Netcalls: End Host Function Calls to Network Traffic Processing Services

Justine Sherry
Daniel C. Kim
Seshadri S. Mahalingam
Amy Tang
Steve Wang
Sylvia Ratnasamy

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-175
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-175.html

July 12, 2012

Netcalls: End Host Function Calls to Network Traffic Processing Services

Justine Sherry *Daniel C. Kim* *Seshadri S. Mahalingam*
Amy Tang *Steve Wang* *Sylvia Ratnasamy*

Abstract

Function calls are a basic primitive by which applications invoke services from external entities. In this paper, we propose "network calls" (netcalls), a general primitive to invoke advanced traffic processing services – such as firewalling or caching – from the network. We design and implement the netcalls API and a backend architecture to support netcalls, allowing end host applications to interact with services not only in their own access network, but any network their traffic traverses. Demonstrating the utility of netcalls, we built three applications to invoke netcalls, along with corresponding network services: interdomain firewalling for DDoS defense, 'opportunistic' traffic compression, and intrusion detection for mobile phones.

1 Introduction

Function calls are a basic primitive by which applications invoke services from external entities. For example, applications make syscalls to the kernel, and remote procedure calls to network servers. In this paper, we propose "network calls" (netcalls), an equivalent primitive to invoke advanced traffic processing services from the network. Processing services in use today include firewalls, intrusion detection systems, proxies, and caches. These devices are typically transparent to end hosts, in that the end hosts cannot communicate with these devices nor are end host applications even aware of their presence.

However, there are cases today where application input or control of network services would be valuable but there's no easy way to do so. For example, we updated three applications to use netcalls to invoke advanced traffic processing services deployed on our testbed: a web client that invokes traffic compression for large downloads, a webserver whose QoS/load monitoring module initiates DDoS defense, and an Android WiFi interface that preferentially connects to networks that deploy intrusion detection services tailored to mobile phones. Today, it is not possible to deploy applications like these.

We anticipate a growing need for traffic processing services due to the trend towards lightweight client devices and growing prevalence of cloud-based services. Such clients will increasingly rely on applications offloaded to run either entirely or partially in the cloud. Consequently, the network's impact on user experience grows and hence so does the role for network optimizations and new services, *e.g.* low latency paths to datacenters, or network caching in cooperation with cloud services. In addition, we speculate that having an API that application developers can write to will be of interest to network operators interested in monetizing their deployment of new features.

In the netcalls design, we take a broad view of what constitutes "the network": invoked services may be implemented by the invoking client's local-area network, the networks of the endpoints the client communicates with, intermediate ISP networks, combinations of the above, and so forth. Within the network, services may be implemented in on-path switches and routers, servers embedded in the network, in the cloud, *etc.* However, netcalls hides this complexity from the application developer, instead presenting a simplified abstraction of a single logical network with a capability that can be turned on, off, or configured. For example:

```
fw = BasicFirewall.init(...);
fw.block("9.8.7.0/24");
```

To application developers, the abstraction of a single logical network is appealing for its simplicity – the client only specifies what service it wants invoked. For operators, this abstraction is appealing since they don't have to expose the details of their network topology or routes, and they have flexibility in how they implement a service.

Nevertheless, implementing this abstraction is challenging because the logical network is in reality a large-scale federation of independent networks each with their own policy and deployment goals. Particularly challenging is that, in such an environment, it is unrealistic to expect the universal adoption of any service, or even of the netcall mechanisms we propose. Instead, we must assume some networks will implement a particular service while others may choose not to. Likewise, some networks might implement the netcall solutions we propose, others not. Thus any solution we devise must be compatible with the characteristics of a federated environment – scale, distributed control, settlements through bilateral relationships, *etc.* – while operating under the constraints of partial deployment. As we elaborate on in the following section, these requirements complicate notions of service discovery, availability and invocation.

The contribution of this paper is the design, implementation, and evaluation of the netcall API and the "backend" netcall architecture that implements the infrastructure needed to support this API. In the rest of this paper, we describe the netcalls API and supporting architecture(§2-§5) and the implementation of netcall

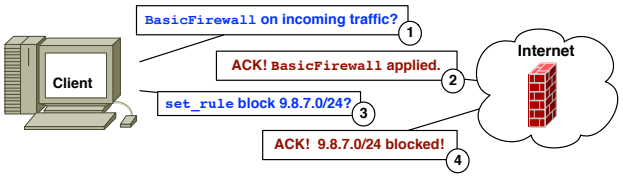


Fig. 1: A client invokes a firewall with netcalls.

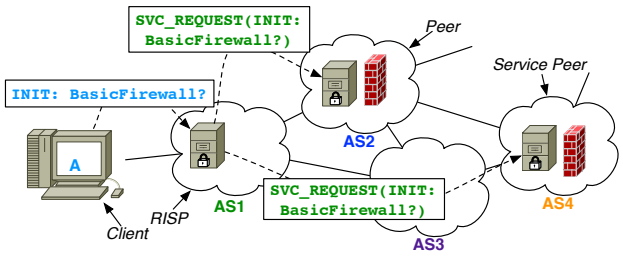


Fig. 2: A Resolution ISP initializes an interdomain service.

services and end host applications (§6). We then evaluate (§7), discuss related work (§8), and conclude (§9).

2 Overview

To illustrate the netcalls API and backend architecture, we begin with an example; we subsequently discuss our key design decisions.

Figure 1 shows a client application using netcalls to initialize a firewall, labeled BasicFirewall. First (1,2), the client initializes the firewall, after which the network will redirect all of the client's inbound traffic to traverse a middlebox, server, or programmable router that can perform firewalling. Then, in (3,4), the client configures the firewall, adding a new rule to its access control list; the firewalling devices update themselves accordingly. After these two steps, all traffic destined for the client will traverse firewalls and be filtered according to the client's demands. However, the client application is never concerned with which middleboxes process its traffic or where its traffic is routed; the client's abstraction is only of a single logical network with a named capability that can be turned on or off, or configured.

Before discussing any technical mechanisms behind this process, we assign terminology to Fig 1. We refer to named capabilities like BasicFirewall as *services*, identified by a *service name*. Clients use netcalls to *initialize* and *configure* services. Initialization (the first step in Fig. 1) refers to instructing the network to redirect the client's traffic to a device that performs the requested service. Configuration occurs in Fig. 1 with (3), when the client adds a new rule to the firewall. We refer to 'configuration' when clients specify service-specific customization, *e.g.* adding a rule to a firewall, specifying a cache timeout policy for a proxy, or requesting summary statistics from a traffic monitor. The configuration step is service-specific in that it would not make sense to specify a cache timeout policy to a firewall, or add a filtering rule to a traffic monitor. Hence, each service name is associated with a *service protocol* – a set of functions defined independently for each service. For BasicFirewall, `add_rule` is a function contained within its service protocol. One final step not illustrated in Fig. 1 is *invocation*, when the client's packets traverse a specialized device in the network and trigger service processing.

Returning to our example, we now look to the network's perspective of initializing the firewall, shown in Fig. 2. When the client initializes BasicFirewall, it sends a request to a server hosted by its local ISP. Since the local ISP does not deploy middleboxes or servers to perform BasicFirewall functionality, it must forward the request to one or more external ISP(s) capable of performing BasicFirewall. The server then consults an autonomous system (AS) level graph of nearby networks, and selects two external ISPs who together serve all of the client's inbound traffic. It forwards the client's initialization request to these two external ISPs, and after their acknowledgments, it stores a record of the client's request and the selected ASes. The ISP then acknowledges success to the client. The ISP must store a record of the client's request in order to handle any future requests from the client. For example, when the client requests to add a rule to the BasicFirewall (shown in Fig 1 but not Fig.2), the ISP must be able to recall which external networks are providing service for the user, and then forward the client's request to them.

We refer to any network that deploys a service like BasicFirewall as a *service network*; in Fig. 2 ASes 2 and 4 are service networks. AS 1, on the other hand, serves as a *resolution ISP* or RISP; we refer to the server the client directs its requests to as a *resolution server*. RISPs are responsible for taking in client requests and then performing *discovery* (finding out which networks support the client's requested service), *resolution* (selecting an appropriate set of networks to perform the service), and *implementation* (either updating the local network to perform the service, or forwarding the client's request to external networks who will then perform the service). We assume that each client contracts with one or more networks to serve as its RISP. We say 'contracts with' because we expect that providers will charge for access to services. Correspondingly, when a RISP forwards a request to another service network, there must be a contractual relationship in place between the RISP and the other network. Service networks that a RISP settles with are called the RISP's *service peers.*

Design Decisions

A core goal in each of our design choices is to adapt to *partial deployment*: the assumption that no service will be adopted by 100% of networks and hosts, nor even will

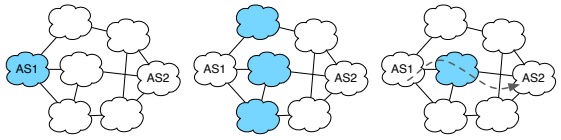
support for netcalls be fully adopted.

RISPs. As described above, clients only convey netcalls to a resolution server hosted at a single network, the client's RISP. We chose this model for two reasons. First, it simplifies application logic. Consider a client invoking some service S that it requires once, on the forwarding path from itself to a destination D. Where are there devices that can perform S? How does D direct its traffic to these devices? By making these questions the responsibility of a RISP, application developers write logic to request services, but don't have to write logic for discovering where and how services are performed in the network. Second, contacting only a single RISP simplifies the process of payment/settlement. Client payments for interdomain services require a contract with a single ISP, rather than several (just as they pay for Internet service today). ISPs maintain a contract with clients (just as they do today) and *service peers*, networks who they have business relationships with for services (just as they have peering relationships for traffic exchange today).

API Design. Directing traffic through a RISP leads to a new challenge: how is the RISP able to resolve requests to services that it does not support itself? For example, in resolving a users initialization request, how is the RISP to know that a firewall should be applied once on inbound traffic, but that a WAN optimizer should be deployed twice at the endpoints of communication? We resolve this by adding a set of parameters called a *placement pattern* to a clients initialization request: placement patterns are a general abstraction by which clients describe where in the network to place the service. We describe placement patterns in depth in §3.

Service Invocation. Netcalls separates service *initialization* from *invocation*: services are initialized when clients request them from their RISP, and processing is invoked when the client's traffic traverses a device supporting the service. This model allows netcall clients to continue to interact with – and invoke services on traffic to and from – other end hosts who have not adopted netcalls. In a traditional model [34, 27, 33], packets destined for service processing contain a new header describing their service demands. Embedding a service-specific header means that the *sender* must have adopted the service; if the sender does not support the new service, it cannot be used. However, for services like BasicFirewall the receiver is the endpoint that wants service processing, not the sender. With netcalls either the sender or the receiver can request processing features; it is not necessary for both hosts to adopt netcalls for one of the hosts to initialize services independently.

Best Effort Availability. As a fundamental consequence of partial deployment, netcalls adopts a service model of *best effort availability*. A client application may request a service that is not supported by any nearby network, in which case the network will return an error informing the client that the service is unavailable. No architecture can compensate for a service the network simply doesn't offer. As a consequence, applications must be prepared to *adapt* to service unavailability. For example, in Section 6 we'll show a mobile phone that turns on extra anti-virus when connecting to a network that *does not* support an intrusion detection system.



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

Fig. 3: INIT request Placement Patterns.

3 API

We now describe the netcalls API in detail. As mentioned previously, netcall services represent standardized protocols accompanied by an identifier (a service name). Clients sent their netcall requests to a *resolution server* which exposes five basic functions to its clients, that together form the *client API*. Clients discover their resolution server via DHCP or manual configuration. While our discussion treats the resolution server as a single entity, a large RISP may use standard approaches for replication to achieve scalability and reliability.

Beyond exposing the API, an additional responsibility of the resolution server is *enforcing policy*, first, in the types of services it allows clients to invoke, and second, in to whom services apply. An example of the former is an enterprise network which hosts a resolution server; the enterprise's server might drop all requests that attempt to manipulate firewall settings. The latter simply means authenticating clients and enforcing that clients can only invoke services for their own traffic.

We now discuss the client API's five functions, focusing on the INIT function, which is the core of the API.

(1) INIT. The core function in the client-RISP protocol is INIT, which allows the client to instantiate a service. In addition to the service name, INIT messages specify *what* traffic the service should apply to, and *where* the service should be performed. As discussed in §2, this is specified through a *placement pattern*, a general model that captures how and where the network should apply the service. We identify three key placement patterns (see Fig. 3) that we believe allow us to represent a broad set of service requirements:

(i) 'Individual', in Fig. 3a, allows a client to specify directly an IP address or ASN; the resolution server then contacts that network directly. Individual requests have no other parameters.

(ii) 'Perimeter', in Fig. 3b, constructs a border of service-performing networks between the requester network and an external address. Perimeter INIT requests

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 client should traverse the service.
User/External Port	Port numbers of traffic that should traverse the service (0 for all port numbers).
Incoming, Outgoing, or Bidirectional	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 client's network, required in the external prefix's network, should be place near the clients network, or near the external prefix's network.

Table 1: Parameters for 'Path' INIT requests allow the requester to describe their service's topological requirements.

use the following 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.
Single Instance: Only one network may construct the perimeter. By default, the RISP may resolve the clients request to multiple networks in order to construct a complete border. However, some services such as intrusion detection require a complete view of *all* traffic.

(iii) 'Path', in Fig. 3c, 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. These parameters allow the client to convey (for example) that a flow receive transcoding once, anywhere along the path; or that a flow receive a dedicated amount of bandwidth at every router across the network. Table 1 shows the list of parameters, which we drew from previous efforts at taxonomizing middlebox behavior [14] and our own analysis of usage scenarios.

Given a placement pattern, the RISP uses its knowledge of network topology acquired from BGP routes and its own peers to select a set of networks that are topologically appropriate and with which the RISP has settlement agreements. The RISP then queries these networks for their willingness to service the client's request (using the protocol described in §4). On receiving a positive response, the client's RISP creates a unique service *token* and stores a *service resolution record* (SRR) that maps from token to the client name, INIT parameters and ASes assigned to perform the service. The RISP then returns the token to the client; if the resolution server cannot resolve the INIT request, it returns an error. Because the RISP will serve as the client's only gateway to the service – the client does not keep track of the specific middleboxes or networks performing his service – the resolution server must keep the SRR in local state in order to forward update messages (*e.g.* to kill the service once no longer needed) to the appropriate external networks.

The four remaining client API functions are:

(2) CONFIG. After the RISP enables the service, the client may need to invoke subsequent service-specific as part of a service-specific protocol. With a CONFIG message, the client specifies the relevant service token and an opaque field within which the client places the desired service-specific messages (for example, `set_rule: block 9.8.7.0/24` for a firewall). On receiving a CONFIG message, the RISP maps the client token to the corresponding set of services ASes and forwards the client's message to these ASes; the RISP does not need to understand any service-specific contents.

(3) KEEPALIVE. This message extends the lifetime of the service, as the RISP periodically culls long-living INIT state past its expiration date.

(4) TERM. Terminates the service.

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

Before moving on, we note that the client interface can be used by different types of 'client.' For example, an enterprise might register with a RISP for services on behalf of its entire network. INIT requests allow the enterprise administrator to specify requests on behalf of a prefix, thus setting a single policy on behalf of the whole subdomain. In this case, the RISP would recognize the credentials of the enterprise networks administrator as overriding the requests of other end hosts within the enterprise, *e.g.* not allowing hosts to disable a firewall.

4 Services and RISPs

We now discuss the features of a RISP supporting netcalls: a *network-to-network API* to communicate with service providing networks (§4.1), a set of algorithms for *service discovery and resolution* to choose which networks should support each user's request (§4.2), and optionally, *availability extensions* which broaden access to interdomain services.

4.1 Network-to-Network Protocol

Just as ISPs expose a resolution server with an API for their customers, ISPs also expose a server and API to other networks. For simplicity, we refer to the network-to-network server as a resolution server as well, although it need not be the same entity as the resolution server exposed to clients. With the network-to-network API, ISPs (1) allow external networks to initialize and configure services and (2) share information to help external networks select appropriate networks for their client's requests.

To allow networks to express their requests for services, the network-to-network protocol provides a simple SVC_REQUEST interdomain function that encapsulates a client's INIT, CONFIG, and TERM requests.

For an ISP to accept a request to invoke a service, however, we must ensure that inter-network actions are both *authoritative* and *accountable*. *Authority* means that requests are verified to originate from an entity with ownership of the impacted IP address or prefix. *Accountability* means that for any service provided, the network always has means to charge for use of the service. Our demand for accountability does not reflect any particular payment model; it only guarantees that there is some network in place to hold responsible for payment so that if networks wish to develop billing models, they may.

These requirements are far from onerous, as peering relationships are sufficient to verify both authority and accountability. As peers, the networks exchange their public keys and allocated prefixes out of band. Networks which are not physically connected may still peer over services alone. A network AS2 can verify the authenticity of AS1's requests 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 and it already 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.

An additional network-to-network function is the INFO_REQUEST function that allows an ISP to inform other networks whether or not it supports a service, tell a network for whom it is performing services whether or not a service is still active, announce who its AS-level neighbors are, and share its AS-level forwarding paths. As we'll show in the following section, this information can help external resolution servers to select appropriate networks to perform services for their customers.

4.2 Discovery and Resolution

Before invoking services, a RISP must first *resolve* a client's request to a particular network that can appropriately provide the client's requested service. In this discussion, we show that resolution is reasonably achieved with limited information and basic techniques. However, we note that the particular data and algorithms we describe are not the only way to acquire the required information: a RISP may use more sophisticated methods, which would only improve netcalls's capabilities.

Individual Request. We start with a simple Individual INIT query: a client u requests that its traffic receive the `EasyTranscode` service in some IP address v's AS. If the RISP peers with the external AS, the resolution server queries the AS's resolution server directly to discover whether or not it offers `EasyTranscode`. The RISP may cache the results of the query to avoid requerying in the future; networks are unlikely to frequently deploy or withdraw services. If the AS supports the services, the RISP sends a SVC_REQUEST message and after returns a service token to the client. If any of these steps fail, the RISP returns an error to the client.

Perimeter Request. 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. First, u's RISP itself might support the service, in which case the tightest perimeter around u is to invoke the service locally. If not, and if u allows the perimeter to extend outside the local AS, the RISP can look to its physically connected peers and invoke the service at its neighbors. If one of its peers does not support the service, it can send the peer an INFO_REQUEST for a list of their peers, and so on[1].

Path Request. To resolve a Path INIT request, u's resolution server needs to know the AS-level paths for traffic both to and from u. Obtaining the path u takes en route to an external destination is straightforward: BGP tables provide an AS Path for every globally announced prefix. For traffic destined to u from some external source v, the resolution server can send an INFO_REQUEST to the resolution server at v's ISP, requesting the AS path. In the absence of a resolution server at v's ISP, u's resolution server can issue a reverse traceroute [19] to measure the path. Once the RISP discovers the relevant path, it can then look up the resolution server for each ASN, query them for whether they support the requested service, and decide whether the path fits the clients 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. Thus the RISP must monitor service perform-

[1] Because the AS graph fans out quickly, in practice it is typically only feasible to invoke services at the local ISPs, or at the local ISP's peers; two hops or more away from the local ISP and one would face invoking services in potentially hundreds of networks

ing paths to ensure they continue to meet the client's demands. While the client application and service interface are responsible for monitoring the service functionality itself (*i.e.* is it performing its duties correctly), the RISP is responsible for ensuring that the client's traffic continues to traverse the network assigned to the service. Our implementation monitors INIT configurations assigned to forward, default paths by watching BGP path updates, periodically issuing traceroutes, and requesting updated AS paths from resolution servers in peer networks. If the path ceases to traverse the service-performing AS, the RISP reports an error to the client.

4.3 Extensions for Improved Availability

Our basic design allows clients to invoke services in any network which their RISP peers with and which their traffic traverses. We now describe extensions which networks can optionally support to expand availability.

Service brokers. Our discussion so far has assumed a RISP only negotiates with service-providing ASes with which it has a direct agreement regarding settlements. While simple, it can be unrealistic for a RISP to maintain service peering relationships with a large number of other networks and this might lead to low availability of services. Brokers address this concern. If a RISP A wishes to invoke a service in an AS C with whom it does not (service) peer, but both A and C are peers of a third AS B, then AS B may serve as a 'broker' for the exchange between A and C.

Multipath routing. This optimization aims to improve availability for 'Path' services in the event that the default AS path does not include an appropriate service-providing network. In such cases, the RISP may provide increased availability by considering alternate policy-compliant AS paths. We chose MIRO [35] as our multipath routing solution because it is simple, backwards-compatible with BGP and functions through bilateral agreements in a manner that does not require the participation of every AS on the path.

Remote RISPs. To expand its a customer base (and provide service access to clients whose ISPs are not netcalls-aware), an ISP may serve as a RISP for end hosts who receive connectivity service from another network. The primary challenge for such a RISP is how to discover the client's paths and nearby topology in order to resolve the client's request. If the client tunnels its outgoing traffic via the RISP (if the RISP is topologically close) then the RISP can use the same strategy as it does for its direct customers. Otherwise, the RISP discovers outgoing paths as it does incoming. It can also leverage active measurement from the client.

5 Designing Secure Services

A 2009 study found that 34% [11] of networks permit their users to send traffic with any source address, not just their own. Exploiting this, an attacker might attempt to spoof traffic in order to avoid service processing, or to receive service processing when it otherwise wouldn't. Hence, services must ensure *data plane security*: that services only process traffic belonging to the real client.

Before delving in to technical solutions, we briefly note that the problem of attacks inflating service charges is analogous to attacks on cloud services. Cloud providers deal with this threat not only through technical means, but through practical pricing schemes; for example, by placing caps on volume processed before shutting down, or by providing logging mechanisms to allow customers to dispute false charges. Thus, while we present technical solutions below, we expect that providers will pair these with responsible business practices.

We imagine the following attack scenarios,[2]

in which an attacker A uses spoofed traffic to manipulate a service invoked by a user B:

(1) To avoid undesired processing, *e.g.* security services, A sends traffic to B appearing to originate from an address B considers benign and has no security rule for. In addition, if multipath routing is available (§4.3), A may try to route around the service.

(2) To inflate B's volume-based service charges, A sends traffic appearing to originate from B or en route to B spoofed as someone B frequently communicates with.

(3) To steal service processing, A sends his own traffic spoofed as B to another endpoint he colludes with.

We present four solutions which address the above three attacks:

Exhaustive Service. Inbound services which the sender might find undesirable (primarily security applications) should be applied to all inbound traffic with a perimeter request. Under this INIT pattern, there is no way to avoid processing. We find that this security policy parallels security policies enforced by typical enterprises: all traffic traverses firewalls, no matter the origin. This resolves attack (1).

Registered Flows. A baseline solution for attacks (2) and (3) is to require that applications "register" every connection. Under this model, services expose a registration function under their service interface that allows a user to register IP addresses, port numbers, and initial sequence numbers for a flow they want processed; thus an attacker cannot invoke the service without knowledge of both endpoints and the valid sequence numbers for the connection. This solution imposes registration latency on

[2] While we spend this section focusing on spoofed traffic, it is worth calling out the attacks we do *not* address. First, we do not resolve Denial-of-Service, although some of our envisioned services could be used to mitigate DoS attacks (*i.e.* firewalling services). DoS is a long-standing problem in the Internet architecture; our control plane neither worsens nor improves this state. Second, we do not consider man-in-the-middle attacks; we assume that users trust their ISPs.

every flow a user wants processed – the following two solutions avoid this overhead.

Connection Termination. For processing services that 'terminate' a TCP session on bilateral traffic (*e.g.* WAN Optimizers or load balancers), spoofing attacks are not possible. The middlebox maintains connections to (and hence completes a handshake with) both endpoints, ensuring that neither endpoint is spoofed. This solution can resolve attacks (2) and (3), but not for processing services that service TCP bilateral sessions without terminating them, or observe only one direction of traffic.

Shared SYN Cookies. For general TCP security, we introduce Shared SYN Cookies, which leverage TCP SYN cookies to validate flows. Traditional SYN cookies resolve "SYN Flood" attacks by allowing a server to identify ACK packets for flows it previously SYN/ACKed, without having to keep state for the incomplete handshake. The server sends SYN/ACK replies to SYN requests with specially crafted sequence numbers; each sequence number includes a coarse timestamp, an MSS value, and a cryptographic hash of the connection's IP addresses, port numbers, and the timestamp. When it later receives an ACK in reply to its SYN/ACK, it inspects the (Seq. Number - 1) value in the ACK and uses the hash to validate that it represents a valid connection.

Shared SYN Cookies are generated using the same technique as normal TCP SYN Cookies, but the user shares its key for generating the SYN cookie hash with the middleboxes processing its traffic. Then, the user generates sequence numbers for all of its connections (both those it initiates and those initiated by another user) using SYN cookies. This allows the middlebox to validate even unilateral traffic in a TCP session. Consider a connection between our user B and another host, C. For traffic that appears to come from B, the middlebox inspects the SYN or SYN/ACK where B announces its initial sequence number and validates that the hash value originated with B. For traffic that appears to come from C, the middlebox inspects the SYN/ACK or ACK where C replies to B's initial sequence number, and similarly validates the hash. After this, it can establish an entry in its flow table for the session. Shared SYN Cookies safeguard against attacks (2) and (3) without any session-specific setup nor middlebox modification to the flow.

6 Implementation

We prototyped the system components of the netcall architecture and developed three applications that use netcalls to leverage network services. We now describe the components of our implementation: our prototype *resolution server* (§6.1), the client-side libraries that implement the netcall API (§6.2) and our three prototype applications (§6.3).

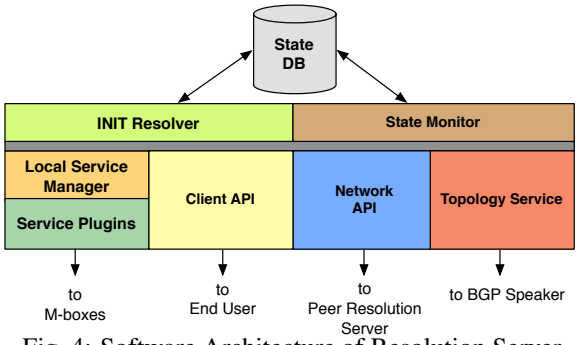


Fig. 4: Software Architecture of Resolution Server

6.1 Resolution Server

Every network that exposes a netcall API deploys a resolution server. Our resolution server is a 4200-line Java web server exposing an RPC interface to end host clients and to other resolution servers.We illustrate the software architecture of the resolution server in Fig. 4.

The *State Database* stores: (i) a list of the clients for which the network serves as RISP, (ii) a list of the network's service peers, with the IP addresses of their resolution servers and the names of the services they offer, and (iii) SRRs for each client request that the network is currently servicing. The *Topology Service* maintains a model of the AS graph and routes which it obtains by combining local BGP routes, the network's BGP peers, and INFO_REQUEST responses from other networks. The *Client API* and *Network API* modules implement the client-to-RISP and RISP-to-network protocols. The *State Monitoring* module monitors BGP updates for route changes that impact SRRs in the state database and updates these appropriately. If a BGP update disables a service by routing away from a service-performing AS, the state monitor sends an error to the user.

The *Local Service Manager* and *Service Plugins* are used to configure the local network to provide traffic-processing services. Our implementation currently assumes that – akin to typical enterprise networks – services are implemented by middleboxes and these middleboxes are deployed adjacent to a switch at a network choke point. For our prototype deployments, we use Click-based [20] software switches and software middleboxes that we run on a general-purpose server. The Local Service Manager maintains a Service Plugin for each service supported by the network. Each plugin implements a standard interface with functions to call for INIT, CONFIG, and KILL requests. When an INIT, CONFIG, or KILL request reaches the Local Service Manager, it calls the appropriate function in the plugin for the service and the plugin communicates with the switches and middleboxes as appropriate. On receiving an INIT request, the plugin for the requested service updates the switch to divert traffic matching the INIT request to the middlebox in question. On receiving a CONFIG request, plugins either forward CONFIG requests to their associated middlebox,

```
import org.netcalls.API.*;
public class CompressSvc{
    ...
    InitPathRequest p = new InitPathRequest(
        localIP, dstIP, localPort, dstPort,
        COMPRESSION_SVC_ID,
        PathParams.BIDIRECTIONAL,
        PathParams.ENDTOEND, ...
    );
    ServiceToken t = ServiceManager.init(p);
    ... //configure service
    return t;
}
```

```
import org.netcalls.API.*;
import org.netcalls.CompressSvc.*;
...
ServiceToken t = CompressSvc.init(ip, port);
... //send data
sock.close();
ServiceManager.kill(t);
```

Fig. 5: Application developers program against standardized libraries to invoke services.

or they implement some functionality at the Resolution Server itself. We provide examples of different plugins for our prototype applications later in this section.

6.2 Netcall Clients

A client discovers its local resolution server via a DNS request to the resolver it is assigned by DHCP. We implemented the netcall protocol over XMLRPC, since it is simple to program against in many languages. The raw RPC interface is not exposed directly to application programmers. Instead, application developers simply import a library and invoke service-specific functions like `initialize_firewall()` or `filter_ip_address(ip)`; the library then generates the appropriate INIT and CONFIG requests. In Figure 5 we show an example of the code implementing the library (top), and the code implemented by the application developer (bottom). As a consequence of these libraries, extending an existing codebase to leverage network services is relatively simple as we illustrate with the applications we describe later in this section.

Our implemented APIs allow any application to invoke any service for its host – even for ports bound other applications. While the Resolution Server stops malicious end hosts from interfering with the traffic of other end hosts, it cannot detect malicious processes interfering with other processes on the same end host. Longer term, we expect that OS support for the protocol will centralize netcall requests through a single, privileged process. Applications, rather than communicating directly with the resolution server, will instead submit their INIT request to the OS, who will either reject the request or accept it and forward it to the resolution server, based on (1) the port numbers bound to the requesting process, and (2) the privileges of the requesting process.

6.3 Applications and Services

We modified three applications to invoke network services through netcalls: (i) an Apache webserver that invokes in-network filtering services for overload protection, (ii) a web client that invokes in-network compression services ("WAN optimizers") for reduced bandwidth consumption and latency and, (iii) the Android OS WiFi management interface that invokes in-network IDS for malware protection. In implementing these, we aimed to answer two main questions. First, do the netcall abstractions allow clients to express requests for a broad range of services? Second, are such services useful to end applications? We believe our experience with these services answers both in the affirmative.

DDoS Defense. The first application we developed is a webserver that invokes firewall services in remote networks when overloaded. We implement this service by extending an existing firewall implementation to allow clients to add new rules with a CONFIG request. Our netcall client is an Apache webserver that we modified to request filtering using network calls. For this, we extend the existing `mod_qos` [5] module in Apache, which detects overload and restricts access to the webserver based on usage patterns, redirecting troubling hosts/prefixes to a 'service overloaded' web page. We augmented `mod_qos` in 97 LOC to invoke firewalling close to the offending hosts' networks to prevent DDoS. Our integration with `mod_qos` demonstrates how application-layer context can beneficially inform network behavior.

Ideally, we would test our application by deploying resolution servers and services at every AS on the Internet; since we cannot do this in practice, we instead leverage EC2 as follows. We install our modified webserver along with its local firewall and resolution server in a testbed in our local environment. We then emulate the wide-area Internet topology over EC2 by having each EC2 node serve as a 'surrogate AS', installing software switches, firewalls and resolution servers at each. We then deployed web clients on 20 EC2 nodes; these clients serve as attackers and send unwanted traffic to our modified webserver. Figure 6 shows a time sequence of aggregate malicious traffic sent, malicious traffic at the switch in the web server's location, and malicious traffic reaching the webserver itself [3]. At 25 seconds, the now-overloaded web server invokes the firewall. At time 45 seconds, the web server invokes firewalls in remote networks. We see that neither the webserver nor its local firewall is ever overloaded, as firewalls deeper in the

[3]Obviously the request rates shown are not enough to overload a web server: we artificially set the bandwidth cap in `mod_qos` to be very low to avoid flagging the attention of network administrators for launching a real DDoS attack!

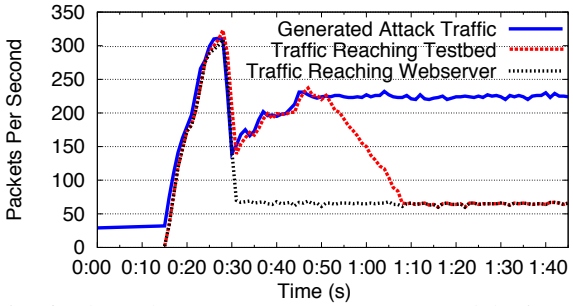


Fig. 6: The webserver reacts to a DDoS attack by invoking network firewalls.

network drop malicious traffic before it reaches the web server's site.

Traffic Compression. Enterprise networks today commonly deploy WAN optimization appliances that compress traffic to minimize bandwidth costs. WAN optimization requires that both the sending and receiving networks deploy appliances that compress/decompress traffic as needed and hence, today, such compression is typically limited to communication between the different geographic sites of a single enterprise. With netcalls' inter-domain discovery capability, clients can instead invoke WAN optimization services in any other network they communicate with. We modified an existing command-line web client to request compression whenever the user flags a particular request as a 'large file' download – *e.g.* downloading ISOs, video streams, *etc.* In parallel with the start of the file transfer, the web client issues an INIT request for an end-to-end service that compresses and decompresses traffic. This change required only 25 LOC. Our traffic measurements showed that downloading from another network which also deployed compression could reduce bandwidth for the connection by on average 27%.

We also investigated the benefits of invoking compression persistently to commonly contacted destinations. Running an enterprise trace through our testbed, we found that, for this enterprise, enabling WAN optimization to and from the ten most commonly contacted external ASes reduced the enterprise's *total* bandwidth utilization by 21%. Invoking compression to and from 100% of external networks would reduce bandwidth by 27% – hence, even partial deployment can provide substantial benefits for this service.

Android Security. Cell provider data networks protect smartphones by filtering malicious traffic, but typical WiFi networks offer no such protection. We designed a netcall-enabled IDS targeted towards Android smartphones, and modified the Android WiFi interface to preferentially connect to networks that deployed this service. Smartphones who invoke the service not only receive traditional firewalling and IDS, but a set of Android-specific filters for real malware [2].

Upon connecting to a new WiFi network – *i.e.*, one

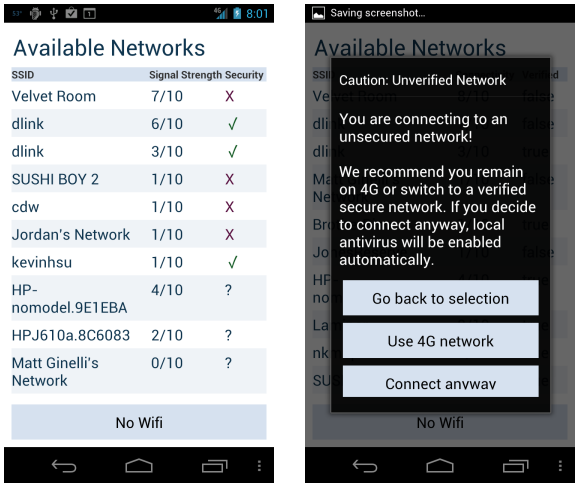


Fig. 7: An end host security suite on the Android device warns users when a security service is unavailable.

not secured by their service provider – the smartphone attempts to enable security features if available. During this process, the security application maintains network connectivity but bans all other applications from connecting to the network.To establish trust between the user and service, the service interface includes a `validate` function, under which the service instance must present a certificate signed by a trusted authority. We imagine trusted authorities to be either the user's service provider (*e.g.* T-Mobile, AT&T) or a dedicated security provider (*e.g.* Norton, McAffee). If the security service is unavailable, the WiFi interface (shown in Fig. 7) prompts the user asking whether to remain on 4G secure service from their provider, connect to the insecure network and launch a local anti-virus, or try to find another hotspot which does provide security features.

To test our security service with real malware, we create a sandboxed deployment with just our Android client, the Android IDS, a resolution server, and a sandbox server that spoofs traffic from the Internet to the Android client and deny malware any access to the public Internet. In building our Mobile IDS, we created 170 new rules to detect 23 classes of malware; when running malware software within our testbed, our rules caught 11 out of 11 malware attacks we deployed in our sandbox. It is possible for the Android phone to perform filtering on its own, obviating the need for any in network functionality at all; however, offloading this filtering work to the network saves battery life. In our experiments on the device without malware, four hours of usage drained battery life to 56%, but while running a local antivirus, four hours of usage drained battery life to 49%. Our modifications to the Android WiFi service to check service availability, warn the user and startup antivirus included 124 LOC.

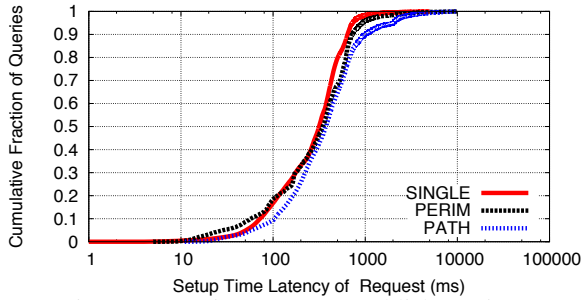


Fig. 8: Setup time (ms) to establish services.

7 Evaluation

We now evaluate our netcalls design for request latency, control plane scalability, service stability, and service availability.

7.1 Latency

Setup latency impacts how service designers can make use of INIT Requests - if the setup latency is high relative to the duration of their intended use, they are unlikely to invoke service processing.

To measure expected latency values, we used a Resolution Server deployment on PlanetLab, having each of 138 PlanetLab nodes serve as surrogate servers for 33,508 ASNs. We assigned each AS a surrogate server with the following algorithm: (1) we assigned the networks that hosted a PlanetLab node the node that they hosted; (2) we assigned networks for which we had router IP addresses [13] the node with the lowest RTT to their address space; (3) we assigned each remaining network the node serving a majority of its peers. Complicating server assignment is the fact that a single resolution server may surrogate for multiple networks; to avoid co-located networks querying each other, we assigned each network a secondary server as well. We then had clients at each site query their local resolution server with INIT requests of each type of placement pattern, such that none of the requests resolved to the local network. We measured the end-to-end latency of each request from the client. Fig. 8 shows the observed latencies for each type of query.

How long does it take to perform an INIT request across the wide area? In the SINGLE case, the median query took 334 milliseconds, and the 90th percentile took 906 milliseconds. These values are closely followed by the PERIM and PATH cases, with median query times of 347 and 374 ms respectively. Overall, the wide area latency (from Resolution Server to Resolution Server) dominates the setup latency - end to end latency for a single request requires slightly more than 2 RTTs.

At what granularity can application developers invoke services without suffering a serious performance penalty? A service setup time of 2-3 round-trip times is negligible overhead for setting up persistent services (*e.g.* firewalls), services for long-lived connections (*e.g.* acceleration for a large file transfer), or frequently used services (*e.g.* invoking a proxy on web browser startup and then proxying all requests to Google). However, for short-lived flows of only a few round trip times, the setup penalty will noticeably impact performance; thus netcalls are not appropriate for this use case.

7.2 Scalability

In this section, we consider the scalability of the netcalls API and whether or not it is feasible for a network to handle INIT requests for a large number of clients. It is impossible to evaluate what we expect to be 'typical' usage patterns before netcalls are deployed; hence we focus here only on upper bounds for the number of INIT requests per second and storage requirements for SRRs. We derive our bounds from a network dataset from a very large enterprise of over 100,000 hosts.

How large are the state requirements for a RISP's resolution server(s)? As an upper bound on state, we consider a model where every host initializes a service for every connection it participates in. As mentioned in the previous section, we expect typical service use to be much less frequent; none of the services we designed depend on per-connection service initialization. In a trace from a week's worth of connections in the large enterprise, at an average point in time there were 108,430 active connections; this would lead to 36.2 MB of SRR state. The peak number of active connections over the course of the week was 240,836; this would lead to 80.4MB of SRR state: a trivial amount of data to store for 100,000 clients.

How many requests per second must a RISP's resolution server(s) be able to handle? Looking at connection load, we once again look for an upper bound, assuming that clients send an INIT request for every TCP session they start (even though INIT requests in practice are likely to be much less frequent). Given an average of 108,430 active connections and an average TCP session length of 60 sec, the average rate of requests would be around 1,800 connections/sec. At peak hours, with an average number of active connections at 270,836, the rate of requests would be about 4,000 connections/sec.

This request load would be easily accommodated by a small number of servers ($<$10), particularly in light of recent results on scaling connection processing[26]. Nevertheless, ISPs can set policies for the number of requests they will accept per client or the granularity at which they will allow INIT requests, hence, ISPs can to some extent control the amount of state and requests they accept.

7.3 Stability

If traffic for a particular service is rerouted away from its service-performing AS due to BGP updates, the client must re-initialize the service. To investigate how often

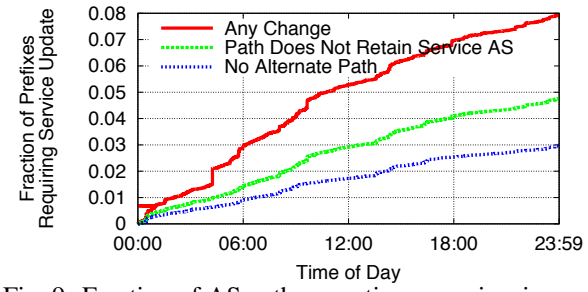


Fig. 9: Fraction of AS paths over time experiencing service failure.

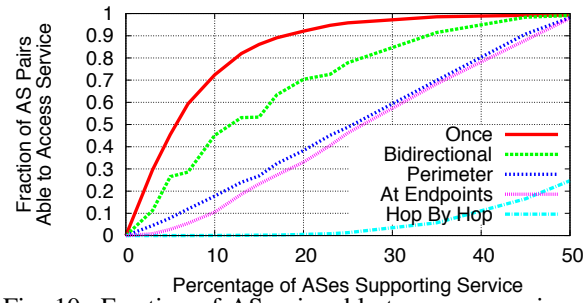


Fig. 10: Fraction of AS pairs able to access service on path between them for various INIT requests.

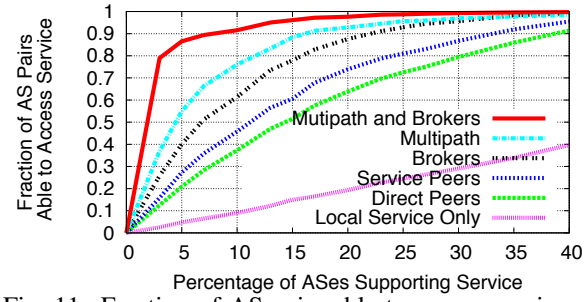


Fig. 11: Fraction of AS pairs able to access service on path between them with and without extensions.

such a scenario occurs, we used routing update logs from the RouteViews [6] project. We created an imaginary AS as a customer of ASes 7018 (AT&T), 3356 (Level 3), and 31500 (a small ISP) and constructed its routing table using default shortest AS path preferences. Then, for each prefix, we selected an AS randomly and labeled it a 'service-performing network'. Starting at Midnight, January 10, 2010, we monitored updates to the imaginary ASes routing table and monitored when updates routed away from service-performing ASes.

How often does a path change remove a service performing path from the route to a prefix? In Figure 9, we show the cumulative fraction of paths which have experienced change over 24 hours. While 8% of paths experience change within 24 hours, only 5% of paths experience change that removes its service-performing AS from the path. Within the first hour, less than half a percent of paths lose their service AS. If the imaginary AS prefers paths that route through the service AS over shortest paths only 3% of paths experience change that loses the service AS; even if one service provider withdraws a path through the service-providing AS, another provider's path may still traverse the same AS.

What fraction of connections will experience service interruption mid-flow? Service interruption only impacts the client if it occurs during a flow on a path in use. Typical connections are short, and to popular prefixes (which studies have shown to have relatively stable paths [29]). To capture the impact of path instability on connections, we combined our BGP trace with a real enterprise trace, assuming that the enterprise was a customer of our imaginary AS and that the traces occurred over the same week long period. Over this period, the hosts in the enterprise contacted over 200 ASes, but we observed only one connection that would have been disrupted by a BGP update. This leads to to believe that from the clients perspective, BGP service interruptions will be negligible.

7.4 Availability

The netcalls architecture expands the reach of deployed services by allowing users to invoke interdomain services. We evaluated this benefit in simulation, using an AS-level routing and topology simulator modeled after those used by other groups [18, 21, 35]. Our simulator modeled the AS graph from January 20, 2010, with 33,508 AS nodes and their peering relationships [4, 6]. For each simulation, we randomly annotated a fraction of the ASes as 'service-adopting' ASes, indicating that the AS supported the requested service; we biased this annotation such that large ASes were twice as likely as the average AS to support the service. We then randomly selected 10k random (Source, Destination) AS pairs and and checked whether the forwarding path(s)[4] between them contained 'service' ASes appropriate to fulfill a service request under a number of constraints.

How often is a service available between a random pair of ASes? In Figure 10, we show the fraction of AS pairs where the networks on the default path between them support the service: 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. About 70% of AS pairs have at least one service-performing AS on a path between them, even when only 10% of networks adopt the service: a 7$\times$ improvement in service availability from extending service invocation across interdomain boundaries. As could be expected, service availability drops for more demanding service types. However, all service types except for 'Hop by Hop' services (at every AS along the path) are available for almost 100% of AS pairs once the service is deployed in only 50% of networks.

When the requesting AS can only invoke services with its service peers, how often is a service available? Be-

[4]If the source AS was multihomed, we checked all its paths.

cause we expect networks to only invoke services with peers, just because a service is deployed doesn't mean that a network will be able to invoke the service (and hence Fig. 10 is an overestimate). In Figure 11, we show how often a service is available once along the forwarding path between source and destination, at an AS which the source AS peers with. When the requesting AS can only invoke services with its physical (Direct) peers, service availability is roughly 35% when 10% of ASes deploy the service. When we assigned each requesting AS 5 additional 'service peers' (selected from Tier-1 ISPs), availability improved such that about 45% of pairs could invoke the service at 10% deployment. Hence, the ability to invoke interdomain services improves service availability even when the requesting network can only invoke services in a restricted number of external networks.

How do the multipath and brokering extensions proposed in §4.3 improve availability? Returning to Figure 11: our proposed extensions dramatically increase availability. With both extensions in use at 10% deployment, almost 90% of AS pairs had access to the service, providing almost universal availability with only very limited service deployment.

8 Related Work

Having sketched netcalls' goals and approach, we briefly contrast our work with related efforts.

Network services. Research has pursued the general vision for in-network services for decades now with several pioneering proposals for specific new services [23, 15, 12]. We focus on the design of a general API rather than a specific service. In this sense, we view our efforts as complementary to this prior work.

Perhaps closer to our goals, is prior work on architectural support for network services more generally [34, 27, 8]. These prior efforts were rooted in the assumption that supporting rich in-network processing required a fundamentally different architecture and hence designed solutions to *replace* the IP service model. In contrast, our design efforts lead us to believe that our goals can be well achieved by *augmenting* the existing service model and see no need to replace IP. A further distinction relative to Active Networks is our more constrained model of network services wherein operators pre-install advanced functionality and expose to users the ability to *invoke* (but not define!) these functions.

Middlebox services today. Some ISPs already expose services to immediate customers [10]; the IETF MidCom group [30] explores standards for communication with *local* middleboxes. Our proposal complements these, targeting users at scale across network domains in a general manner. Traffic processing is also available through overlay or cloud services [1, 3].

Middlebox-centric network architectures. Prior proposals on integrating middleboxes into the broader architecture focused on the naming implications of middleboxes, proposing that users explicitly address the specific middleboxes for their traffic to traverse [9, 33, 31]. These proposals resolve the tension of applying unsolicited functionality to users' traffic. But, they leave unresolved how users discover these middleboxes, how users reason about which middleboxes to select given routing and topology conditions and the role of network providers and their policies in offering and managing middlebox-based services. netcalls tackles the above unresolved questions and, in so doing, proposes a different approach. To reduce complexity for end users and provide network operators with a stake in service selection, we argue 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.

Typed Networking [25], recognizing that hosts may want to avoid certain processing, envisions a 'negotiation' between middleboxes and hosts where boxes on the forwarding path signal the user, that can then opt out of processing. They do not consider an opt-in capability and hence issues of service discovery and availability.

Network evolution. Recent efforts [24] define open APIs between switches and operators within a single domain; they do not discuss the APIs a network exposes externally–to end users and to other networks. Our work likewise complements recent research on programmable routers [16, 17, 22] by showing how the capabilities they enable can be exposed to users.

Service Discovery is a common component of many systems. Most of these however operate in contexts, with goals or technology different from ours; *e.g.* targeting ISP-assisted application-layer service composition [28], wide-area discovery using IP multicast [7], using new naming infrastructures [32], *etc*.

9 Conclusion

We presented netcalls, an API by which user applications invoke advanced processing functions from the network. We presented three end host applications that invoke netcalls to defend against DDoS, compress high-bandwidth connections, and secure against malware.

We do not by any means expect that netcalls 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 Internet - 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.

References

[1] Akamai Security Solutions. http://www.akamai.com/security.

[2] Androguard. http://code.google.com/p/androguard/.

[3] Aryaka WAN Optimization. http://www.aryaka.com.

[4] CAIDA AS Relationships. http://www.caida.org/data/active/as-relationships/.

[5] mod_qos. http://opensource.adnovum.ch/mod_qos/.

[6] RouteViews Project. http://www.routeviews.org.

[7] E. Amir, S. McCanne, and R. Katz. An Active Service Framework and its Application to Real-Time Multimedia Transcoding. In *Proc. ACM SIGCOMM*, 1998.

[8] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse Through Virtualization. *IEEE Computer*, April 2005.

[9] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *Proc. ACM SIGCOMM*, 2004.

[10] T. Benson, A. Akella, and A. Shaikh. Demystifying Configuration Challenges and Trade-offs in Network-Based ISP Services. In *Proc. ACM SIGCOMM*, 2011.

[11] R. Beverly, A. Berger, Y. Hyun, and K. Claffy. Understanding the Efficacy of Deployed Internet Source Address Validation Filtering. In *Proc. ACM SIGCOMM Internet Measurement Conference*, 2009.

[12] R. Braden, L. Zhang, S. Benson, S. Herzog, and S. Jamin. Resource Reservation Protocol (RSVP). RFC 2205.

[13] CAIDA. The Internet Topology Data Kit. http://tinyurl.com/caidaitdk.

[14] B. Carpenter and S. Brim. Middleboxes: Taxonomy and Issues. RFC 3234.

[15] S. Deering, D. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. An Architecture for Wide-Area Multicast Routing. In *Proc. ACM SIGCOMM*, 1994.

[16] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. ACM Symposium on Operating Systems Principles*, 2009.

[17] S. Han, K. Jang, K. Park, and S. Moon. PacketShader: A GPU-Accelerated Software Router. In *Proc. ACM SIGCOMM*, 2010.

[18] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing: The Internet as a Distributed System. In *Proc. USENIX Network Systems Design and Implementation*, 2008.

[19] E. Katz-Bassett, H. Madhyastha, V. K. Adhikari, C. Scott, J. Sherry, P. van Wesep, T. Anderson, and A. Krishnamurthy. Reverse Traceroute. In *Proc. USENIX Network Systems Design and Implementation*, 2010.

[20] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router.

[21] N. Kushman, S. Kandula, D. Katabi, and B. Maggs. R-BGP: Staying Connected in a Connected World. In *Proc. USENIX Network Systems Design and Implementation*, 2007.

[22] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *Proc. USENIX Network Systems Design and Implementation*, 2011.

[23] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM SIGCOMM Computer Communication Review*, 2001.

[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, March 2008.

[25] C. Muthukrishnan, V. Paxson, M. Allman, and A. Akella. Using Strongly Typed Networking to Architect for Tussle. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, 2010.

[26] A. Pesterev, J. Strauss, N. Zeldovich, and R. T. Morris. Improving Network Connection Locality on Multicore Systems. In *Proc. ACM EuroSys*, 2012.

[27] L. Popa, N. Egi, S. Ratnasamy, and I. Stoica. Building Extensible Networks with Rule-Based Forwarding (RBF). In *Proc. USENIX Operating Systems Design and Implementation*, 2010.

[28] B. Raman, S. Agarwal, Y. Chen, M. Caesar, W. Cui, P. Johansson, K. Lai, T. Lavian, S. Machiraju, Z. M. Mao, G. Porter, T. Roscoe, M. Seshadri, J. Shih, K. Sklower, L. Subramanian, T. Suzuki, S. Zhuang, A. D. Joseph, Y. H. Katz, and I. Stoica. The SAHARA Model for Service Composition Across Multiple Providers. In *IEEE Pervasive*, 2002.

[29] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP Routing Stability of Popular Destinations. In *Proc. ACM SIGCOMM Internet Measurement Workshop*,

2002.

[30] M. Stiemerling, J. Quittek, and T. Taylor. Middlebox Communication (MIDCOM) Protocol Semantics. RFC 5189.

[31] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. *Transactions on Networking*, April 2004.

[32] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active Names: Flexible Location and Transport of Wide-Area Resources. In *Proc. USENIX Symposium on Internet Technologies and Systems*, 1999.

[33] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes No Longer Considered Harmful. In *Proc. USENIX Operating Systems Design and Implementation*, 2004.

[34] D. Wetherall, U. Legedza, and J. Guttag. Introducing New Internet Services: Why and How. In *IEEE Network*, May/June 1998.

[35] W. Xu and J. Rexford. MIRO: Multi-Path Interdomain Routing. In *Proc. ACM SIGCOMM*, 2006.