Programmable Interfaces to Advanced Network Processing

Justine Sherry Sylvia Ratnasamy

Draft: Please Do Not Distribute

ABSTRACT

Modern networks implement a range of advanced processing functions that inspect, transform and even store the content of packets they transport. This sophistication, however, is not well reflected in the "interface" the network offers its users through protocols such as IP and programming abstractions such as sockets. As a result, network operators cannot easily expose the availability of advanced traffic processing functions to users nor can users tell the network when and how they want to use these features.

To address this problem, we propose an architecture, NetInvoke*, that allows for a richer interaction between the network infrastructure and its users. NetInvoke is a framework of interfaces and algorithms by which users can request advanced traffic processing while allowing operators control over whether and how a user's request is met. NetInvoke ensures user requests are serviced in a manner that is policy-compliant, robust under network failure, accountable and incrementally deployable.*

1. INTRODUCTION

The Internet was originally designed to provide only one function to its users: best-effort packet transport. Correspondingly, a programmer's interface to the network through sockets and IP allows the user to name little more than the destination of its packets, information required for forwarding.[1] This simple interface was sufficient while networks strictly adhered to implementing the original vision of simple "bits in, bits out" packet forwarding. Modern networks however implement sophisticated processing functions that go beyond simply forwarding packets to inspect, transform and even store the content of packets they transport. For example, today's networks show widespread deployment of middleboxes and router "service blades" that process traffic for redundancy elimination, compression, encryption, protocol acceleration, access control, transcoding, intrusion detection, filtering, caching and so forth [1, 3, 4, 5, 6].

Thus, the Internet today provides many more functions than solely packet transport. Meanwhile, the network interface presented to users still represents only forwarding. Specifically, with the existing interface users can tell the network *where* their packets should be sent but not *how* these packets should be processed, even though the network is capable of diverse processing options.

This mismatch between deployment and abstraction leads to clumsy interactions between the network and its users. Users have no explicit mechanism to opt in, opt out, or configure any of the embedded network functionality. For example, a user A cannot easily request that the packets it receives from S be processed by an intrusion detection system (IDS) before delivery or that a flow to D *not* be routed through a WAN optimizer. At the same time, in the absence of explicit directives from end hosts, operators are left to determine whether and how to apply advanced functions as best they can. Network operators today do so through out-of-band and typically manual configuration or "hijacking" based on ad-hoc packet classification heuristics. The former leads to relatively static and coarse-grained decisions while the latter can lead to unwanted, even incorrect, behaviors. In both cases, the application developer, although well-positioned to make such decisions, is out of the loop entirely.

An additional drawback is that the absence of a standardized approach to discovering and invoking such in-network functionality makes it difficult to *federate* these functions into services that span multiple network domains. For example, a network operator that supports firewall or caching functionality might want to leverage the same functionality in its neighboring networks so as to offer its customers a better experience—dropping packets closer to the source or serving content from caches closer to the requesting user. Without such federation, the reach of services deployed by a network operator is forever limited to a single domain. From the standpoint of ISPs, this not only prevents them from scaling their service (with the performance or security improvements such scaling might bring their customers) but also makes it difficult to compete with third-party overlay-based services.[2]

This paper thus explores the question: given a network infrastructure that implements diverse traffic processing options, what abstraction(s) should the network expose such that users can easily leverage these in-network capabilities? Our proposed solution is NetInvoke, a framework that en-

[1] By a *user* we refer to applications running at endhosts where application behavior may be configured by application developers, individual users or system administrators as appropriate; *e.g.*, it is common that enterprise system administrators can override end-user or default configurations.

[2] For example, a commonly cited reason for the failure of ISP attempts to launch CDN services in the late 1990s was their inability to match Akamai's global footprint [*private commn. B.Maggs*].

ables configuration and inter-domain invocation of advanced network functionalities. We stress that our intent with NetInvokeis not to suggest that the existing network interface is flawed and should be overhauled but rather that it be *augmented* to accommodate advanced in-network capabilities.

NetInvoke is not the first to take aim at integrating middleboxes and the functionality they provide into the broader network architecture. Prior proposals however focused largely on the naming implications of middleboxes, proposing that users explicitly address the specific middleboxes for their traffic to traverse[10, 29]. These proposals resolve the tension of applying unsolicited functionality to users' traffic. But, they leave unresolved the question of how users discover these middleboxes, how users reason about which middleboxes are appropriate for use given routing and topology conditions and, perhaps more importantly, they leave unspecified the role of network providers and their policies in offering and managing middlebox-based services. NetInvoke tackles the above unresolved questions and, in so doing, proposes a somewhat different approach. To reduce complexity for end users and provide network operators with a stake in service selection, we propose that the appropriate abstraction is instead to have users name the functionality itself and leave the network to resolve how and where it is performed. This allows end hosts to decide when to use advanced services, while allowing providers better control of how to support the desired functionality, allocate resources for use, and manage their users traffic.

The design of NetInvoke thus envisages an architecture in which ISPs expose advanced in-network functions through high-level *service interfaces* that are common across different networks; end-user applications can thus program against standardized features, akin to how applications today program against well-known interfaces to Internet services[14]. The difficulty in realizing this vision stems from the fact that the Internet is a federated system and, as such, the functionality a user requests may not be ubiquitously deployed. Moreover, each provider has its independent policy and charging goals that must be respected. Thus a user's invocation must be reconciled with the reality of where the requested functionality is available and hence, whether and how the request should be serviced.

NetInvoke addresses these challenges with a suite of interfaces and algorithms that serve as the "glue" between invocation and deployment, directing user invocations to the appropriate network domain(s) in a manner that: (1) respects the policies of all intermediate networks, (2) adapts to changes in network conditions that impact service availability and, (3) ensures accountability between the user and network domains that actively participate to service the user's request. Because NetInvoke is intended to augment rather than replace IP, we can design it be both incrementally deployable and backward compatible with the existing IP architecture.

To some extent, NetInvoke represents a technical architecture in support of a presumed operational/business model; a

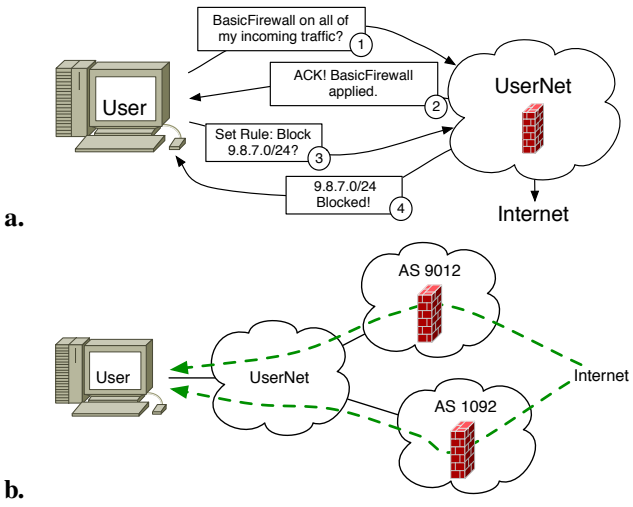


Figure 1: User and network views of service application.

model that makes two key assumptions: (1) that ISPs will seek to offer advanced traffic processing features as a service to users and (2) that ISPs will federate to scale these offerings through bilateral settlements over higher-layer services. While we don't pretend to know whether this model is a certain or even likely outcome, we argue it is viable and desirable. ISPs today already deploy advanced processing features in support of single-domain customer services—e.g., multicasting for U-Verse, VPN services to enterprises. And IP itself serves as an example of ISPs successfully building a federated service through bilateral relationships. We believe this model is desirable because, in the long run, the restrictiveness of the existing network interface acts as a barrier to innovation since it limits the ability for applications to easily incorporate the new services that network operators might deploy and hence the ability for ISPs to monetize their deployment efforts.

The remainder of this paper is organized as follows: we start with a high level overview of NetInvoke in § 2, present its detailed design and evaluation in §3-6, discuss related work in §7 and conclude.

2. OVERVIEW

NetInvoke aims to allow users, *i.e.* end host applications, to program against standardized interfaces to in network functionality. Figure 1a shows our desired interaction: the user names the type of functionality it wants, and provides configurations specific to its behavior. In this example, the user wants to use a simple network firewall, which we refer to as '`BasicFirewall`.' After enabling the `BasicFirewall` (1,2), the user requests that a new rule be added to the firewalls access control list (3,4). Note that the user is never concerned with selecting specific middleboxes to perform the processing, nor does it have to learn about network routing and topology. Instead, it simply names the functionality it wants, leaving the network to perform this selection.

At the risk of abusing an often overloaded term, we refer to this named functionality as a **service**, identified by a **service name** and associated with a set of **service interfaces**. In Figure 1a, `BasicFirewall` is the name for a simple firewall service with `add_rule` and `remove_rule` as its service interfaces. Akin to how developers discover and use APIs to Internet services, we assume service names and interfaces are published through out-of-band means well-known to application developers. While different networks that implement the same functionality are not *required* to support identical service interfaces, we assume they will often chose to do so, incented by the desire to interoperate with applications developed against the interfaces of "first mover" ISP(s) and a rich-get-richer phenomenon.

With application developers programming against only high-level interfaces, their ISP must take on the responsibility of actually selecting where the service is performed. In NetInvoke, a user that wants to use advanced services has one or more **framework provider(s)**, an ISP they direct their service requests to. We term this the user's **framework ISP**, or FISP. For now, we assume that a user has a single FISP which is the same as its access ISP; as we describe later, these are not strict requirements.

The FISP may support `BasicFirewall`, and simply reconfigure local settings to enable the service for the user. Or, it may not support `BasicFirewall` at all. In Figure 1b, the FISP leverages the processing capabilities in neighboring networks, who perform the firewalling functionality on all traffic destined to the user. The ability to enable service processing in other networks allows the FISP to provide access to services it does not itself support, a feature that maximizes service availability when new functionalities are deployed incrementally. Further, the ability to enable processing in other networks provides for a more diverse set of services available to users. Services like WAN Optimization, which requires processing at both endpoints of communication, or Bandwidth Reservation, which requires a QoS guarantee at every hop along the forwarding path, require the cooperation of multiple networks.

Thus, beyond interfaces that speak directly to services, our model requires that users have interfaces to communicate with their networks *regarding* services. In NetInvoke, this communication takes place using a suite of **framework interfaces**. Framework interfaces are distinct from service interfaces: they encapsulate requests *about* services, informing the network what functionality they want, and where. Networks only need to understand service interfaces for the services they provide. If a user tells a firewall to block a particular protocol, or a cache to store a specific type of content, only those networks that perform firewalling or cache processing need to understand these service-specific directives. Framework interfaces are general requests required for service selection—*e.g.*, asking the network to enable or disable a service, or to apply it to 'all incoming traffic'. They don't configure a particular service but tell the network what service to apply and provide a description of where and for what traffic to apply it to. With the framework interface, a user can describe the service they need sufficiently such that their FISP knows how to select external networks to provide the service, without any a priori knowledge of what the service is or how it works.

The main challenges NetInvoke must address include:
(1) User Interface: a user must be able to express its service demands to its FISP, even though the FISP may not support the service or know its requirements.
(2) Network Discovery and Resolution: given a user's request, the FISP must be able to discover the networks the user's traffic traverses, learn which of those networks support the user's desired service, and select an appropriate set of them to provide the service.
(3) Interdomain Authentication and Accounting: a network accepting a service request from another FISP must be able to *authenticate* the request, or verify that the FISP has authority to make configuration on behalf of the user, as well as *account* for the request, or charge the FISP for the services provided.
(4) Service Monitoring: a FISP, once it has selected a set of networks to perform a service for the user's INIT request, must also monitor that the user's traffic continue to receive the service even if routing updates occur.

We discuss these aspects of our design in §3 and §4.

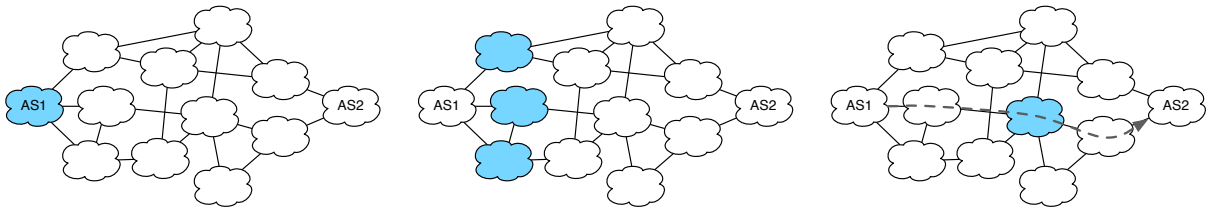
Before turning to the design, we clarify the scope of what NetInvoke tackles. NetInvoke only augments IP and hence cannot solve problems inherent to the current network layer such as BGP convergence issues or network DDoS. That said, users may be able to mitigate the *impact* of some of these problems by using advanced processing services—*e.g.*, through a multipath routing or in-network filtering service. NetInvoke also does not address the question of how a specific service is implemented within a network domain—*e.g.*, addressing how middleboxes or routers are provisioned and placed within the network, how user requests are load-balanced over these. While certainly non-trivial, the success of large-scale cloud services and existing in-network services suggest this is tractable and we leave it to future work to explore this issue in depth.

3. USER INTERFACE

We start by describing the interface NetInvoke offers end user applications (§3.1) and how this interface might be used in the context of different services (§3.2). In the following section we describe the functionality networks must implement in order to support this interface.

3.1 User-FISP Interfaces

As discussed previously, services in NetInvoke represent standardized protocols (service interfaces) accompanied by an identifier (a service name). All communication between a user and its FISP negotiates the use of service interfaces. The challenge to this negotiation is not only that the FISP



(a) Individual **(b) Perimeter** **(c) Path**

Figure 2: INIT request Invocation Patterns.

may not support the desired service, but that the FISP may not know anything about the service's functionality at all. Without information beyond the service name, the FISP has no way to learn that a Firewall should be performed once on inbound traffic, while a WAN Optimizer should be applied on outbound traffic at both endpoints of communication. Our objective is therefore to allow the user to express high level requests that the FISP can resolve *without* a priori information about the service.

Each FISP exposes five basic functions to its users, that together form a *framework interface* as seen by users. To express these functions, the user must communicate with some entity representing 'the network'. For this, the FISP exposes an *interface server* which users can discover either through a DHCP lookup or through manual configuration. Users send messages to the interface server to express their service requirements. While our discussion treats the interface server as a single entity, a large FISP may use standard approaches for replicating interface servers to divide the task of managing requests from a large user base.

We now discuss our five functions, focusing on the INIT function, which is the core of the framework interface.

INIT. A user invokes the INIT function to enable a service. The user specifies a service name and a set of parameters, and the FISP either provides the service functionality locally or selects a set of external networks to provide the functionality. In essence, the INIT message provides information about *what* traffic the service should apply to, and *where* the service should be performed; the FISP then uses its knowledge of the network to resolve this into a set of specific networks that the traffic traverses and which are service capable. We discuss how the FISP acquires this knowledge and makes resolution decisions in the following section (4). If the FISP cannot resolve the INIT request, it returns an error message to the user. If it can resolve the request, it forwards the INIT to external networks it selected and then returns to the user a token identifying the request. As we shall see, this token is later used when the user needs to update or interact with the service configuration.

Since services vary in where they should be invoked, parameters allow the FISP to select an *appropriate* set of networks to perform the users request: for some services, appropriate networks are those closest to the requester, while for others, the appropriate networks are those furthest away. A user thus needs a service-agnostic abstraction by which to express what networks are appropriate choices for its requested service. For this, we introduce the notion of an *invocation pattern* which is a model that captures how and where the network should apply the service. We identify three key invocation patterns that we believe allow us to represent a broad set of service requirements—these are shown in Fig. 2. INIT requests thus come in one of the above three forms, each of which describes a particular invocation pattern with its associated set of parameters.

'Individual', in Fig. 2a, is the most basic of the three invocation patterns. With this INIT request, the user specifies an IP address, and the FISP assigns the service to the ASN with authority over that address. There are no other parameters. A user issues an Individual INIT request when it wants the service supplied in a particular network, *e.g.*, ad insertion for flows in AT&T's network.

'Perimeter', in Fig. 2b, constructs a border of service-performing networks between the requester network and an external address, such that all possible paths between the request and the provided address must traverse the border. Perimeter INIT requests use a limited number of parameters:
External Prefix: the border is constructed between the requester and this specified prefix. A /0 requests a border around the requester for *all* incoming traffic.
Proximity: whether or not the border should be near the requester, or pushed towards the specified address.
Partial Coverage: If a complete border cannot be constructed, construct a partial border.

Finally, **'Path'**, in Fig. 2c, applies a service along a path between the requester and a specified IP address or prefix, for either incoming or outgoing traffic. Path requests include the most descriptive parameters of the three invocation patterns. For instance, the user may specify that the service be applied in a network near the requester, or near the external address. The user may specify that the service be applied once along the path, or multiple times along the path. Drawing from previous efforts at taxonomizing middlebox behavior [11] and our analysis of usage scenarios (§3.2), we arrive at the list of parameters shown in Table 1.

For outbound 'Path' INIT requests (from the requester to an external prefix), the user can additionally specify that a path request be 'active' or 'default'. For default configurations, the INIT request will only succeed if the default forward path conforms to the INIT parameters. However, if the request is marked 'active', the user permits the FISP to redi-

Function	Forwarding requirements
User's prefix	Prefix or IP (/32) whose traffic should traverse the service.
External prefix	External prefix or IP whose traffic to or from the user should traverse the service.
Incoming, Outgoing, or Bilateral	Whether the service should be applied on traffic destined to the requester, from the requester, or on traffic flowing both directions to and from the requester.
Frequency	How many times the service needs to be supported on the traffic's path. (1) once, (2) twice, (3) as many times as possible, (4) at every hop of the path.
Placement	Whether the service is required in the user's network, required in the external prefix's network, should be place near the users network, or near the external prefix's network.
Single Instance	If true, traffic cannot be split across multiple instances of the same interface. A user of an Intrusion Detection System would likely set this value to true, since this interface needs to performs analysis over *all flows* destined for to the user to provide complete analysis.
Active/Default	If 'Active', outgoing traffic may be redirected along alternate forwarding paths to reach instances of the service. User must embed INIT token in their traffic to be redirected. If 'default', traffic flows over default routing paths only.
Path Inflation	For 'Active' requests, whether or not the chosen forwarding path is allowed to be much longer than the default path to reach the service.

Table 1: Parameters for the 'Path' INIT requests.

rect their traffic along alternate forwarding paths in order to reach networks that support the service (we discuss in the following section how the FISP can perform such multipath routing). For these active requests, the user specifies an extra parameter; whether or not 'inflation' is allowed in the final path. If inflation is allowed, the alternate forwarding path may be longer than the default path. The user may mark the request as 'no inflation' if the service is performance enhancing and an inflated path would detriment performance.

Once an 'active' INIT request is resolved and invoked, the user embeds the INIT token in packets may be redirected along the service-performing path. Requiring that the user embed the token in flows to be redirected provides the user with fine grained control over when the network redirects traffic to reach the service. Thus, if the active request inflates the path, but the service need only apply to HTTP or VoIP traffic, the user can label only that traffic for redirection, leaving other traffic to traverse the default path. Note that embedding tokens in this manner also allows users to establish multiple INIT configurations for the same destination, sending packets along paths with potentially different characteristics or service instances.

These invocation patterns and their respective parameters allow us to describe a wide range of service requests, as we illustrate in § 3.2.

After enabling the user's services in external networks, the FISP creates local state that maps the user's token to details of the networks assigned to perform the user's service for later use. We describe in § 4.2 implications of this state for FISPs.

RELAY. After the FISP enables the service, the user may need to invoke subsequent service-specific functions. For this, the user can use RELAY messages as a shell for the messages defined in the service interface. With a RELAY message, the user specifies the relevant INIT token, and then places whatever service-specific messages are defined under the protocol. On receiving a RELAY message, the FISP maps the user token to the corresponding set of services ASes and forwards the user's message to these ASes. A user may likewise receive RELAY messages *from* the interface server, containing messages generated by a service AS.

KEEPALIVE. This message extends the lifetime of the service. The network periodically culls long-living INIT state, but not without first performing a check with the user. The FISP sends the user a KEEPALIVE? query along with the INIT token; if the user responds with a KEEPALIVE of its own, the service state is not evicted.

TERM. Deletes state for the INIT configuration, after forwarding the termination signal to the external networks assigned to the user's service.

GETCONFIG. Requests a list of all configurations (INIT tokens, services, and parameters from INIT messages) which apply to the user's IP address or prefix, to aid management.

3.2 Service Examples

To test whether NetInvoke's parameterizable INIT messages can express the needs of a broad set of services, we explored their application to six potential in-network services. The parameter settings we derived for each are shown in Table 2. Due to space constraints we only describe two of these in greater detail—WAN Optimization and a Federated Firewall. The remaining scenarios are detailed in our technical report (available on request to the PC chairs).

WAN Optimization

WAN Optimization typically operates solely between dedicated endpoints, often between dispersed networks of a single enterprise, processing traffic between the two endpoints for de-duplication, compression and protocol-specific opti-

Service Type	Invoc. Pattern	Parameters
WAN Optimization	Path	Twice; At Local, Near External; Bilateral; Single Instance: No; Active, Inflation Not OK
Bandwidth Reservation	Path	Hop by Hop; Outgoing; Single Instance: No; Active; Inflation OK
Path Selection	Path	Once; Near Local; Outgoing; Single Instance: No; Active; Inflation OK
Transcoder	Perimeter	User's IP address or ASN; Near Requester; Partial Coverage OK
Anonymizer	Perimeter	/0; Near Requester; Partial Coverage not OK
Firewall	Perimeter	/0; Away from Requester; Partial Coverage OK

Table 2: INIT parameters for six example services.

mizations. The benefits of such processing are twofold: a reduction in bandwidth costs and an improved user experience through what vendors term 'application acceleration.' Today, manual configuration limits such services only to preconfigured endpoints.

With NetInvoke, a FISP that supports WAN Optimization can discover when compatible processing is available in external networks, and thus allow its users to use WAN Optimization in a fine-grained and opportunistic manner.

INIT request. To initialize end-to-end WAN Optimization, the user sends a Path INIT request containing the user's own IP address and the address of the external host to which the traffic is to or from. The path INIT request should declare that the service be applied bilaterally, so that both incoming and outgoing traffic for the user be minimized. The frequency should be set to 'twice', and the placement should be at the user's network and near the external address, so that both endpoints process the traffic. 'Single instance' should be set to false, since different flows can be split across multiple WAN optimizers without disrupting service. The user may choose active or default; an active request would broaden the scope of WAN optimizers near the external endpoint available for use.

Resolution. Because the user's request requires local support of the service, the FISP first looks to verify that WAN Optimization is a locally supported feature. If it is, it then looks at the default path and checks whether the destination AS, or the second-to-last network in the AS path to the destination, supports the WAN Optimization service. If so, the FISP relays the user's request to the external service AS, establishes service state, and reports back to the user. If not, and if the FISP supports the discovery of alternate paths (described in §4.1), then it will check these alternate paths to see if they involve a different AS as a 'last hop' before the destination AS. Note that the FISP will only accept an alternate path which does not have any path inflation. If the FISP does discover such an AS, it queries the AS for service availability. If either the local AS does not support WAN Optimization, or it exhausts all paths to the destination without finding an external AS to support the second instance of the service, the FISP returns an error to the user.

Federated Firewall

Because the Internet does not provide the ability to shut off traffic from an undesired source, firewalls perform filtering within the network necessary to drop malicious traffic and block Denial of Service attacks. However, firewalling deployed near the recipient is not always sufficient to mitigate the degradation in service from a massive flow of traffic as in a DDoS attack. Consequently, 'pushback' mechanisms propose that filters propagate backwards from the victim towards the originator of the attack, dropping the flow of malicious traffic before it overwhelms the recipient [22] Since NetInvoke enables multiple networks to support the same service interface, firewalling under this model can provide for *federated* firewall service across multiple networks.

INIT request. To enable the firewall, the user would first start with a Perimeter INIT request, requesting that the firewall be deployed between the user and all external addresses (/0). The user might allow incomplete coverage, since even if firewalling cannot be totally deployed, some filtering may lessen the load of an overwhelming amount of traffic.

Resolution. To resolve the request, the FISP would first look locally if the user's immediate network supports the firewall. If it did, it would invoke the service there. If not, it would look to the set of networks neighboring the user's network, and see if they supported the service. For those that did not support the service, the FISP would query their neighbors, and so on. Once the FISP has either discovered appropriate networks to construct a perimeter, or exhausted some limit for querying, it would invoke the firewall in firewall enabled networks it discovered and report back to the user.

4. NETWORK REQUIREMENTS

We now describe the functionality that FISPs and service ASes must implement in order to support the interface presented in Section 3.

4.1 Discovery and Resolution

Ideally, to resolve the user's INIT request into a specific set of service-providing networks, the FISP would have a detailed map of the entire Internet, with an omniscient view of routes and network conditions. Realistically, this is neither something networks are capable of, nor is it an easy goal to work toward. Instead we require only that networks obtain a view of the network that is sufficient to address all three types of INIT queries.

In this section, we describe how the FISP can discover (1) what services any network supports, (2) a set of neigh-

bors for any network, (3) an AS-level set of paths to and from the user for any external IP address. We show that these three pieces of information are sufficient to resolve Individual, Perimeter, and Path INIT queries. However, we note that the particular data and algorithms we describe here are not the only way to acquire the required information. A FISP may use more sophisticated methods or exploit inter-ISP relationships to acquire a more detailed view of the network than we describe and such sophistication would only improve NetInvoke's capabilities. With our discussion, we aim only to show that resolution is reasonably achieved with limited information and basic techniques.

Service Discovery. We start with a simple Individual INIT query: a user u requests that its traffic en route to v receive the `EasyTranscode` service in v's AS. With an Individual INIT, the FISP knows which AS it wishes to perform the service: it performs a lookup on v's IP address and retrieves the appropriate ASN. It needs only to know whether or not v's AS supports the NetInvoke framework and the `EasyTranscode` service. The FISP obtains this information through a simple DNS lookup. We imagine a universal registry; perhaps a lookup on `7018.NetInvoke.arpa` would resolve to the IP address for a server controlled by 7018. For simplicity, we refer to this server as an interface server as well, although it need not be the same entity as the interface server exposed to users.

From there, the FISP interface server queries 7018's interface server directly to discover whether or not it is capable of performing `EasyTranscode`. If the AS does not support `EasyTranscode`, the FISP returns an error to the user. In addition, the FISP may cache the results of the query to avoid requerying in the future; networks are unlikely to frequently deploy or withdraw services.

Neighboring Networks. We now move to an example where u sends a Perimeter INIT request, requesting that a `BasicFirewall` be deployed between itself and another address v. If the the perimeter is to be placed around u itself, then it is straightforward for u's FISP to obtain the required topology information: BGP provides an AS path for every globally announced prefix and the union of these paths reveals the networks that might potentially form such a perimeter, starting with u's AS itself. However, more work is required if u's request is that the firewall be placed as close to v as possible. If v's ISP supports `BasicFirewall`, this request is simple as a minimal border around v is its own ISP. If the ISP does not support `BasicFirewall`, the FISP will have to look to other networks to construct a border around v's ISP. Hence, Perimeter INIT requests require that the FISP have some understanding of AS-level topology, as one has to discover a set of ASes which form a border between the user and v. One plausible approach is simply that u's FISP query the destination AS for the information. However, this may fail if the destination AS does not support NetInvoke, is untrusted, or unwilling to reveal the information. Hence, we instead assume the FISP learns the set of neighbors for any AS using published Internet topology maps. We recognize that most up-to-date sources of such information are measurement based, but the accuracy of topology mapping efforts [2] and inferences from neighboring ASNs in BGP paths make us consider this a reasonable, if imperfect, solution. Moreover, a FISP can always complement this information with that learned from directly querying ISPs when possible.

Once the FISP has access to the set of neighbors for any network, it can start at one of the endpoints identified in the INIT request and recursively query each ASes neighbors until it discovers enough `BasicFirewall` ASes to construct a suitable border. In our starting example, in which u requests a perimeter between itself and v, near v, the FISP would first start by looking up v's ASN. Then, it would query the AS to see whether or not it supported `BasicFirewall`. If so, it would invoke the service in that AS, constructing minimal perimeter around v. If v's AS does not support `BasicFirewall`, the FISP must then query each of the AS's neighbors and see if all of them support the service. If any of them do not, it may query their neighbors, and so on. This outward querying continues until either the FISP can construct a perimeter, or the FISP crosses some threshold for number of queries performed. If it reaches the threshold and fails to find a total perimeter, it either fails and returns an error (if the user demanded complete coverage), or forwards the INIT request to the set of networks which do support the service (if the user allows partial coverage).

Routing. Finally, to resolve a Path INIT request, u's FISP needs to know the AS-level paths for traffic both to and from u. For default requests, the FISP only needs to discover the default forwarding path. Obtaining the path u takes en route to an external destination is straightforward: BGP tables provide an AS Path for every globally announced prefix. Alternatively, the user or FISP can issue a traceroute to obtain the same information, converting the traceroute's IP level path into an AS level path [23]. For traffic destined to u from some external source v, recent work [18] makes obtaining the reverse traceroute (from v to s) reasonable. In addition, just as ISPs deploy looking glass servers today, networks may choose to deploy looking glass services within NetInvoke to provide a better view of traffic for services impacting their customers' traffic. Once the FISP discovers the relevant path, it can then look up the interface server for each ASN, query them for whether or not they support the requested service, and decide whether or not the path fits the users INIT parameters. If it does, it enables the service at the appropriate ASes, and if not, it returns an error. We note that, unlike Individual and Perimeter services, Path services are vulnerable to service disruption caused by routing updates that change a path after services are enabled; we discuss the implications of such changes in § 4.2 and measure their effect in § 6.

To maximize service availability for Path requests, NetInvoke uses a multipath route discovery protocol that reveals

alternate paths to a destination. Thus, even if u's default path to v does not traverse an AS that offers `EasyTranscoder`, a FISP can enable multipath routing to redirect u's traffic along an alternate path that traverses an `EasyTranscoder` AS. If the user flags their Path INIT request as 'active', the FISP may redirect the user's traffic along an alternate route which fits the users service demands. After weighing the literature on multipath routing proposals, we chose MIRO-style [32] routing because it is simple, backwards-compatible with existing routing and BGP and functions through bilateral agreements in a manner that does not require the participation of every AS on the path and allows transit networks to enforce policy in the multipath routes they allow. Under MIRO, u's FISP can query other networks that support MIRO and ask them to expose any alternate paths they have to v. The queried AS can choose which paths it wishes to expose or not, allowing it to enforce any policy from only exposing the default path, to exposing all valley-free paths, to exposing any path it has available. If any of the exposed paths fit u's constraints, the FISP can select that path and tunnel u's traffic to the queried AS, using it as a 'waypoint' that will then forward it along the path selected by the FISP.

We use a simple algorithm to explore alternate paths: for a destination D, a FISP queries in turn every AS along the path to D for alternate paths until it finds a path which conforms to the user's INIT request. If it doesn't find a matching path, it may go back to the alternate paths discovered, and query the ASes in those paths as well.

4.2 Monitoring

Once the FISP successfully resolves a user INIT request, it maintains state relevant to the user's request. Associated with the user's INIT token, the FISP stores: (1) the parameters of the user's INIT request, (2) the set of networks assigned to provide the user's service, and (3) the paths that traverse those networks. This information is useful, for example, when a user wants to RELAY a message to all of the service instances for a particular INIT request, or calls GETCONFIG to get a list of rules applied to its own traffic. More challenging, the FISP must also monitor the service performing paths to ensure that they continue to meet the user's demands. While the user application and service interface are responsible for monitoring the service functionality itself (*i.e.* is it performing its duties correctly), the FISP is responsible for ensuring only that the users traffic continue to traverse the network assigned to the service in case of routing updates or failures.

For this, we consider default and active routing for PATH requests. The FISP can monitor INIT configurations assigned to forward, default paths by watching BGP path updates, or by periodically issuing traceroutes and watching for AS-level path changes. For INIT configurations for reverse paths, it can periodically issue traceroutes along this path or have an external accomplice at the other endpoint inform it of changes. Finally for active requests, the waypoint AS (as part of MIRO's operation) will notify the FISP explicitly that the desired path is no longer valid when the FISP attempts to forward traffic along the now invalid path.

In any of these cases, if the updated path still contains the service-performing ASes and conforms to the initial INIT parameters, no change is necessary. If the INIT configuration is active, the network may re-query to find an alternate path that traverses the originally assigned service-providing ASes and reroute traffic along that path. If the traffic can no longer traverse the originally assigned service-providing ASes, it reports an error to the user. The user can then either respond with a TERM message (to cancel current usage), or a KEEPALIVE message, in which case the network may query to find entirely new paths which conform to the original INIT request, terminating the old services in the process.

4.3 Authentication and Accounting

The previous sections described inter-ISP interactions that negotiate the use of services and alternate routes. We require that these interactions between networks be both *authorized* and *accountable*. *Authorization* means that requests are verified to originate from an entity with authority over the impacted IP address or prefix. Without authorization, malicious networks or users could easily attack others by invoking service interfaces on their behalf (for example, by creating a BasicFirewall setting to block all of another user's incoming traffic). *Accountability* means that for any service provided, the network always has means to charge for use of the service. A network performing a service should have a way to prove that another network requested the service, and have a clear relationship with someone responsible to pay for the service. Our demand for accountability does not reflect any particular payment model, but only guarantees that there can be some network in place to hold responsible for payment so that if networks wish to develop billing models, they may. To allow networks to communicate their service needs with both authorization and accountability, NetInvoke provides two simple interdomain functions that *encapsulate* a user's requests: SVC_REQUEST and PEER_REQUEST.

SVC_REQUEST. SVC_REQUEST allows a FISP to forward a user's request to an external network. A sender AS1, invoking a service interface in AS2, sends a SVC_REQUEST to AS2 message which wraps any of the basic interface functions (for example, INIT). AS1 signs the request with their private key and sends it to the interface server for AS2.

If the two networks are peers, this extremely simple interaction is sufficient to provide both authorization and accountability. As peers, the networks exchange their public keys and allocated prefixes out of band as part of their real-world business relationship. AS2 can verify the authenticity of the request by checking the message's signature and checking that the IP or prefix impacted by the configuration is in a prefix allocated to AS1. AS2 knows that the configuration is accountable, because it has a signature proving the request came from AS1 (proof of the transaction) and it al-

ready has channels in place for settlement with its peer. An AS could lie to a peer, but peering relationships reflect real-world trust. Violation of that trust is cause for severance of the relationship and even legal action. Thus, it is reasonable to believe that a peer will not misrepresent itself in this real-world context.

Networks which are not physically connected may still peer at the service level. With such *service peering*, unconnected networks establish business arrangements just as they would as physical peers. They exchange public keys and allocated prefixes, and establish channels for billing. Thus, the same model of real-world trust which allowed peering networks to exchange services can extend to any two networks whose owners are willing to sign a legal contract.

PEER_REQUEST. With SVC_REQUESTs, a FISP enters direct settlements with service ASes. While simple, it can unrealistic for a FISP to maintain service peering relationships with a large number of other networks. To resolve this a second function, PEER_REQUEST allows non-peers to invoke services with each other while achieving authentication and accountability. PEER_REQUEST works by allowing the service network to authenticate that the FISP has authority over the user's address, but perform accounting with some other network. The FISP can forward its request to a peer it shares with the service network, and then this peer can relay the request to the service network, as such offering to pay for service use on the FISP's behalf. By forming a chain of accountability, the service network AS3 can charge the mutual peer, AS2, who will in turn charge the original requesting FISP, AS1.

With this, AS3 no longer needs to account with the requester AS1, but only to authorize AS1. AS3 can use one of two mechanisms to authorize AS1. First, AS3 can make use of Public Key Infrastructure if available. While the deployment of PKI is a formidable demand, current proposals for secure routing require exactly the same support as NetInvoke. Both S-BGP [19] and so-BGP [31] perform origin authentication, which requires a mapping from ASN to public key, and from any IP address prefix to the public key of the AS authorized to announce that prefix. Thus, NetInvoke can build off of the same infrastructure that serves these protocols, if deployed.

In the absence of such infrastructure, AS3 can perform a 'checkback' to authorize the requesting AS by asking the IP address impacted by the nested INIT message. With a checkback request, the AS sends the full signed request back to the impacted IP. The host can then verify that the signature represents its own provider, and affirm that the invocation is acceptable. However, the checkback option has several limitations. First, this option can only be used for invocations performed on behalf of a single IP address; checking back for INIT functions which impact whole IP prefixes is impractical. More significantly, the communication can be man-in-the-middled by an intermediary. However, this attack is well-constrained: the man-in-the-middle must already have access to the traffic between the end user and FISP, implying that the man-in-the-middle should be able to manipulate the user's traffic already. Thus, while the threat is constrained, the network has some responsibility to ensure that this option is only used for interfaces which don't threaten security-critical operations.

Once the initial INIT command has been established through AS2, AS1 can exchange SVC_REQUEST messages directly with AS3. To terminate the configuration, the terminating party must once again intermediate the SHUTDOWN message through a RELAY_SVC message through AS2, to ensure that AS2 is informed of the service termination as well.

5. DESIGN DISCUSSION

We claimed earlier that NetInvoke is deployable and backward compatible with the existing IPv4 infrastructure. We now review the key requirements of our design with consideration for deployability.

So far, we have assumed that all ISPs implement NetInvoke's framework interfaces and functions and hence user applications have relied on their local network provider serving as a FISP. We now describe how this assumption can be lifted. A user whose local network provider does not support NetInvoke has two options. First is that the user serve as its own FISP. This is feasible provided the user is: (1) capable of implementing the network topology and route discovery mechanisms from § 4, using measurement and available topology repositories and (2) that remote ASes are willing to directly settle with users. If the latter holds, a user might also use alternate routes, tunneling directly to waypoints. We believe this scenario is practical for users in large enterprises or campus organization where direct interaction/settlements with remote ISPs is more likely.

The second option, amenable to even individual users, is that a network other than the user's direct provider serve as its FISP. One question is the vantage point from which the (non-local) FISP makes INIT resolution decisions. If the user is able to issue its own topology measurements, then the external FISP can use these (in combination with general topology information it has available) to select services in other networks on the user's behalf. We envisage that a FISP provides its remote users with the required measurement tools when activating the user-FISP account. Every request must be verified with a checkback, though. Further, active service requests are not possible unless the user forward their traffic to the FISP for redirection - the FISP cannot make changes with the user's local ISP.

With the above, we argue NetInvoke is practical for deployment for the following reasons:

- NetInvoke accommodates partial ISP adoption of both service and FISP functionality
- NetInvoke operates with the incumbent ISP infrastructure, requiring no new overlay service providers (unlike [8]) or global naming services (unlike [26, 10])

- Although our design benefits from availability of a PKI, it does not strictly require it (unlike [26])
- NetInvoke imposes on an ISP the burden of 'service peerings' (negotiating new services with remote ISPs) and MIRO-style peerings (negotiating the use of alternate paths with remote ISPs). However all such peerings are optional; the downside to being degraded service availability to the FISP's users. Moreover, an ISP can bootstrap these negotiations with the peering relationships it already has with its physical peers.
- NetInvoke does not require that both endpoints in a communication support framework interfaces or a specific service. *E.g.*, a popular server can invoke a federate firewall service with no change to the client side (unlike proposals such as [33, 26])
- Finally, we (optionally) use MIRO for multipath route discovery. As shown in [32], MIRO is backwards compatible with BGP and can be implemented with existing router equipment.
- NetInvoke only affects a router's fast path when processing packets for an active, Path service. In this case, the router must map a user token (as opposed to the packet's destination IP address) to the next-hop on the alternate path; this is easily done with an exact-match entry and (based on the router architecture) can be implemented with TCAMs on the fast path prior to the regular longest-prefix match lookup in the FIB.

The above gains do not come for free. The compromise NetInvoke makes is that it offers only best-effort access to services (which, given the topology-sensitive nature of some services, appears inevitable under partial deployment) and cannot overhaul the fundamentals of the underlying IPv4 service model (*e.g.*, we conjecture that much of the complexity and deployment burden of proposals such as capabilities[33] and RBF[26] arise because they transform the core IP service model to be 'default off'[16]).

Finally, our design also aims to align the incentives of all parties: users benefit through improved application behavior; service ASes can charge for service use while FISPs benefit through charging users (for FISP connectivity) and potentially through settlements with service ASes that they select on behalf of their users.

6. EVALUATION

Although NetInvoke guarantees only best-effort access to services, it incorporates various mechanisms such as multipathing and remote discovery aimed at maximizing service availability under partial deployment. In this section, we evaluate these mechanisms, measuring service availability under a range of network and deployment conditions.

The ability of a FISP to satisfy a user INIT request depends on several input factors, including:
(1) What fraction of ASes deploy a service?
(2) Are edge or backbone networks more likely to support a service?

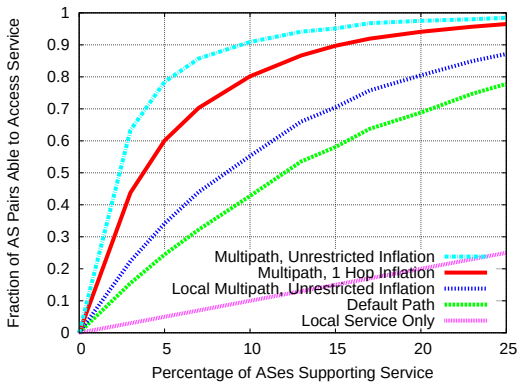


Figure 3: Fraction of AS pairs able to access service with and without multipath.

(3) What alternate routes are ISPs willing to export?
(4) What number and location of service ASes are needed to satisfy the user's INIT request?

We note that the above factors are not specific to the details of NetInvoke, and hence our results are largely intrinsic to any interdomain use of services under partial deployment. To the best of our knowledge, this is the first evaluation of the feasibility of access to network services under the constraints of limited deployments.

Methodology. We built an AS-level routing and topology simulator modeled after those used by other groups [17, 20, 32], but tailored to studying service access and path diversity. Our simulator modeled the AS graph from January 20, 2010, with inferred peering relationships provided by CAIDA [2] and RouteViews [7]. This data included 33,508 AS nodes, which we further annotated as either 'backbone' or 'edge' networks. We define backbone networks to be networks with 250 or more peers. The remainder of the nodes we annotated as edge networks.

For each simulation, we randomly annotated a fraction of the ASes as 'service' ASes, indicating that the AS supported the processing required for some desired service interface. For the majority of our simulations we assumed that networks sought services that were anywhere on their forward path, once, and that the INIT request was flagged 'active.'

To simulate routing, we assigned each AS a single prefix. Each network then propagated paths using standard valley-free routing policies. When choosing between multiple paths of equal policy rank, networks selected shortest paths, followed by paths with the lowest next hop AS number. For our simulations, resulting paths represented standard BGP routing to each AS. We then executed MIRO [32] style multipath routing. In the majority of our simulations, a sending network inspected its default path and queried each AS for alternate routes to the destination, starting with the next hop AS and moving toward the destination.

6.1 Benefit of Multipath

We investigated the likelihood of AS pairs having a service-

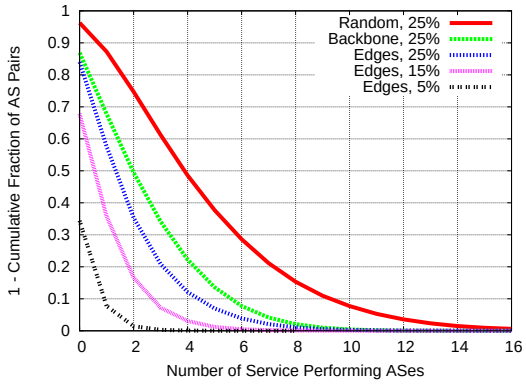


Figure 4: Diversity of single service paths.

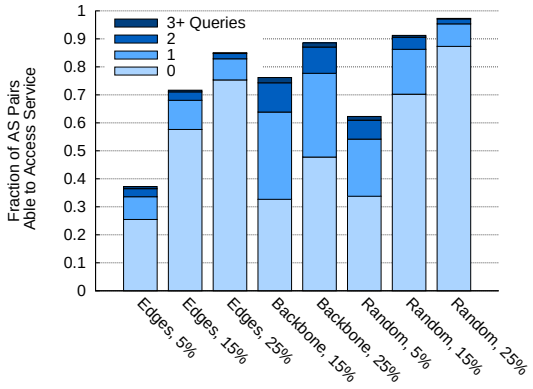


Figure 5: Number of queries until discovery of single service providing path.

providing path between them with and without multipath routing. We simulated service deployment in settings where between 0 and 25% of randomly chosen ASes enabled the desired services. For each simulation, we randomly selected 10,000 source-destination AS Pairs. We then inspected all paths available under four routing models: (1) *default* routing, (2) *'local'* multipath, in which a (potentially multihomed) FISP selects among paths from its neighbors only and performs no additional queries, (3) *multipath with one-hop inflation*, in which the FISP considers paths exposed by ASes along the default route and a path is only accepted it is are no more than one hop longer than the default path, and (4) *unrestricted multipath* which removes the 1-hop restriction of the previous case. The first two models do not require MIRO's support for multipath while the latter two do. We restricted the MIRO-based schemes to use of a single 'waypoint' AS. We then logged whether each AS pair had a path which traversed a service supporting AS under each model.

Figure 3 shows the fraction of AS Pairs with access to the service for each of the multipath models. The x-axis shows that a random x fraction of ASes perform the required processing. The y-axis shows the fraction of AS pairs with a path between them which includes at least one service supporting AS. For completeness, we also plot the case where users can only access a service if it is supported by their local ISP; this is the case today with no access to services in remote ISPs and, as expected, we see that service availability in this case simply tracks deployment levels. At all levels of service deployment, we see that both multipath and access to services in remote ISPs significantly increase the fraction of AS pairs able to access services. At limited deployments, the gains due to multipath are most significant. When 5% of networks perform service processing, 24.5% of AS pairs encounter a service-performing AS on their default path, while 35.2% do so with 'local multipath'. This is encouraging in that it represents a 5-7$\times$ improvement in service availability relative to the baseline deployment level *without* requiring the mechanisms of MIRO. However the absolute service availability remains low. But, when the sender uses MIRO and accepts paths which inflate the AS path by no more than one hop, 60.1% of the pairs can access a service performing AS. Even for services with widespread support, there are noticeable gains from multipath: 96.5% of AS pairs can access a service AS using multipath with one hop inflation.

It is conceivable that FISPs have different policies regarding what service-providing ASes they would like to interact with. As an indicator of their ability to act on such policies, we also considered the prevalence of service supporting ASes between source and destination. *I.e.*, when a sending AS has a path to its destination which supports a desired function, how often does it have choice? Figure 4 shows the number of service-supporting ASes for AS pairs at different deployment levels and for different deployment models. The graph plots the complementary cumulative fraction of AS pairs with a given number of service-supporting ASes available to them. Thus, a given (x, y) coordinate says that y fraction of AS pairs had greater than x service ASes available to them. We see that at high levels of deployment, a majority had more than one AS available to provide the desired service. The number of options drop when service support is restricted to edges or backbone networks, though a majority of paths still had more than one AS available. At lower deployment levels, choice was less common.

In summary, we found that service availability is greatly improved through multipath routing and access to remote services and that multipath routing often revealed more than one service providing AS to choose from.

6.2 Querying Overhead

We logged the number of queries required as an AS queried its AS path until it found its first AS that provided a service. We queried approximately 200,000 random pairs and allowed a single waypoint and one hop of inflation. We display the results in Figure 5.

We first observe that the majority of AS Pairs find their first service-performing path locally (*i.e.*, at their FISP), or with their first external query. This means that the majority of gains due to multipath come by querying only one hop upstream, to a physical peer or provider. Thus, an AS which

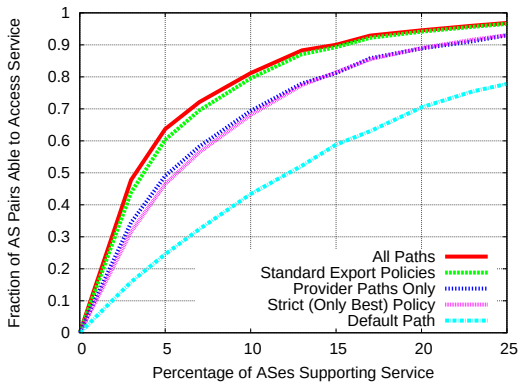


Figure 6: Impact of network policy on service discovery.

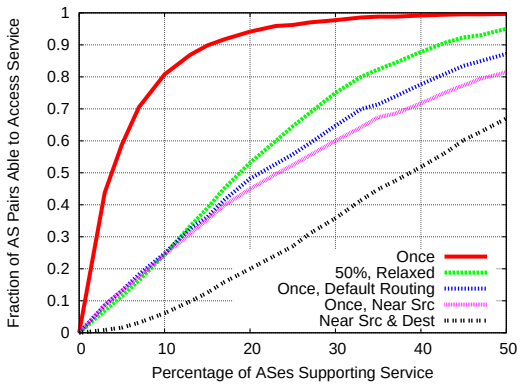


Figure 7: Fraction of AS Pairs able to access service given varying service demands.

establishes peering relationships for multipath can reap the majority of benefits even if they only query their direct neighbors; *i.e.*without entering relationships with new ASes.

We also observe that external querying provides great benefit when services are available in Backbone rather than Edge networks. This is intuitive since Backbone deployments are 'deeper' in the network and hence require additional querying.

In summary, we find that the querying cost for service discovery in NetInvoke is low and that FISPs can offer their users high service availability through interaction with just their existing neighbors in the AS topology.

6.3 Export Policy

A network acting as a 'waypoint' AS for multipath may export alternate paths to requesting networks, or choose to keep them hidden. At 10 intervals between 0 and 25%, we randomly chose 10,000 AS pairs and found all paths between source and destination given that intermediary ASes exposed All Paths, exposed paths under one of five policies discussed in MIRO [32]:

(1) All Paths: expose all available paths. This is a liberal policy but may in fact be a reasonable one since it is likely that ASes will charge for waypoint routing and hence the requesting FISP appears as a customer.

(2) Standard Export Policies: use the previous-hop AS to determine whether the backup path would be valley-free. Expose only those paths which are valley-free.

(3) Provider Paths Only: expose all paths whose next-hop AS is a provider.

(4) Strict (Only Best) Policies: expose all paths which are 'as good' as the best path. That is, if the default path is a customer path, expose only customer paths, etc.

(5) Default Path: expose only the best path as in default routing.

We allowed the paths one waypoint, and no more than one AS hop of inflation. Figure 6 shows the results of these policies at 0-25% random deployment. On the x-axis we show the fraction of ASes supporting services, and on the y-axis we see the fraction of AS pairs with service available.

As in MIRO, we observe that standard export policies are very close to the 'best' case in which an AS exposes all available paths. Even the strict model provides significant gains over default paths, more so at low deployment levels than high. Under strict policy, where the network exposed only paths as good as the default path, 46.5% of AS pairs found service paths at 5% service deployment, and 93.0% of AS pairs found paths at 25% deployment.

6.4 Service Requirements

Some types of services may require more than one instance of the service to operate. To investigate the feasibility of resolving these more demanding constraints, we evaluated service availability under a variety of scenarios:

(1) Once: the service must occur once, anywhere on the path

(2) 50%, Relaxed: the service must occur in more than 50% of ASes on the path. The request is 'relaxed', in that we allow for up to two waypoints and two hops of path inflation.

(3) Once, Default Routing: the service occurs once and is available on all paths between source and destination (either in the source AS, all of the source ASes neighbors, at the destination AS, or all of the destination ASes neighbors).

(4) Once, Near Source: the service is available either in the source AS, or the source AS's next hop to the destination.

(5) Near Source and Dest: the service is available near both the source (source AS or next hop from source) and the destination (destination AS or hop prior to the destination.

At deployment levels between 0 and 25%, we simulated paths selection for 10,000 AS pairs and for all but the 'Relaxed' path we allowed only one waypoint and one hop of inflation. Figure 7 demonstrates the relevant results. Like Figure 3, the y-axis shows the fraction of AS pairs with a path that provides access to the service, and the x-axis shows the fraction of ASes deploying the desired service interface.

We see that location constraints limit the availability of paths more than requirements for multiple instances of the service. The demand that 50% of hops on the AS path performs better than any of the models where location of the ser-

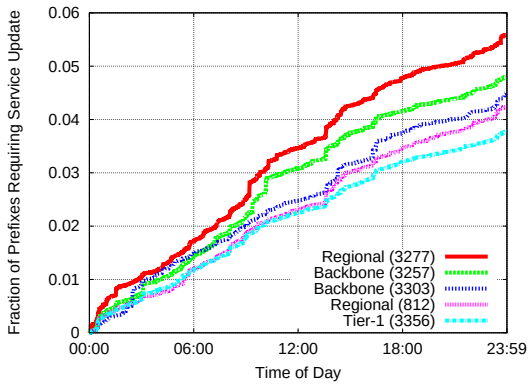


Figure 8: Stability of paths on January 21, 2010 for five ASes monitored by RouteViews. Simulation assumes prefixes are assigned services at midnight.

vice is restricted to the edges. This makes sense for two reasons. First, those without location constraints simply have a larger pool of service-performing ASes to choose from. If $x\%$ of ASes support a service, but only $y\%$ of them are in edge networks, those without constraints have $(x - y)\%$ of ASes to draw upon that edge constrained requests can not. Second, and more importantly, diversity occurs at the center of the path, not at the endpoints. A service which is required 'near source' is either in the source AS, or in its provider - which for 36% of networks in our dataset is only one AS. Thus, a large fraction of 'near source' queries have only two, fixed ASes to access services from. On the other hand, a service which has no location constraints, or is expected near the middle of the AS path, could be one of a much larger number of well-connected Tier-2 or Tier-1 networks.

In summary, service availability drops for more demanding service types, as would be expected given the constraints of partial deployment. However, service availability can be significantly improved even for more demanding service types *provided* they tolerate some path inflation and/or allow some flexibility in where service-providing ASes may be located.

6.5 Service Stability

The previous aspects of our evaluation focused on availability of services given static paths. Yet, BGP updates and path withdrawals may remove paths which previously traversed service-providing ASes. Thus, a FISP providing access to a service interface in an external network must monitor updates to ensure that the traffic continue to traverse appropriate ASes, or inform the user that the service is no longer available.

Methodology. Our simulator does not model updates, since it requires only basic input (a relationship-annotated AS graph) and generates static paths for all ASes. Instead, we used real updates as observed by RouteViews [7] monitors to evaluate how BGP updates and withdrawals would impact service usage. We selected five ASes monitored by RouteViews and loaded their prefix announcements. Then, acting as a single-homed customer of that AS, we randomly chose an AS on the path to every prefix and selected that AS as a 'service' AS. For 24 hours, we monitored updates and withdrawals announced by the provider AS and logged when the paths to each prefix changed, and when those changes resulted in the path no longer traversing our randomly selected service AS. As single-homed stub ASes represent 36.3% of ASes in our topology snapshot, we believe the single-homed scenario to be a common case. In addition, we believe this common case to be a worst-case scenario. Because ASes with multipath peers have multiple options locally for path selection, even if their default path revokes a path traversing their service AS they are more likely to have an alternate route which traverses the service AS.

Results. Figure 8 shows the fraction of prefixes over the course of a day which experienced updates or withdrawals that removed the service AS from the path to the desired prefix. While over the course of the day, about 10% of the prefixes experienced some update, only about half of those updates removed the service AS from the path to the prefix. In the worst case, AS 3277 (Regional) had 5.7% of prefixes experience updates over the course of a day which removed the service AS from the prefix path. The best case, AS 3356 (Tier 1), 3.7% of prefixes experienced such changes.

This reflects an upper bound on service reconfiguration required by an AS and user. Just because a service AS is no longer available in the default path does not mean that a similar path is unavailable as a backup path from the provider. Further, measurement studies show that the most commonly used paths are generally stable, and that ASes experiencing frequent updates are less popular [27]. Thus, we consider the overhead of updating service state in case of path failure to be manageable.

7. RELATED WORK

Programmable Network Architectures. Active Networks [30] was perhaps one of the earliest proposals for a richer interaction between users and the network infrastructure. However, as should be clear, our proposals are very different. Where active networks allowed users to program their own services directly into the network, we rely on ISPs to deploy new services and only allow users the freedom of *invoking* these services—a more limited but also more tractable approach. The recent Rule-Based Forwarding[26] (RBF) proposal is closer in spirit to NetInvoke. In RBF, users write simple rules that are executed by routers. Like NetInvoke, RBF thus aims for a richer user-to-network interaction and allows rules to invoke in-network functionality installed at routers. However RBF does not address the question of how users and ISPs discover and select these functions. There are also several fundamental design differences: *e.g.*, NetInvoke's design builds in the means for ISPs to discover and negotiate service use in a policy-compliant manner. In RBF, how policy compliance is negotiated is external to its design and hence RBF requires per-packet policy verification on the

datapath and a PKI to generate these certificates. More generally, RBF represents a clean slate replacement for IP while NetInvoke only augments IP, resulting in different tradeoffs as described in Sec. 5.

Integrating Middleboxes. We discussed research efforts on integrating middleboxes [10, 29] in § 1. Within the IETF, the Middlebox Communication Working Group [28] is establishing standards for communication directly with local and explicitly-addressed middleboxes. However, this communication layer does not provide the Internet-scale discovery and federation of NetInvoke. Finally, Typed Networking [25], recognizing that end hosts may not wish certain processing to operate on their traffic, envisions a 'negotiation' between middleboxes and end hosts. They propose an architecture where middleboxes encountered along a user's regular forwarding path signal the user in order to allow the user to opt out of such processing. This does not address an opt-in capability and hence the discovery, federation and other capabilities NetInvoke seeks to enable.

Programmable Routers. NetInvoke also relates to recent research on building programmable network infrastructure[9, 15, 13, 24, 21]. These recent efforts focus on the router-level architecture in support of such programmability while NetInvoke tackles the network-level question of how the capabilites enabled by such programmable infrastructure should be exposed to end systems and applications. In this sense, our efforts are complementary.

Tussles and innovation. Middleboxes and in-network services more generally represent a potential 'tussle' point as first articulated by Clark *et al.* [12]. We take inspiration from their discussion and aim for a design point that balances control between both users and networks: users have the control to opt in/out of advanced processing while networks can choose what services they offer and to which users. Finally, our work is of relevance to recent community discussions on enabling network innovation. Our proposed approach however, is more conservative than most—NetInvoke is not a clean-slate architecture, nor does it offer solutions to hard problems such as securing the network layer or improving its reliability. Instead, we propose a path based on augmenting IP to enable incremental upgrades to the capabilities of the network infrastructure.

8. CONCLUSION

We presented NetInvoke, a framework that allows user applications to program against high level interfaces to in-network services. NetInvoke provides a suite of interfaces and algorithms that allow users to express their service demands while giving networks control over whether and how these demands are met.

We do not by any means expect that NetInvoke is the final say in discussion of how to best integrate advanced network processing into the network architecture. However, we believe our contribution - a vision for high-level, programmable interfaces that provide access to federated services across the entire network - moves the space forward towards a practical design that is easy to use from the perspective of application developers, while providing network providers a stake in selection, deployment, and profit from advanced services.

9. REFERENCES

[1] Astaro: Security gateway. `http://www.astaro.com`.
[2] CAIDA AS relationships. `http://www.caida.org/data/active/as-relationships/`.
[3] Narus: Real Time Traffic Intelligence. `http://www.narus.com`.
[4] Riverbed: Application Acceleration. `http://www.riverbed.com`.
[5] Sourcefire: Network Security. `http://www.sourcefire.com`.
[6] Symantec: Data Loss Protection. `http://www.vontu.com`.
[7] University of Oregon RouteViews Project. `http://www.routeviews.org`.
[8] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. *Computer*, April 2005.
[9] M. B. Anwer, M. Motiwala, M. bin Tariq, and N. Feamster. Switchblade: A platform for rapid deployment of network protocols on programmable hardware. In *SIGCOMM*, 2010.
[10] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *SIGCOMM*, 2004.
[11] B. Carpenter and S. Brim. RFC 3234: Middleboxes: Taxonomy and Issues, February 2002.
[12] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: defining tomorrow's Internet. *IEEE/ACM ToN*, 13:462–475, June 2005.
[13] M. Dobrescu, N. Egi, K. Argyraki, B. gon Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *SOSP*, 2009.
[14] Facebook. Facebook api. `http://developers.facebook.com/`.
[15] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: A GPU-accelerated software router. In *SIGCOMM*, 2010.
[16] M. Handley and A. Greenhalgh. Steps towards a DoS-resistant Internet architecture. FDNA, 2004.
[17] J. P. John, E. Katz-Bassett, A. Krishnamurthy, and T. Anderson. Consensus routing: The Internet as a distributed system. In *NSDI*, 2008.
[18] E. Katz-Bassett, H. V. Madhyastha, V. Adhikari, C. Scott, J. Sherry, P. van Wesep, T. Anderson, and A. Krishnamurthy. Reverse traceroute. In *NSDI*, 2010.
[19] S. Kent, C. Lynn, J. Mikkelson, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 2000.
[20] N. Kushman, S. Kandula, D. Katabi, and B. M. Maggs. R-bgp: Staying connected in a connected world. In *NSDI*, 2007.
[21] G. Lu, C. Guo, Y. Li, Z. Zhou, H. Wu, Y. Xiong, T. Yuan, R. Gao, and Y. Zhang. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI*, 2011.
[22] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *CCR*, 2001.
[23] Z. M. Mao, J. Rexford, J. Wang, and R. H. Katz. Towards an accurate AS-level traceroute tool. In *SIGCOMM*, 2003.
[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *CCR*, March 2008.
[25] C. Muthukrishnan, V. Paxson, M. Allman, and A. Akella. Using strongly typed networking to architect for tussle. In *HotNets*, 2010.
[26] L. Popa, N. Egi, I. Stoica, and S. Ratnasamy. Building extensible networks with rule-based forwarding (RBF). In *OSDI*, 2010.
[27] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP routing stability of popular destinations. In *IMW*, 2002.
[28] M. Stiemerling, J. Quittek, and T. Taylor. RFC 5189: Middlebox communication (MIDCOM) protocol semantics, March 2008.
[29] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, 2004.
[30] D. Wetherall, U. Legedza, and J. Guttag. Introducing new Internet services: Why and how. In *in IEEE Network*, May/June 1998.
[31] R. White. Architecture and deployment considerations for secure origin BGP (soBGP). `http://tools.ietf.org/html/draft-white-sobgp-architecture-02`, IETF Draft (expired), 2006.
[32] W. Xu and J. Rexford. MIRO: multi-path interdomain routing. In *SIGCOMM*, 2006.
[33] X. Yang, D. Wetherall, and T. Anderson. TVA: a DoS-limiting network architecture. *ToN*, December 2008.