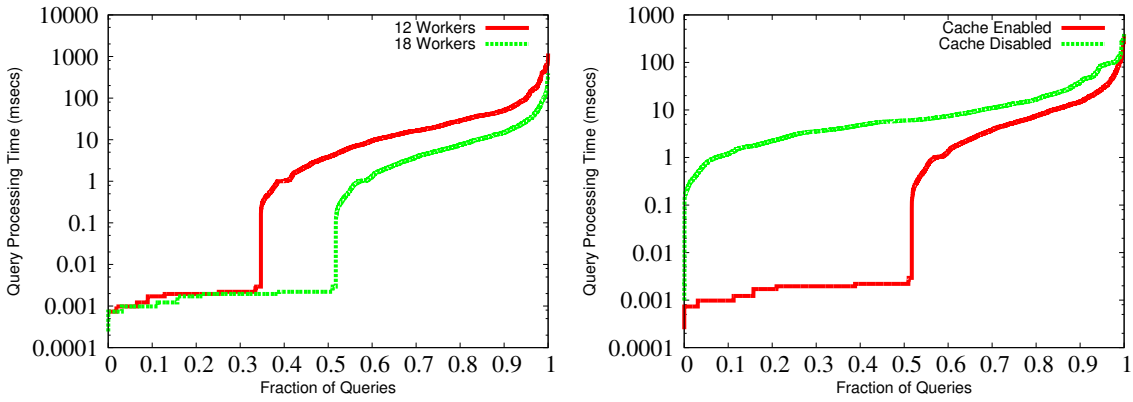

Figure 6.5: The plot on the left hand side presents the CDF of the service times for 12 and 18 workers with caching enabled, and the plot on the right hand side presents the comparison of the CDFs of service times for the caching enabled and the caching disabled path services with 18 workers.

In order to understand the impact of routing information (i.e., update) arrival frequency on the cache hit rates, we measured the cache hit rate as the update frequency is varied for a fixed number of 12 workers. The cache hit rates for the update intervals of 10, 15, 20, 25, and 30 seconds are shown in the plot in Figure 6.6. In this experiment, the number of relays in each update was set to a million relays.

As expected, the cache hit rates increase as the update frequency decreases. The increase in the cache hit rates translate to improvement in the service times for the path service.

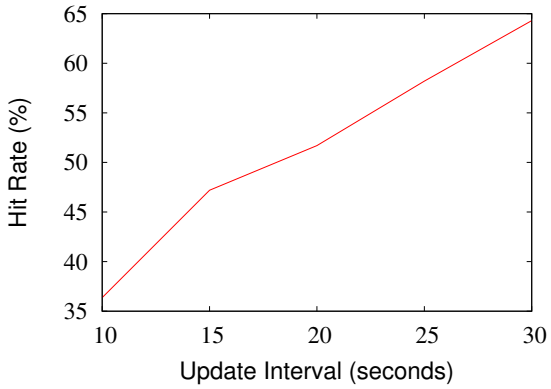

Figure 6.6: Effect of routing information arrival frequency on the cache hit rate.

In these experiments, the path service considers all the pre-computed paths between the source and the destination domains. In the current Internet, many access domains are multi-homed, and they have strong incentive to control how traffic enters and leaves their network to balance the load on their access channels. In the next section, we consider the scenario where each access domain selects a subset of its access channels for incoming and outgoing traffic.

6.4.3 Adding Ingress and Egress Traffic Engineering Policies

As cloud and content delivery services become popular in the Internet, multi-homing is also becoming important for providers to load-balance and to fail-over traffic. Multi-homed providers have strong incentive to control the ingress and egress routes of traffic in order to get maximum utilization from the aggregate bandwidth of the access channels.

In the current Internet, multi-homed access providers use name-resolution-based (DNS) and BGP-based solutions to perform ingress and egress traffic engineering [59]. In the BGP-based approach, an access provider A assigns blocks of IP addresses that

it owns to access channels by advertising each address block selectively to different upstream transit providers. In the name-resolution-based approach, the access domain runs an authoritative DNS server to resolve server names dynamically to IP addresses.

In the loose source routing architectures, we envision a mapping service which translates endpoint identifiers (EIDs) to sets of ingress and egress channels in the source routing system. Similar to the name resolution based approach of the Internet, access providers can control their ingress/egress traffic by providing authoritative mapping servers that dynamically resolve EIDs of the end-systems within their domain to ingress/egress channels. Before contacting the path service to obtain paths from the source domain S to the destination domain D, the end-systems query the authoritative mapping services of S and D to obtain a set of egress and a set of ingress channels, respectively. Once they obtain the ingress and egress channels, the end-systems query the path service to request paths from a set of egress channels to a set of ingress channels.

In our experiments, we assumed that the end-systems obtain 2 (random) egress and ingress channels for each multi-homed source and destination domains from the mapping service. Figure 6.7 shows the service times of the path service with access domains selecting up to 2 egress/ingress channels with 6, 12, and 18 workers. The average service times corresponding to 6, 12, and 18 workers are 7.12, 5.98 and 4.67 msecs, respectively. In these experiments the query processing times are significantly reduced in comparison to the results shown in Figure 6.4, because the ingress and egress channel selection reduces the number of paths considered by the path service when computing the results.

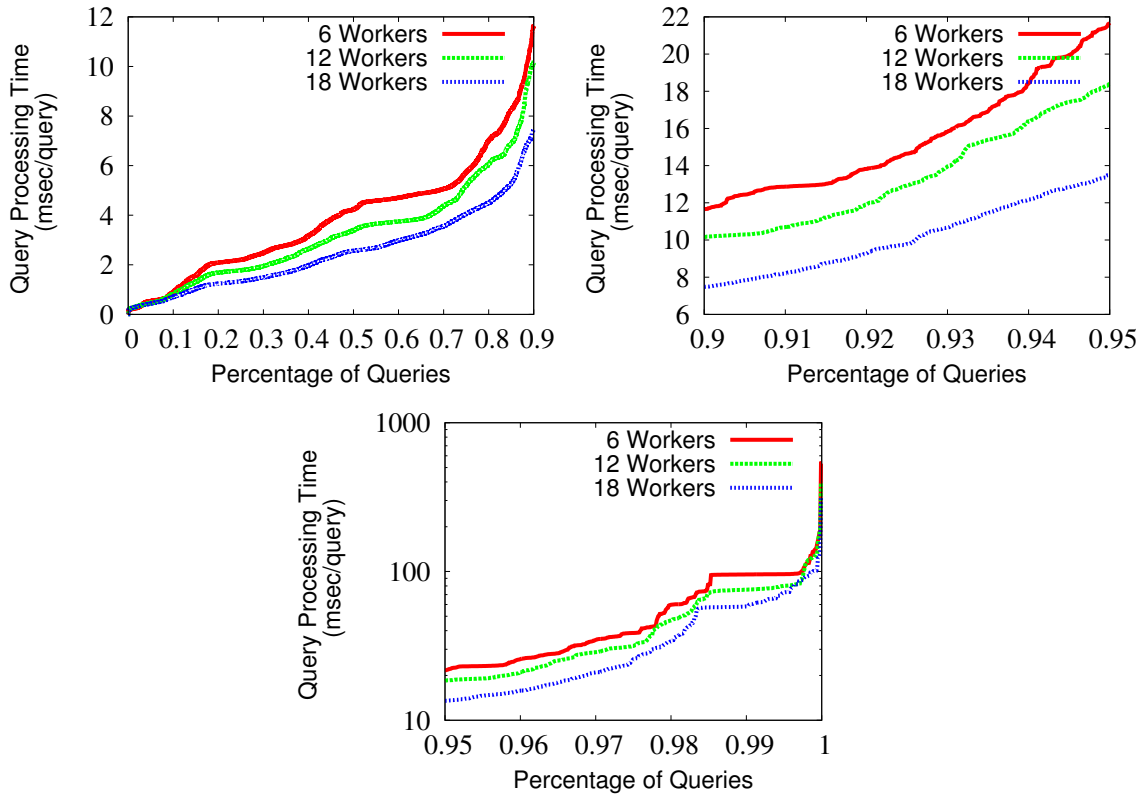

Figure 6.7: Cumulative distribution of query processing times for 6, 12, and 18 workers with ingress/egress traffic engineering.

6.4.4 Differences in Query Processing Times across Workers

An important goal of our design is the equal sharing of work among the workers. To achieve equal sharing of work among the workers, we make sure that each worker is assigned roughly an equal number of paths with roughly equal lengths, as explained in Section 6.3.3. Because the final merge of the individual query results of the workers has negligible contribution to the service times (below 0.01 milliseconds), the processing time of a query is dominated by the query processing time of the slowest worker.

We measured the query processing time differences between the slowest worker and the fastest worker to evaluate the balance of computational load across the workers. In the experiments that we present in this section, the path service uses all the pre-

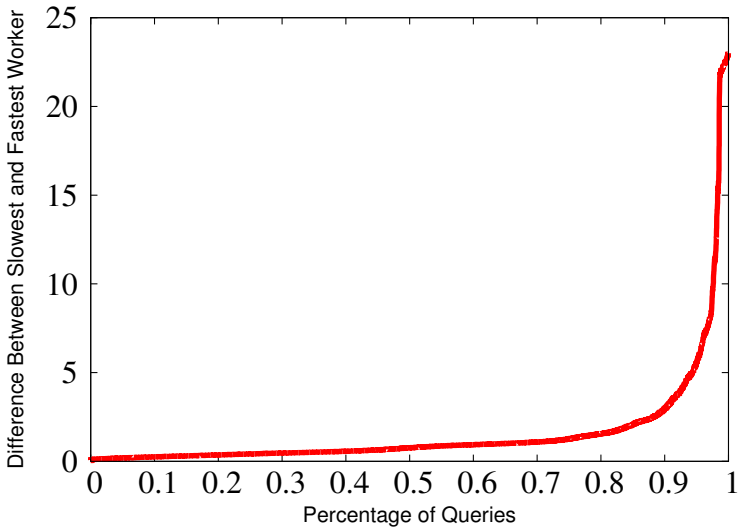

Figure 6.8: CDF of the query processing time differences between the fastest worker and the slowest worker.

computed paths between the source and the destination domains to respond to queries (i.e., no traffic engineering), and the caching was disabled.

Figure 6.8 shows the cumulative distribution of the query processing time differences between the fastest workers and the slowest workers across 100,000 random queries. For 80% of the queries, the difference in the service times between the fastest and the slowest worker is below 1.2 milliseconds. For less than 2% of the queries, the difference in service times is greater than five milliseconds and less than 23 milliseconds. The queries with the largest service time differences (i.e., over 20 milliseconds) correspond to destinations with service times over a second. Therefore, in comparison to the actual query processing times, the differences in the service times between the fastest and the slowest workers are fairly small (i.e., around 2% of the query service time).

One of the possible contributing factors to the differences in query processing times across the workers is the differences in the number of path comparison operations that each worker performs when computing the final set of paths (i.e., the task of the path sorter component). In particular, once a path's likelihood of satisfying a set of

constraints is computed, the path sorter compares the path with the worst path in the current top five paths list. If the path is worse than the last path in the top five paths list, the worker skips to the next path in the partition. Otherwise, the worker continues with the comparison of the path with the second worst path, and possibly the third worst path, and so on. In the next section, we describe the simulation of a path service consisting of multiple instances of the single server implementation which was described in this section.

6.5 Experiments with Queuing

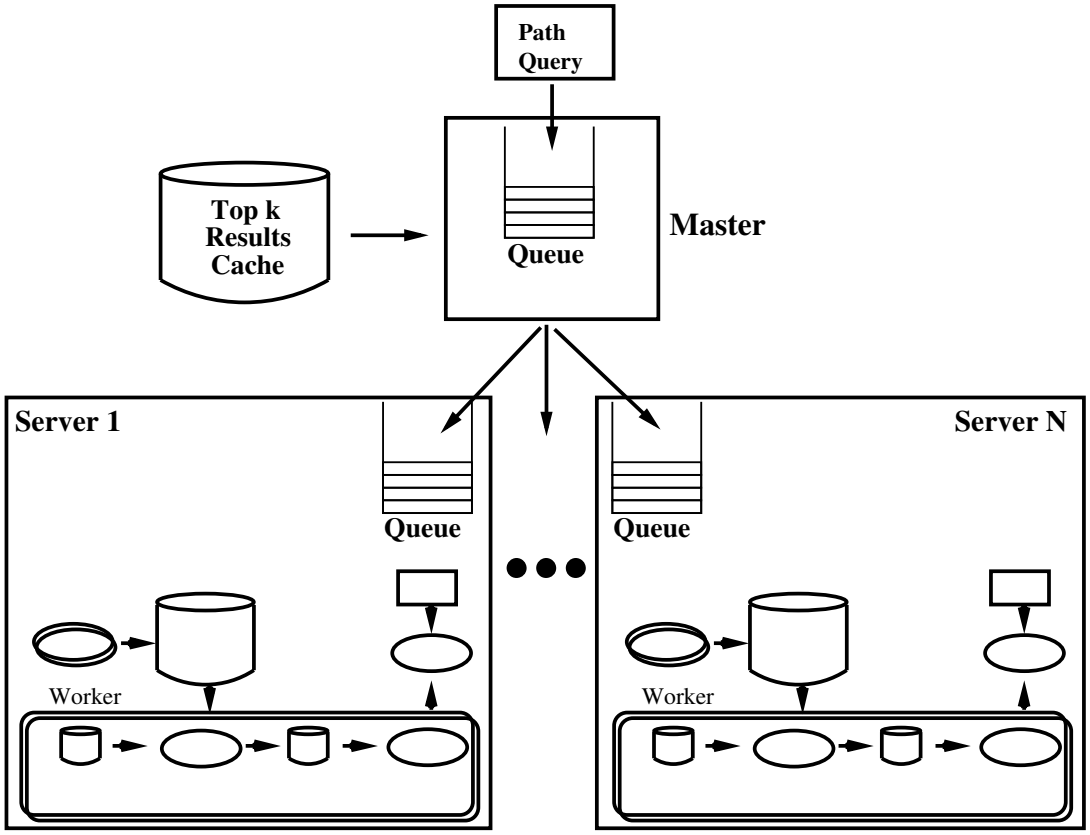

Figure 6.9: Path service with N replicated servers.

In the previous section, we considered a single server responding to queries that arrive one at a time. Obviously such a single server implementation cannot handle

realistic workloads. In this section, we consider a path service system that consists of multiple identical instances of the (slightly modified) single server implementation shown in Figure 6.9. The single server instances shown in the Figure 6.9 differ from the original implementation of the single server path service in that they do not have path caches to store the results of recently computed queries and have input queues to store incoming queries. Instead of having a separate path cache at each individual server, a single (external) path cache is implemented. The arriving queries in the distributed system are handled by a *master node*. Queries arrive to the master node according to the flow arrival times in the NetFlow data during the busiest hour of the 24 hour data at an average rate of 30,200 queries per second.

The master node assigns an incoming query to a single server and collects individual results from servers. To optimize the responsiveness of the system, the master node keeps track of the workloads of the servers and assigns an incoming queue to the server that is closest to completing its current workload (i.e., currently queued queries at the server). We compute the service times of the queries in the distributed system using the query service times obtained from the single server experiments with 18 workers and disabled path cache in Section 6.4.1.

Figure 6.10 shows the average query processing times for different number of servers running in parallel. With caching enabled, only 10 servers are sufficient to serve queries with an average service time of 3.2 milliseconds, when queries arrive to the system at the average rate of 30,200 queries per second. In this simulation, we assumed that the input queues of the servers were unbounded, which resulted in very large query service times when the number of servers was small.

In this simulation, we restricted the number of servers that handle a particular query to one. A possible extension to this approach is for multiple servers to process a query concurrently and return individual results, which are then merged by the master node to compute the final result. In that case, the paths to destinations

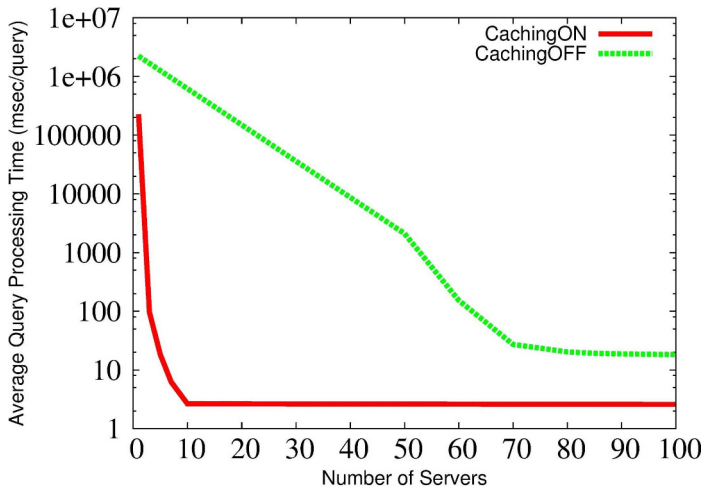

Figure 6.10: Average query processing times for different number of servers with caching disabled and enabled.

would be partitioned among the servers as opposed to partitioning among workers in a single server. This extension would allow larger sets of pre-computed paths (that one server by itself cannot store in its memory) to be considered by the system. We leave the implementation of such an extension to future work.

6.5.1 Overhead of Relay Advertisements

The majority of the relay updates consists of the fast-changing attributes of relays, and only a small portion of updates need to carry both the slow-changing and the 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 14 bytes. In particular, a typical update contains a globally unique relay identifier and the value of each individual bin (ranging between zero and 100) in the two histograms corresponding to latency 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 total bandwidth capacity re-

quired to transmit one million relays every 10 seconds under these assumptions is only around 11.2 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.

6.5.2 Stability of the System

Our loose source routing system with the path service contains a feedback loop, because it includes the load on the network to compute paths, and the paths that are returned by the system are eventually used by the end-systems to contribute new load on the network.

The original ARPANET [34], predecessor to the Internet, used hop-by-hop routing with load-dependent routing metrics in the path selection process. The nodes in the ARPANET network became aware of the topology and cost of all the channels in terms of load-dependent metrics through periodic broadcast of link-state updates upon significant changes in the channel costs. ARPANET had major problems with oscillations, especially when the utilization of the channels in the network is high [48]. The oscillation was due to the fluctuation of the channel costs: nodes advertised their current channel costs was not a good prediction of the new channel costs after all nodes re-route their traffic (on a single "best" path for each destination) based on the most recently reported channel costs. As a result, the channel costs changed rapidly, which triggered new broadcasts of updates, causing constant fluctuations in the routes. The effect of such rapid changes in routing (i.e., selection of best paths) is a disproportionate load assignment to the channels. In particular, while some channels are highly utilized, others stay under-utilized at any time.

Later, the ARPANET developers changed the instantaneous queue length metric to slower-changing metrics (e.g., average delay on the channels over few seconds)

in order to prevent rapid oscillations. However, this hardly improved the oscillation problem since the fundamental problem—that is, the feedback loop between next-hop selection and load—was still there. ARPANET was the earliest attempt to achieve dynamic load balancing with hop-by-hop routing.

One of the reasons that hop-by-hop routing is unable to achieve load-balancing with load-dependent metrics is the selection of only a single "best" path to each destination by each node. In the case of source routing, the path service can possibly balance the load between source and destination domains across a large number of paths. A possible way to achieve load-balancing across multiple paths using the path service is to randomize the path selection. In particular, instead of computing and storing (in the cache) only five paths per query as we did in the experiments, the path service could compute and store (in the cache) a much larger number (e.g., top 500) of paths that are most likely to satisfy a query. Then for each query, the path service could return a randomly selected set of five paths from the top 500 paths. The overhead of storing larger number of paths in the path cache is negligible, because a separate node with sufficient amount of memory can be dedicated for this task.

6.6 Adding a Disjointness Criteria

An important benefit of source routing is that it enables end-systems to react to failures (e.g., disconnections) by simply diverting their traffic from the primary (i.e., currently used) paths to alternate (i.e., backup) paths. However, if primary and backup paths have common (i.e., overlapping) channels, switching to a backup path may not lead to recovery, since the failure may occur on a common node or channel.

In this section, we consider the problem of computing disjoint[5] (i.e., non-overlapping) inter-domain level paths that also satisfy the performance constraints for robust com-

[5]Two channels in the topology that may appear to be disjoint can actually traverse the same physical wire or conduit. This presents additional problems, but for the purposes of this thesis we will assume channels that are disjoint in the graph are in fact disjoint.

munication. Unfortunately, there may be no channel-disjoint or node-disjoint paths between two access domains in the Internet (e.g., if one of them is single-homed). Therefore, we consider a metric to compute the *degree of disjointness* on a pair of paths. We use this metric to find *maximally disjoint* inter-domain paths—that is, paths with minimum overlap.

A commonly used metric to compute the degree of disjointness on two paths is: $\text{Disjointness}(P_1, P_2) = \frac{\text{Number of common components}(P_1,P_2)}{\text{minimum}(\text{length}(P_1,P_2))}$ [77]. The "number of common components" is the number of channels common to two paths when computing channel-disjoint paths, or it is the number of common domains when computing node-disjoint paths. The disjointness metric produces values between zero and one, where zero means disjoint. We use the above disjointness metric as the ground truth in the experiments below. The disjointness metric is the inverse of the *similarity* metric which is computed as: $\text{Similarity}(P_1, P_2) = 1 - \text{Disjointness}(P_1, P_2))$.

Our goal is for end-systems to obtain maximally disjoint paths from the path service. However, disjoint paths may not be needed in all the communications in the Internet—short-lived, best-effort communications (i.e., accessing a web page), for example. Also, some communications can tolerate a few RTTs of delay to obtain disjoint paths from the path service upon failures. An on-demand approach would save the path service from unnecessarily computing maximally disjoint paths for all path requests. For the path service to compute on-demand maximally disjoint paths, we introduce a new type of query, namely the *disjoint path query*. In the disjoint path queries, the end-systems provide a single path P (which was obtained from the path service earlier) in addition to the rest of the items in a typical path query: the destination, the number of paths requested, and the traffic class selection[6].

For disjoint path queries, the path service computes and selects paths according to two criteria: i) the least overlap with the path P supplied by the application, and

[6]A possible extension to the disjoint path queries, which can easily be supported, is for end-systems to also select between channel-disjoint and node-disjoint computation.

ii) the highest likelihood of satisfying the application's performance requirements. In order to compare paths based on two metrics, we use a nonlinear combination of the two metrics, similar to the approach used by the exact QoS routing algorithms when computing paths with multiple constraints [74] (See Section 4.3). In particular, the path service computes the *maximum* of the similarity (i.e., inverse of disjointness) and the likelihood metrics to produce a single "goodness" metric for the paths. The path service then returns the paths with the highest goodness metric to the users in response to disjoint path queries. By using the maximum operation, the path service computes paths with the highest likelihood of satisfying the performance requirements that are also maximally disjoint from the given path P.

The computation of disjoint path queries involve a large number of disjointness computations. More specifically, the path service must compute the disjointness of the given path P against all the pre-computed paths that are destined to P's destination. Because the pre-computed paths change slowly, a pre-computation approach, whereby the disjointness metric is computed pairwise between all possible pairs of paths within the groups of paths with the same destination in advance, can be considered. Pre-computation of disjointness can also be parallelized across distributed nodes to scale the computation; however, the pre-computed disjointness information requires about a terabyte[7] of storage (ideally in the main memory for fast access). Therefore, the path service computes disjointness of the user supplied path against the pre-computed paths in an on-demand fashion.

To reduce the complexity of the on-demand disjointness computations, we use approximations of the (ground truth) disjointness computations. In the approximate disjointness computations, each path (i.e., set of channels) is represented by a *Bloom Filter (BF)* [15] which is a compact bit vector representation of a set using a prob-

[7]There are on average 10,000 pre-computed paths to each destination and around 35,000 destination domains which results in approximately $35000 \times \binom{10000}{2} \approx 10^{12}$ disjointness values to be computed.

abilistic data structure. The path service computes BF of each path when the path is constructed (during the pre-computation phase) and stores its representation in the path record. The metrics we consider approximate the size of the intersection (i.e., overlap) of two Bloom Filters, and then normalize the size of the approximate intersection to produce values between zero and one. More specifically, we consider the following approximate disjointness metrics that operate on two bit vectors B_1 and B_2 with equal sizes:

1. $\text{Disjointness}_1(B_1, B_2) = \frac{\text{popcount}(B_1 \text{ AND } B_2)}{\text{minimum}(\text{popcount}(B_1),\text{popcount}(B_2))}$
2. $\text{Disjointness}_2(B_1, B_2) = \frac{\text{popcount}(B_1 \text{ AND } B_2)}{\text{maximum}(\text{popcount}(B_1),\text{popcount}(B_2))}$
3. $\text{Disjointness}_3(B_1, B_2) = \frac{\text{popcount}(B_1 \text{ XNOR } B_2)}{\text{Number of bits in } B_1}$
4. $\text{Disjointness}_4(B_1, B_2) = \frac{\text{popcount}(B_1 \text{ AND } B_2)}{\text{popcount}(B_1 \text{ OR } B_2)}$

Population count or *popcount* of a bit vector B is the sum of the digits of B. In recent CPU architectures, population count is implemented as a single hardware instruction which operates on 64 bit values. Because of its efficiency, we use the popcount operation on a BF B to approximate the size of the set represented by B.

The first disjointness metric performs a bitwise AND operation on the two BFs, which is an approximation for an intersection operation on two sets represented by the filters. Then, the metric divides the popcount (i.e., approximate size) of the intersection set with the (approximate) size of the smaller of the two sets. The second metric differs from the first by using the maximum operation instead of the minimum operation in the denominator. The third disjointness metric uses a bitwise XNOR operation and the popcount of the result to approximate the size of the intersection of the two sets. The XNOR of two BFs computes the approximate intersection of two filters by counting the number of positions where the two filters have the same value. The popcount of the intersection is then normalized by the number of bits in

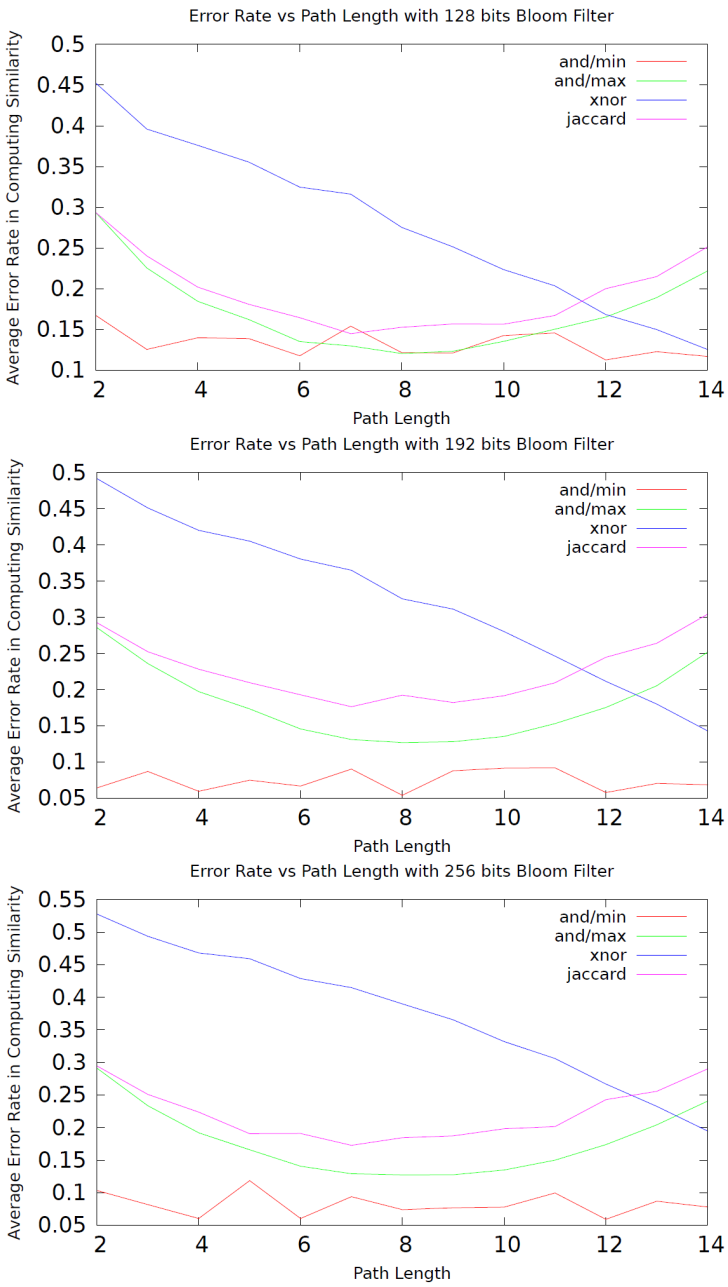

Figure 6.11: Average error rate of the disjointness approximations when a random fixed length path is compared to a million random paths with arbitrary lengths.

the BF. The final metric, also known as the *Jaccard distance*, computes the size of the intersection set and divides the intersection size by the size of the union set.

The approximations produced by the four metrics are compared against the ground truth to produce an *error rate* between zero and one, over a million path pairs. A

large error rate indicates a large absolute difference with the ground truth value. We present the error rates of the four metrics in Figure 6.11. In the plots shown in Figure 6.11, AND/MIN, AND/MAX, XNOR, and Jaccard refer to the first, second, third and fourth metrics, respectively. Our results can be interpreted as either channel-disjoint or node-disjoint path computations.

In each plot, a random path with fixed length L is compared with million random paths with lengths selected randomly from the range of 2–14 which corresponds to the range of path lengths in our network model. The comparisons are repeated for L values between 2 and 14 (inclusive) to obtain the error rate for each L value in the plots. We perform the same set of comparisons using BFs of sizes 128, 192, 256. The selected BF sizes are chosen to be multiples of 64, because the popcount instruction operates on 64 bit values. The false positive rates for the maximum length (14 channels) path are 0.015, 0.0015 and 0.00015 for BFs of sizes 128, 192, and 256, respectively.

According to the error rates in Figure 6.11, the AND/MIN metric provides the best approximation of the ground truth. The popcount of the bitwise XNOR operation on two filters over-estimates the size of the intersection when the two sets have a small number of elements. This is mainly because the BF representations of two small sets have a large number of matching positions with a zero value. This results in an XNORed BF with large numbers of ones. As a result, the third metric performs better when comparing larger paths as can be seen in the plots.

Both the Jaccard and AND/MAX metrics under-estimate the disjointness when the difference in the lengths of the two paths is large. In particular, when comparing a small path P_S with a large path P_L, dividing the size of the intersection by either the size of the larger set or the union set results in a small disjointness metric even when P_S is a subset (i.e., prefix) of P_L. The error curves of both AND/MIN and Jaccard reach their minimum when the path length of the path is the median length

(i.e., 8), because the length difference between the random and the median length path is smaller on average over a million random paths.

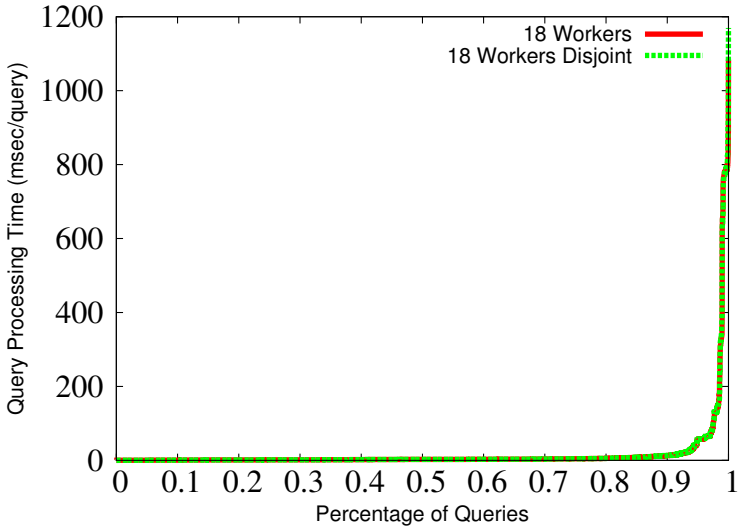

Figure 6.12: Difference in the cumulative distributions of service times for the regular path queries and disjoint path queries.

Based on the above results, we incorporated the AND/MIN approximation to the path service implementation. We used 192 bit Bloom Filters, because the AND/MIN metric is more stable in 192 bit and 256 bit filters, and the performance of the metric on 192 bit and 256 bit filters does not differ significantly. The BF representations of the paths are pre-computed in advance by the path service and stored in the path records. The query processing times of disjoint path queries are negligibly higher than regular path queries as shown in the Figure 6.12. The average service times of the regular path queries differed from the disjoint path queries by 0.25 milliseconds over 100,000 random path queries with disabled path caching. The above result demonstrates that the path service can compute disjoint path queries with very little overhead.

Chapter 7

Deploying Source Routing in the Internet

In this chapter, we describe a (loose) source routing system and its prototype implementation which can be deployed alongside the current Internet routing and forwarding system. An important property of the source routing system is that legacy, unmodified Internet applications can leverage the new features provided by the system such as path selection. The prototype is implemented as part of a larger project called the *ChoiceNet Future Internet Architecture* [71].

The goal of the ChoiceNet project is to develop an "economy plane", where conventional network layer services like routing and forwarding, as well as new network layer services like caching or transcoding can be purchased by users from any provider. In this chapter, we focus on a small portion of the ChoiceNet project involving the network layer services needed to implement a source routing system: namely, transit providers that sell relay (i.e., forwarding) services and path services that sell paths. In particular, the path service purchases relay services (in advance) from transit providers and sells paths containing one or more relay services to the users.

In the current Internet, IP source routing is disabled mainly because of the lack of policy enforcement mechanisms that would enable transit domains to dictate how and which packets enter and leave their network. The source routing system, which we describe in this chapter, makes use of a data plane (i.e., in-band) policy enforcement to

make efficient policy-compliance checks on packets. The in-band policy enforcement mechanism used in our system is similar to that used in Platypus [66]. In particular, packets carry policy-compliance tokens for each relay along the path. The tokens serve as proof-of-purchase for the relay services. The end-systems pay the path service to obtain paths along with the tokens for the relay services along the paths. The details of the payment and the policy enforcement mechanisms are on-going work as part of the ChoiceNet project and is outside the scope of this thesis. In the rest of the chapter, we assume the existence of payment mechanisms whereby the path services pay the transit providers and then turn around and charge end-systems for the paths. The path services provide tokens to the end-systems along with the paths, but we do not provide details on how the tokens are generated or how they are checked by the domains. For additional information about the payment and tokens see [83].

In addition to the path service, the source routing system consists of *forwarding elements* that perform the relay function and policy-compliance checks on the source-routed packets, and a *wrapper library* which end-systems use to run legacy, unmodified Internet applications on the source routing system. The wrapper library can be linked against a legacy application to enable users to express their application-specific performance constraints through a configuration file. The wrapper library includes the performance constraints in the path queries it sends to the path service. The performance constraints can include an upper-bound latency value in milliseconds and a lower-bound bandwidth value in megabits per second.

The system uses Internet Protocol version 6 (IPv6), because IPv6 allows variable-length source-routing headers to be carried as extension headers. The source-routing header contains the path and the policy-compliance tokens. As described in Section 2.1, IPv6 defines several types of extension headers for the purposes of carrying optional network layer information. Our packets use *destination options*, whose original purpose is to carry optional information which needs to be examined only by the

packet's destination end-point; therefore, they remain untouched by the legacy Internet routers and middle-boxes. The use of destination options header for our purposes does not interfere with the use of that header by the existing IPv6 protocol, because we use a new option type within the destination options header which is currently not allocated to any existing IPv6 services.

As a consequence of using the IPv6 protocol to deliver source routing packets, the system uses IPv6 addresses to name channels and end-systems. In our system, a channel is a unidirectional, point-to-point, best-effort transmission facility identified by a pair of interface (IPv6) addresses, the interface address of the channel's origin (egress interface of a node) followed by the interface address of the channel's destination (i.e., ingress interface of a node). The end-systems are identified by any one of the IPv6 addresses of their interfaces. The reason for naming channels using both the origin and destination endpoint IPv6 addresses is to enable end-systems to construct return paths by directly reversing the (forward) paths carried in the incoming packets in order to send return traffic. In particular, a receiver constructs a return path by reversing the order of channel identifiers and the order of the IPv6 addresses (i.e., origin and destination) in each channel identifier. Our system uses "bi-directional" policy-compliance tokens that are usable in both directions of a path by the end-systems; therefore, a receiver can use the tokens in the incoming packets to send policy-compliant packets back to the sender. In contrast, the deprecated IP source routing, explained in Section 4.1.1, uses a reverse route recording mechanism where routers on the forward path overwrite the path information in the packets with the reverse path for the receivers to use in the return traffic. Because our in-band policy enforcement mechanism requires the path information in the packets to be unmodified for the receivers to be able to construct policy-compliant packets for their return traffic, our forwarding elements do not use a reverse route recording mechanism. However, our approach is similar to the IPv4 loose source routing in that the

processing of packets at the forwarding elements involve modifying the destination address of the packets. In particular, the forwarding elements replace the destination address of the packets with the address of the next hop on the path. Consequently, the routing of packets between consecutive hops in a loose source route can be performed by the hop-by-hop routing mechanism similar to IP loose source routing option.

For reasons of simplicity of bootstrapping and to enable partial deployment (explained later), the source routing system currently uses IPv4 routing to deliver control traffic, but we plan to remove this requirement in the future. The control traffic in our system comprises the relay advertisements from transit providers to the path services, queries from the end-systems to the path services, and query responses from the path services to the end-systems. The relay advertisements contain the ingress and egress channels of the transit domains and the performance information. We also make the assumption that the end-systems can discover the locations (i.e., IPv4 addresses) of available path services through out-of-band means.

In Section 7.1, we describe the prototype implementation of the system. Then, we describe our experiments with the implementation in Section 7.2. Finally, we discuss a potential scenario for real-world deployment of the system in Section 7.3.

7.1 Prototype Implementation and Experiments on GENI

We implemented a prototype that demonstrates the capabilities of our source routing system. Our prototype demonstrates that legacy, unmodified Internet applications can utilize the multi-path capabilities of our system by using a *wrapper library* in Section 7.1.1. Also, we present a software router implementation of a forwarding element in Section 7.1.2.

7.1.1 The Wrapper Library

One benefit of a source routing system for applications is to leverage the multi-path routing features. However, legacy, unmodified Internet applications are not designed to use the features of our source routing system. In order to use legacy, unmodified applications in our system, we implemented a preloaded shared library called *wrapper*. The wrapper library replaces standard network I/O function calls such as connect() and send() with functions that are able to leverage the multi-path capabilities of our path service. The wrapper can be linked against a wide range of existing, legacy Internet applications including things such as ssh, browsers, iperf, etc. We refer to an application which is linked against the wrapper library as a *wrapped* application.

The wrapper library initially reads a user-provided configuration file to obtain the performance constraints of the (wrapped) application and the IPv4 address of the path service. At some point a standard TCP client application will make a connect() call to connect a socket file descriptor to a destination IPv6 address, both specified as arguments. The connect() call is intercepted by the wrapper_connect() call implementation in the wrapper. The pseudo code of the wrapper_connect() is shown in Listing 7.1. The wrapper_connect() call first checks if the file descriptor in the arguments is an IPv6 socket. If so, the wrapper sends a path query request to the path service. The wrapper includes the performance constraints read from the configuration file along with the local machine's and the destination machine's IPv6 addresses in the path query. The path service returns paths (in the code shown here, only one path is returned) and policy-compliance tokens (for transit domains to authenticate packets) to the wrapper. The wrapper inserts the path and the tokens returned by the path service into an IPv6 extension header (Line 15) and makes a setsockopt() system call (Line 18) to attach the extension header to all future outgoing IPv6 packets that will be sent from the socket. The setsockopt() call is given the argument IPV6_DSTOPTS which indicates a destination options header to

be used within the extension header.

```
1  int wrapper_connect (int sockfd,const struct sockaddr
2                                  *dst_addr, socklen_t len)
3  {
4    unsigned char *ext_hdr;
5    unsigned char del_tokens[], final_tokens[];
6    struct sockaddr *next_hop_addr = NULL;
7    ... [other declarations]
8
9    if(is_ipv6(sockfd)){
10     //Obtain paths and proof-of-compliance tokens
11     //from the path service.
12     Path path = request_path(PathService_addr,src_addr,
13                        dst_addr, constraints[], tokens[]);
14
15     ext_hdr = build_ext_header(path, tokens);
16     next_hop_addr = get_next_hop(path);
17
18     if(setsockopt(sockfd,IPPROTO_IPV6,IPV6_DSTOPTS,
19                                        ext_hdr,extlen) < 0){
20       die("setsockopt␣IPV6_DSTOPTS␣error!");
21     }
22   }
23   if(next_hop_addr){
24     rc = connect(sockfd, next_hop_addr, len);
25   }
26   else
27     rc = connect(sockfd, dst_addr, len);
28
29   return rc;
30 }
```

Listing 7.1: Psuedo code for the wrapper's modified connect() call.

Finally, the wrapper_connect() function makes a call to the real connect I/O system call. The destination address (i.e., second argument) passed to the real connect call (Line 24) is the address of the next forwarding element that the packet is to visit. The next hop address is extracted from the path, which is returned by the path service and saved in the next_hop_addr variable (Line 16). The original destination of the IPv6 packet is stored as part of the path carried in the extension header. The forwarding elements on the path eventually restore the original destination of the

packets as explained in Section 7.1.2. If the wrapper intercepts a connect call from an IPv4 socket, the destination address is not modified and passed as it is to the real connect call (Line 27).

After the connect call returns, the wrapper is not called (directly) until the socket is closed, at which point the wrapper deletes all the state allocated for the socket. All the client application's packets sent over the socket, including the handshake (i.e., SYN and ACK) packets, are sent with extension headers by the operating system, because the IPV6_DSTOPTS has been set for the socket. In the extension header, the wrapper stores multiple source routing related fields including the path information, a next-hop field, and the proof-of-compliance tokens. The path information contains a sequence of channel identifiers where each consecutive channel pair along the path is a relay. The first and the last identifiers are different from the other channels identifiers. In particular, the first channel only contains an origin interface identifier corresponding to the IPv6 address of the source end-system, and the last channel only contains a destination interface identifier corresponding to the IPv6 address of the destination end-system. The next-hop field points to the next channel that the packet is to visit next. The forwarding elements increment the next-hop field upon processing the packet.

At the receiving end, a wrapped server application creates a socket and waits for incoming packets. In the case of a TCP server, the wrapped application makes a listen() call before the three-way handshake, which involves responding to an incoming SYN packet with a SYN-ACK packet. A challenge with wrapping the receiving-side TCP application is the SYN ACK packet. In particular, the network layer application programming interface (i.e., socket interface) does not enable access to the headers of the incoming SYN packets. Without access to the incoming SYN packets, the operating system will treat them like normal IPv6 packets, creating a SYN-ACK packet destined for the sender, but not carrying a (reversed) source route in the IPv6

options field. Because the receiving-side wrapped TCP application did not see the SYN packet, it cannot construct a SYN-ACK packet with a return path based on the path carried in the SYN packet. To address this problem, the wrapper uses the libpcap library [4] to obtain a copy of all incoming IPv6 SYN packets carrying destination options extension headers. The wrapper reverses the path carried in the SYN packet and forms a SYN ACK packet (using raw sockets) with the reversed path and the (bi-directional) tokens carried in the SYN packet.

For UDP applications, the client side wrapper performs the path requests in the first I/O call that tries to send data through a UDP socket. At the receiving (i.e., server) side, the wrapper makes a setsockopt() call with IPV6_RECVDSTOPTS parameter to receive extension headers containing destination options along with the data portion of the packets. After the wrapper obtains the extension header of an incoming packet, it reverses the path and extracts the tokens in the incoming packets to form outgoing packets.

7.1.2 Forwarding Elements

The forwarding element is implemented using the Click modular router [49] on a Linux host. The forwarding element intercepts only the IPv6 packets with extension headers containing destination options. It forwards legacy Internet IPv4 packets as normal so that control traffic can be delivered.

Upon the arrival of a source-routed IPv6 packet, the forwarding element first checks the tokens in the packet to confirm that the packet is policy-compliant. If the token is valid, the forwarding element copies the egress IPv6 address of the next channel on the path to the IPv6 destination address field in the main IPv6 header. Once the address is modified, the forwarding element hands the packet to the Linux kernel to have it forwarded through the appropriate outgoing interface.

Another task of the forwarding element is to send *relay advertisements* to the

path service to report the performance across its relays—each ingress, egress channel combinations of the node. In our experiments, we used only slow-changing routing information including the propagation delay and the bandwidth capacity.

In the next section, we present our experiments with the system.

7.2 Experiments

We tested our prototype implementation consisting of the wrapper library, the forwarding element, and the path service, using the Global Environment for Network Innovations (GENI) experimental network [63]. GENI is a virtualized network consisting of end-hosts and networking fabric that allows multiple researchers to simultaneously run their experiments.

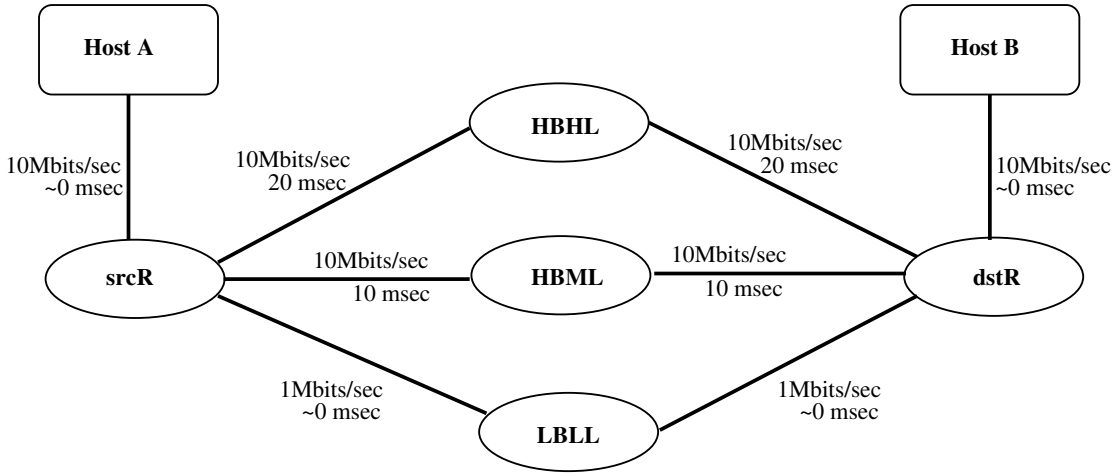

Figure 7.1: Topology used in the GENI experiments. The topology has 3 paths between the hosts, each having different qualities: 1) high bandwidth and high latency (*HBHL*), 2) high bandwidth and medium latency (*HBML*), and 3) low bandwidth and low latency (*LBLL*).

We used the sample topology given in Figure 7.1 for our experiments. The topology has 7 nodes with 2 hosts (named *hostA* and *hostB*) and 5 routers (named *srcR*, *HBHL*, *LBLL*, *HBML*, *dstR*). All the nodes in the experiments are Xen virtual machines. The router nodes run the forwarding element described earlier, and the hosts

run wrapped legacy Internet applications such as iperf, wget, ssh, and scp. One of the hosts also runs the path service implementation with a single worker thread to compute query results and a single updater to process relay advertisements. Given the small size of the topology, a single worker sufficed to compute paths.

The three (loop-free) paths in the topology between the two hosts traverse the three routers *HBHL* (high bandwidth, high latency), *HBML* (high bandwidth, medium latency), *LBLL* (low bandwidth, low latency) that are adjacent to channels with different propagation delays and bandwidth capacities. The high bandwidth routers are adjacent to 10 Mbits/sec channels, while the low bandwidth routers are adjacent to 1 Mbits/sec channels. The high latency routers are adjacent to channels with approximately 20 msec propagation delay, medium latency routers are adjacent to channels with approximately 10 msec delay (see Figure 7.1). The rest of the channels in the topology have close to 0 msec propagation delay and 10 Mbits/sec bandwidth.

The wrapper library is configured with the latency (in msecs) and throughput (in Mbits/sec) of the desired path via a configuration file. The configured constraints on latency and bandwidth are sent to the path service as part of the path queries. The path service returns the path whose propagation delay and bandwidth capacity attributes are "closest" to the constraints. By returning the closest (i.e., best fit) rather than the best path, we make sure that all the paths (not just the best) are utilized. To obtain the closest path, the path service computes an aggregate fit metric for each path. In particular, the service computes the absolute value of the differences between the constraints and the attributes in the two dimensions of latency and bandwidth. Then, a nonlinear combination, i.e., maximum, of the two differences is computed to produce a single fit metric. The path service sorts the paths according to their fit metrics and the path with the smallest metric is returned to the wrapper.

By entering different performance constraints in the configuration file, users or their applications are able to obtain different paths between the hosts. For example,

by entering a 10 Mbits/sec upper-bound on the bandwidth and a 20 msec lower-bound on the latency in the configuration file, the user obtains the high bandwidth, medium latency (*HBML*) path. We used a wrapped iperf program to measure the throughtput on the paths between the hosts. Because of the overheads involved in copying addresses in the packets and verifying tokens, the throughput achieved by the wrapped applications are lower than the capacity. In particular, the wrapped iperf reported 5.6 Mbits/sec of throughput on the high bandwidth paths (*HBHL* and *HBML*), which had a capacity of 10Mbps and no cross traffic..

7.3 Deploying the Source Routing System in the Internet

In this section, we consider the deployment of our source routing system alongside the current Internet. The deployment of our system in the Internet means that domains forward source-routed IPv6 packets and send relay advertisements to the path service. We assume that the system is globally deployed in all domains and explain how the source routing system would work in that scenario. Then, we consider a more realistic scenario, where only a subset of the domains forward source-routed packets and send relay advertisements (i.e., partial deployment).

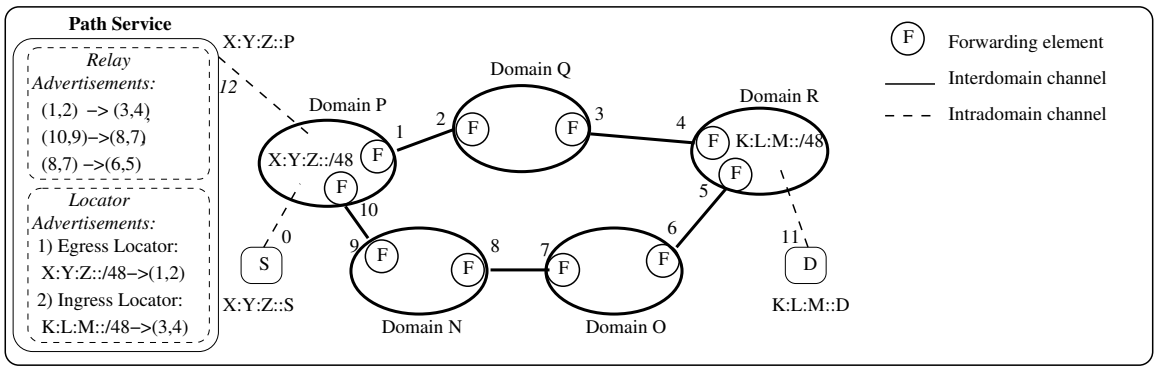

Figure 7.2: A simple scenario for a globally deployed source routing system.

Consider the simple topology in Figure 7.2, where 5 domains are inter-connected.

The endpoints of the channels are numbered from 0 to 12. Each numbered interface is associated with a globally routable IPv6 address[1]. We assume that the border nodes in the domains are capable of routing legacy Internet packets (IPv4) and processing source-routed IPv6 packets similar to the forwarding element. In the figure, end-system S is located within domain P and end-system D is located within domain R. Both end-systems are assigned IPv6 addresses from the IPv6 prefixes assigned to their local domain. IPv6 address prefixes X:Y:Z::/48 and K:L:M::/48 are assigned to the domains P and R, respectively[2]. The path service is shown in Figure 7.2 as located within domain P. The notations for the channel identifiers in the example use the numbers corresponding to the origin and destination interfaces surrounded by brackets, as in (1,2) for the channel connecting the domains P and Q. We assume that each transit domain periodically sends relay advertisements to the path services with slow-changing and fast-changing information.

In order for the path service to compute inter-domain paths between two end-systems, some means is necessary to map end-system IPv6 addresses (EIDs) to the ingress and the egress channels of their access domains. We refer to the ingress and egress channels corresponding to an EID as the *ingress locators* and *egress locators*, respectively. In order to map IPv6 addresses to ingress and egress locators, the path service collects *locator advertisements* from the domains in addition to the relay advertisement. In particular, an egress locator advertisement maps a source IPv6 prefix to a set of egress channels, while an ingress locator advertisement maps a destination IPv6 prefix to a set of ingress channels. In order to map a given destination (source) IPv6 address to an ingress (egress) locators, the path service finds a longest prefix match of the destination (source) address among all the prefixes obtained from the ingress (egress) locator advertisements. Once the matching prefix is found, the

[1]We assume that the IPv6 protocol is globally deployed.

[2]In the IPv6 address notations, we use an upper-case letter instead of each group of four hexadecimal digits for brevity.

path service extracts the ingress channels from the ingress locator advertisement of the prefix. The path service collects locator advertisements periodically, upon changes in the set of locators of prefixes. We expect such changes to be infrequent.

Some of the relay and locator advertisements that are collected by the path service from the transit providers are shown in Figure 7.2. The relay advertisements are shown with only the connectivity information. In the figure, we use an arrow between an ingress channel and an egress channel to represent the connectivity of a relay. To obtain paths to the end-system D, S sends a query to the path service providing the IPv6 address of D (K:L:M::D), a set of performance constraints, number of requested paths and its own IPv6 address (X:Y:Z::S). The path service first obtains the egress locators of S and the ingress locators of D using the locator advertisements. In Figure 7.2, the longest prefix match of S's address (i.e., X:Y:Z::/48) maps to the channel (1,2), and the longest prefix match of D's address (i.e., K:L:M::/48) maps to (3,4) in the locator advertisements. The path service computes the paths from the egress locator and ingress locator (in this case there is only one) and returns it to S.

In the case of global deployment, the source-routed packets are only processed by the ingress forwarding elements of the domains. The processing of a source routed packet involves incrementing the next-hop field, copying the destination interface address of the next channel on the path to the destination of the packet, and verifying the tokens as explained before. The forwarding of the packets across the domains (i.e., between ingress and egress) can be performed by hop-by-hop routing, MPLS or any other intra-domain routing approach.

In Figure 7.2, the path from S to D is P_1: (0,)(1,2)(3,4)(,11). The destination interface address of the first channel and the origin interface address of the last channel are empty, as they represent the source and destination end-system identifiers, respectively. Below we describe how a packet with path P_1 is processed on its way to D. The wrapper library running in S sets the initial destination of the outgoing

packets to the destination interface address of the second channel of the path (i.e., interface 2). Once the packet arrives at the interface 2 of domain Q, the ingress forwarding element sets the new destination of the packets to the destination interface IPv6 address of the third channel (i.e., interface 4). Similarly, once the packet arrives to the ingress forwarding element of the domain R, the new destination of the packet is set to the destination interface IPv6 address of the final channel (i.e., interface 11). Finally, the packet is routed to the destination D in the domain R.

New routing and forwarding systems that require global deployment have a high barrier to deployment in the Internet, because there is no gain for the provider who deploys, unless the system is simultaneously deployed by all the providers in the Internet. On the other hand, an incrementally deployable routing system that can provide benefits to end-systems and providers even with partial deployment has comparably lower barrier to deployment.

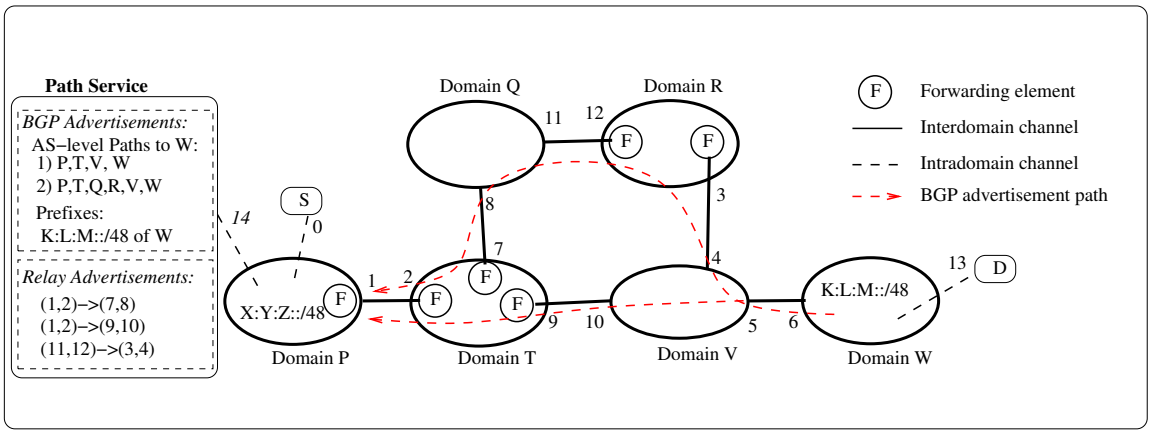

Figure 7.3: A simple scenario for a partially deployed source routing system.

Because our source routing system is compatible with the existing Internet routing and forwarding system, it is possible for end-systems to leverage multi-path capabilities even when only a few transit providers deploy the source routing system. End-systems accrue more benefit from the source routing system as more providers deploy the system, because the path choices increases with increasing number of deployment.

Consider the example shown in Figure 7.3, where only three of the six domains (P, R, and T) participate in the source routing system. In this example, the domains P and W are access domains, and the rest are transit domains. The domains R and T periodically send relay advertisements to the path service located in the domain P, with slow-changing and fast-changing information. The partial deployment limits the topology information known by the path service, since only the participating domains advertise routing information. In particular, the path service is unable to even compose a global map of the AS-level connectivity without complete set of slow-changing relay information from all the domains. Also, the path service is unable to map destination addresses of end-systems within non-participating access domains to ingress or egress locators due to lack of locator advertisements from non-participating domains.

A possible workaround is to utilize BGP advertisements. In particular, the BGP advertisements contain AS-level paths to prefixes that can be used to build an inter-domain connectivity map and also to map prefixes to ASs. We assume that the path service has access to BGP advertisements that are received by a subset of the participating domains. In Figure 7.3, the path service obtains the BGP advertisements received by the domain P. Domain P receives two BGP advertisements for the prefix K:L:M::/48 over two distinct paths as shown in the figure with dashed lines with arrows. From the two advertisements, the path service infers two AS-level paths to domain W and associates the prefix K:L:M::/48 with the domain W. More specifically, the path service learns the two AS-level paths to Q: i) P,T,V,W and ii) P,T,Q,R,V,W. By using the complete BGP advertisements from all the participating domains, the path service could form a more complete map of the AS-level connectivity of the Internet.

Even though the path service has a connectivity map of the topology, the performance (i.e., fast-changing) information from non-participating domains are missing.

For example, the performance information along the relay: $(7, 8) \rightarrow (11, 12)$ of the non-participating domain Q is not known to the path service. The path service only knows that the domain T and R are connected through Q from the BGP advertisements. However, even with an incomplete set of routing information, the path service can still compute (possibly useful) end-to-end paths between the end-systems. One possible strategy is to pick the shortest paths with the most participating domains—that is, the paths with the most known performance information. The end-systems can benefit from the partial deployment, because they can possibly obtain multiple paths to destinations. Also, the providers can benefit from deployment by charging users for source routing their traffic.

In the partial deployment scenario, the processing of source-routed packets by the participating domains differ from the full-deployment scenario. In particular, the packets are processed by both the ingress and egress forwarding elements of the participating domains. For example, consider the following path P_2 from S to D in Figure 7.2: (0,)(1,2)(9,10)(,13). If the source-routed packet with the path P_2 were to be processed only by the ingress forwarding elements of the participating domains, then the packet would be eventually forwarded from domain T to domain V with a destination of 10. Because domain V ignores the source routes on the packets, the transmission of the packet with the path P_2 would terminate at the ingress forwarding element (with interface 10) of V and therefore, the packet would never reach the destination D. In order to route packets across non-participating domains, the participating domains must process packets at both the ingress and the egress forwarding elements. More specifically, instead of forwarding the packets with path P_2 to domain V with a destination address of 10, the egress forwarding element of T (with interface 9) modifies the destination of the packet to be the destination interface of the next channel on the path P_2, which is 13. However, the processing of source-routed packets at the ingress forwarding elements of the domains is only

necessary if the next hop domain of the ingress node is a non-participating domain.

In this chapter, we described our implementation of a source routing system and experiments on GENI. We also considered ways to deploy the system in the current Internet and briefly described a scenario where the system is partially deployed. A more detailed investigation of how the path service can compute paths under the partial deployment scenario is left for future work.

Chapter 8

Conclusion and Future Work

In this thesis, we have introduced a design for a scalable path service which can compute application-specific paths for end systems. End-systems send path queries to the path service to request inter-domain paths to destinations with one or more performance constraints. The path service considers the time-scales of change in the routing information and distinguishes between the slow-changing connectivity and the fast-changing performance information of paths. In particular, the path service performs a pre-computation step to generate a large subset of all possible paths. The set of pre-computed paths is updated infrequently with permanent changes in the connectivity of the domains. On the other hand, the path service computes the performance information of paths frequently as an on-demand computation. Performing the path generation separately from the computation of performance information leads to new opportunities. More specifically, the path service can parallelize the construction of paths and the computation of path performance information. The path service also takes advantage of caching of recently computed query results.

We have evaluated the performance of a path service implementation on a single server with 24 cores. Because a single server is insufficient to handle a realistic workload, we have also considered a distributed path service with multiple identical copies of the single server implementation working in parallel. In the evaluation of the distributed path service, we have used traffic traces collected in a campus network to

generate a realistic workload for the service. We have shown that with only 10 path service instances working in parallel, the path service can serve all the end-systems in a single access domain. We have also extended the path service to provide maximally disjoint paths and have shown that the path service can compute maximally disjoint paths with very little additional cost.

We have implemented a simple source routing system consisting of a wrapper library and a forwarding element. The wrapper library is used by the end-systems to leverage the path selection capability of the source routing system with legacy, unmodified IPv6-capable Internet applications. In particular, the wrapper library can link against (i.e., wraps) IPv6 applications to obtain paths from the path service. Once it obtains paths from the path service, the wrapper inserts source routes in the outgoing IPv6 packets of the wrapped application within the extension headers. The forwarding element is a software router which forwards the source-routed IPv6 packets generated by the wrapped IPv6 applications. Finally, we have considered a deployment scenario for our source routing system alongside the current Internet architecture. The deployment of the system by a domain requires the domain to forward source-routed IPv6 packets and periodically send relay and locator advertisements to the path service. We have shown that the system can work with only a subset of all the domains participating in the source routing system. In particular, under partial deployment, the path service uses BGP advertisements as slow-changing routing information to form a topology map and uses an incomplete set of performance information to compute paths.

8.1 Future Work

In the future, we hope to extend our work by deploying a distributed path service on a cluster of computers to verify the results that we obtained from simulating the distributed path service. In our simulation of the distributed system, we assumed

multiple instances of the single server implementation running in parallel. Each server in the simulation had identical copies of the entire set of pre-computed paths in order to compute the results to queries. A possible extension would be to parallelize the computation of queries across multiple servers. In particular, each server in the cluster would obtain a proper set of the paths instead of all the pre-computed paths. This extension would allow testing of the path service with larger sets of pre-computed paths than we used in our experiments by distributing the set of paths across many servers.

Another possible direction is to investigate path computation and performance measurement strategies in the partial deployment scenario where a subset of the domains in the Internet forward source-routed packets and report performance information. Providing paths in a partial deployment scenario is challenging due to incomplete slow-changing and fast-changing routing information. A possible strategy for the path service is to collect measurements across default (i.e., BGP) paths between pairs of distant participating domains. However, performing measurements between all pairs of participating domains is unlikely to scale; therefore, the system must choose a scalable strategy to measure the performance across a subset of default paths.

In our experiments, we assumed that the Internet inter-domain topology with source routing system is similar to the current Internet. In future work, we hope to understand the implications of deployment of source routing in the inter-domain topology of the Internet. With significant changes in routing and economics, we expect differences in the structure of the domain-level topology to emerge as a consequence of the deployment of source routing. In order to understand the potential effects of the architectural changes in the structure of the inter-domain topology, a possible approach is to use an agent-based model to simulate the behaviors of transit providers and users, similar to the approach by Dhamdhere et al. in [6].

Bibliography

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

[2] Open MPI: A High Performance Message Passing Library. http://www.open-mpi.org/.

[3] The CAIDA UCSD Routeviews Prefix to AS Mappings for IPv4 and IPv6, June-August 2013. http://www.caida.org/data/routing/routeviews-prefix2as.xml.

[4] The Libpcap Project. http://www.tcpdump.org/.

[5] IEEE 802.2 International Standard for Information Technology Telecommunications and Information Exchange Between Systems - Local Area Networks - Media Access Control (MAC) Bridges. http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6887401, 1993.

[6] C. Dovrolis A. Dhamdhere. The Internet is Flat: Modeling the Transition from a Transit Hierarchy to a Peering Mesh. In *Proceedings of ACM CoNTEXT*, 2010.

[7] J. Abley, P. Savola, and G. Neville-Neil. Deprecation of Type 0 Routing Headers in Internet Protocol Version 6, December 2007. RFC 5095.

[8] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. *Resilient Overlay Networks*, volume 35. ACM, 2001.

[9] David G Andersen, Alex C Snoeren, and Hari Balakrishnan. Best-path vs. Multi-path Overlay Routing. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pages 91–100. ACM, 2003.

[10] Onur Ascigil, Song Yuan, Jim Griffioen, and Ken Calvert. Deconstructing the Network Layer. In *Proceedings of 17th IEEE International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6, 2008.

[11] ATM Forum Technical Committee and Others. Private Network-Network Interface Specification Version 1.0, 1996.

[12] D. Awduche, A. Chiu, A. Elwalid, I. Widjaja, and X. Xiao. Overview and Principles of Internet Traffic Engineering, May 2002. RFC 3272.

[13] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. Web Search for a Planet: The Google Cluster Architecture. *IEEE Micro*, 23(2):22–28, 2003.

[14] Bobby Bhattacharjee, Ken Calvert, Jim Griffioen, Neil Spring, and James PG Sterbenz. Postmodern Internetwork Architecture (PoMo). Technical report, 2006. NSF-FIND Proposal, ITTC Technical Report ITTC-FY2006-TR-45030-01.

[15] Burton H Bloom. Space/time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[16] John W Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed Content Delivery Across Adaptive Overlay Networks. *IEEE/ACM Transactions on Networking (TON)*, 12(5):767–780, 2004.

[17] Kenneth L Calvert, Jim Griffioen, and Leonid Poutievski. Separating Routing and Forwarding: A Clean-slate Network layer Design. In *4th International Conference on Broadband Communications, Networks and Systems, BROADNETS*, pages 261–270. IEEE, 2007.

[18] I. Casteneyra, N. Chiappa, and M. Steenstrup. The Nimrod Routing Architecture. RFC 1992, August 1996.

[19] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. On Inferring AS-level Connectivity from BGP Routing Tables. Technical report, Citeseer, 2002.

[20] Shigang Chen and Klara Nahrstedt. On Finding Multi-constrained Paths. In *IEEE International Conference on Communications ICC*, volume 2, pages 874–879, 1998.

[21] J. Noel Chiappa. Technical Presentation: Map-Distribution/Explicit Routing Architectures for QoS Routing and Traffic Engineering. `http://mercury.lcs.mit.edu/~jnc/tech/routing_slides.html`. [Online; accessed 10-November-2014].

[22] D. Clark, J. Wroclawski, K. Sollins, and R. Braden. Tussle in Cyberspace: Defining Tomorrow's Internet (position paper). In *Proceedings of ACM SIGCOMM 2002, Pittsburgh, USA*, pages 347–356, August 2002.

[23] Hans De Neve and Piet Van Mieghem. TAMCRA: a Tunable Accuracy Multiple Constraints Routing Algorithm. *Computer Communications*, 23(7):667–679, 2000.

[24] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[25] S. Deering and R. Hinden. Internet Protocol Version 6 Specification, December 1998. RFC 2460.

[26] C. Demichelis. IP Packet Delay Variation Metric for IP Performance Metrics (IPPM), November 2002. RFC 3393.

[27] Amogh Dhamdhere and Constantine Dovrolis. Twelve Years in the Evolution of the Internet Ecosystem. *IEEE/ACM Transactions on Networking (ToN)*, 19(5):1420–1433, 2011.

[28] Edsger W Dijkstra. A Note on two Problems in Connexion with Graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[29] R. Droms. Dynamic Host Configuration Protocol, March 1997. RFC 2131.

[30] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained Network Time Synchronization using Reference Broadcasts. *ACM SIGOPS Operating Systems Review*, 36(SI):147–163, 2002.

[31] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN. *Queue*, 11(12):20, 2013.

[32] Internet Engineering Task Force. Locator/ID Separation Protocol (LISP) IETF Working Group. `https://datatracker.ietf.org/wg/lisp/`, 2009.

[33] Internet Engineering Task Force. Multi-path Transmission Control Protocol (MPTCP) IETF Working Group. `https://datatracker.ietf.org/mptcp`, 2014.

[34] Howard Frank, Robert E Kahn, and Leonard Kleinrock. Computer Communication Network Design: Experience with Theory and Practice. In *Proceedings of the Spring joint Computer Conference*, pages 255–270. ACM, 1972.

[35] Lixin Gao. On Inferring Autonomous System Relationships in the Internet. *IEEE/ACM Transactions on Networking (ToN)*, 9(6):733–745, 2001.

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

[37] F. Gont. Security Assessment of the Internet Protocol, July 2011. RFC 6274.

[38] James Griffioen, Kenneth L Calvert, Onur Ascigil, and Song Yuan. Separating Routing Policy from Mechanism in the Network Layer. *Next-Generation Internet: Architectures and Protocols*, pages 219–232, 2011.

[39] Arpit Gupta, Muhammad Shahbaz, Laurent Vanbever, Sean Donovan, Russ Clark, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, and Ethan Katz-Bassett. SDX: A Software Defined Internet Exchange. In *Proceedings ACM SIGCOMM 2014, Chicago, USA*.

[40] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm, November 2000. RFC 2992.

[41] Ningning Hu and Peter Steenkiste. Evaluation and Characterization of Available Bandwidth Probing Techniques. *IEEE Journal on Selected Areas in Communications*, 21(6):879–894, 2003.

[42] Information Sciences Institute, University of Southern California. Internet Protocol Specification, September 1981. RFC 791.

[43] Cisco Internetwork Operating System (IOS). Netflow, 2008.

[44] Jeffrey M Jaffe. Algorithms for Finding Paths with Multiple Constraints. *Networks*, 14(1):95–116, 1984.

[45] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a Globally-deployed Software Defined WAN. In *Proceedings of the ACM SIGCOMM*, pages 3–14. ACM, 2013.

[46] S. Kawamura. A Recommendation for IPv6 Address Text Represenation, August 2010. RFC 5952.

[47] Frank Kelly and Thomas Voice. Stability of End-to-end Algorithms for Joint Routing and Rate Control. *ACM SIGCOMM Computer Communication Review*, 35(2):5–12, 2005.

[48] Atul Khanna and John Zinky. The Revised ARPANET Routing Metric. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 45–56. ACM, 1989.

[49] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[50] Turgay Korkmaz and Marwan Krunz. A Randomized Algorithm for Finding a Path Subject to Multiple QoS Requirements. *Computer Networks*, 36(2):251–268, 2001.

[51] Nate Kushman, Srikanth Kandula, and Dina Katabi. Can you hear me now?!: It must be BGP. *ACM SIGCOMM Computer Communication Review*, 37(2):75–84, 2007.

[52] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed Internet Routing Convergence. *ACM SIGCOMM Computer Communication Review*, 30(4):175–187, 2000.

[53] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed Internet Routing Convergence. *IEEE/ACM Transactions on Networking (TON)*, 9(3):293–306, 2001.

[54] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet Inter-domain Traffic. *ACM SIGCOMM Computer Communication Review*, 41(4):75–86, 2011.

[55] Karthik Kalambur Lakshminarayanan, Ion Stoica, Scott Shenker, and Jennifer Rexford. Routing as a service. Technical Report UCB/EECS-2006-19, EECS Department, University of California, Berkeley, Feb 2006.

[56] Yong Liu, Honggang Zhang, Weibo Gong, and Don Towsley. On the Interaction Between Overlay Routing and Underlay Routing. In *24th Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM*, volume 4, pages 2543–2553. IEEE, 2005.

[57] Hongbin Luo, Yajuan Qin, and Hongke Zhang. A DHT-based Identifier-to-locator Mapping Approach for a Scalable Internet. *IEEE Transactions on Parallel and Distributed Systems*, 20(12):1790–1802, 2009.

[58] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[59] Michael Mitzenmacher. The Power of Two Choices in Randomized Load Balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10):1094–1104, 2001.

[60] P. Mockapetris. Domain Names - Concepts and Facilities, November 1987. RFC 1034.

[61] 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.

[62] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers, December 1998. RFC 2474.

[63] G. P. Office. Global Environment for Network Innovations - System Overview. `http://www.cra.org/ccc/files/docs/GENISysOvrvw092908.pdf`, 2008.

[64] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. 1999.

[65] Dan Pei, Lan Wang, Daniel Massey, S Felix Wu, and Lixia Zhang. A Study of Packet Delivery Performance During Routing Convergence. In *Proceedings of IEEE International Conference on Dependable Systems and Networks*, 2003.

[66] Barath Raghavan and Alex C Snoeren. A System for Authenticated Policy-compliant Routing. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 167–178. ACM, 2004.

[67] A. Reitzel. Deprecation of Source Routing Options in IPv4, August 2007. RFC 791 update.

[68] Y. Rekther and S. Hares T. Li. A Border Gateway Protocol 4 (BGP-4), January 2006. RFC 4271.

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

[70] Matthew Roughan, Subhabrata Sen, Oliver Spatscheck, and Nick Duffield. Class-of-service Mapping for QoS: a Statistical Signature-based Approach to IP Traffic Classification. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 135–148. ACM, 2004.

[71] 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.

[72] Jacob Strauss, Dina Katabi, and Frans Kaashoek. A Measurement Study of Available Bandwidth Estimation Tools. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, pages 39–44. ACM, 2003.

[73] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy H Katz. OverQoS: Offering Internet QoS Using Overlays. *ACM SIGCOMM Computer Communication Review*, 33(1):11–16, 2003.

[74] P. Van Mieghem and F. A. Kuipers. On the Complexity of QoS Routing. *Computer Commununications*, 26(4):376–387, 2003.

[75] Route Views. University of Oregon Route Views Project, 2000.

[76] Tam Vu, Akash Baid, Yanyong Zhang, Thu D Nguyen, Junichiro Fukuyama, Richard P Martin, and Dipankar Raychaudhuri. DMAP: A Shared Hosting Scheme for Dynamic Identifier to Locator Mappings in the Global Internet. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 698–707. IEEE, 2012.

[77] Feng Wang and Lixin Gao. Path Diversity Aware Interdomain Routing. In *IEEE INFOCOM*, pages 307–315, 2009.

[78] Feng Wang, Zhuoqing Morley Mao, Jia Wang, Lixin Gao, and Randy Bush. A Measurement Study on the Impact of Routing Events on End-to-end Internet Path Performance. *ACM SIGCOMM Computer Communication Review*, 36(4):375–386, 2006.

[79] Zheng Wang and Jon Crowcroft. Quality-of-service Routing for Supporting Multimedia Applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1228–1234, 1996.

[80] Tom White. *Hadoop: the Definitive Guide*. O'Reilly Media, Inc., 2009.

[81] Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun. The Resource Pooling Principle. *ACM SIGCOMM Computer Communication Review*, 38(5):47–52, 2008.

[82] Damon Wischik, Costin Raiciu, Adam Greenhalgh, and Mark Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *NSDI*, volume 11, pages 8–21, 2011.

[83] T. Wolf, A. Nagurney, J. Griffioen, K. Calvert, G. Rouskas, I. Baldine, and R. Dutta. ChoiceNet Project. `www.ecs.umass.edu/ece/wolf/ChoiceNet/`.

[84] Xiongqi Wu. *A Network Path Advising Service*. PhD thesis, University of Kentucky, May 2015.

[85] Xiongqi Wu and James Griffioen. Network Path Advising Service for the Future Internet. In *4th IEEE/IFIP International Workshop on Management of the Future Internet in conjunction with IEEE Network Operations and Management Symposium (NOMS)*, pages 1127–1134, 2012.

[86] Xiongqi Wu and James Griffioen. Supporting Application-based Route Selection. In *International Conference on Computer Communication and Networks (ICCCN)*. IEEE, 2014.

[87] Li Xiao, Jun Wang, King-Shan Lui, and Klara Nahrstedt. Advertising Interdomain QoS Routing Information. *IEEE Journal on Selected Areas in Communications*, 22(10):1949–1964, 2004.

[88] Xiaowei Yang, David Clark, and Arthur W Berger. NIRA: a New Inter-domain Routing Architecture. *IEEE/ACM Transactions on Networking*, 15(4):775–788, 2007.

[89] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G Andersen. SCION: Scalability, Control, and Isolation on Next-generation Networks. In *IEEE Symposium on Security and Privacy*, pages 212–227, 2011.

[90] Yong Zhu, Constantinos Dovrolis, and Mostafa Ammar. Dynamic Overlay Routing Based on Available Bandwidth Estimation: A Simulation Study. *Computer Networks*, 50(6):742–762, 2006.

Vita

- Education
  - Western Kentucky University, Bowling Green, KY, *M.S. in Computer Science*, Jan. 2004 – Aug. 2006
  - Baskent University, Ankara, Turkey, *B.Eng. in Electrical and Electronics Engineering*, Sep. 1999 – May 2003
- Employment History
  - University of Kentucky, Lexington, KY Research Assistant, Jan. 2007 - Aug. 2011 & Jan. 2012 – now
  - University of Kentucky, Lexington, KY, Teaching Assistant, Aug. 2011 – Jan. 2012
  - Western Kentucky University, Bowling Green, KY, Teaching and Research Assistant, Jan. 2005 – Aug. 2006
- Publications
  - X. Chen, T. Wolf, J. Griffioen, O. Ascigil, R. Dutta, G. Rouskas, S. Bhat, I. Baldin, K. Calvert, "Design of a Protocol to Enable Economic Transactions for Network Services", *In Proceedings of the IEEE International Conference on Communications (ICC), 2015*
  - O Ascigil, K. Calvert, J. Griffioen, "On the Scalability of Interdomain Path Computations", In Proceedings of the 13th IEEE/IFIP Networking, 2014
  - D. Brown, O. Ascigil, H. Nasir, C. Carpenter, J. Griffioen, K. Calvert, "Designing a GENI Experimenter Tool to Support the ChoiceNet Internet Architecture", *In Proceedings of the International Workshop on Computer and Networking Experimental Research using Testbeds (CNERT 2014), in conjunction with ICNP, 2014*
  - D. Brown, O. Ascigil, J. Griffioen, K. Calvert, "ChoiceNet Gaming: Changing the Gaming Experience with Economics", *In Proceedings of 19th International Computer Games Conference (CGAMES), 2014*
  - S. Yuan, O. Ascigil, J. Griffioen, K. Calvert, "Leveraging Legacy Software in Clean-Slate Network Architectures", *In Proceedings of 21st IEEE International Conference on Computer Communications and Networks (ICCCN), 2012*

- O. Ascigil, K. Calvert, "Implications of Source Routing", *In Proceedings of the ACM Conference on CoNEXT Student Workshop, 2012*
- *O. Ascigil, "On the Scalability of Source Routing Architectures", In Proceedings of 19th IEEE International Conference on Communications and Network Protocols (ICNP), 2011*
- J. Griffioen, K. Calvert, O. Ascigil, S. Yuan, "Separating Routing Policy from Mechanism in the Network Layer", *In Next-Generation Internet Architectures and Protocols, Cambridge University Press, 2010*
- O. Ascigil, Y. Diao, C. Ernst, D. High, U. Ziegler, "Generating 4-Regular Hamiltonian Plane Graphs", *Journal of Combinatorial Mathematics and Combinatorial Computing, 2010*
- O. Ascigil, S. Yuan, J. Griffioen, K. Calvert, "Deconstructing the Network Layer", *In Proceedings of 17th IEEE International Conference on Computer Communications and Networks (ICCCN), 2008*