Outsourcing Network Functionality

Glen Gibb
grg@stanford.edu

Hongyi Zeng
hyzeng@stanford.edu
Stanford University
Stanford, CA 94305

Nick McKeown
nickm@stanford.edu

ABSTRACT

This paper presents an architecture for adding functionality to networks via *outsourcing*. In this model, the enterprise network only forwards data; any additional processing is performed by external Feature Providers (FPs). FPs provide and manage features, scaling and moving them in response to customer demand, and providing automated recovery in case of failure. Benefits to the enterprise include reduced cost and management complexity, improved features through FP specialization, and increased choice in services.

Central to the model are a policy component and a Feature API (FAPI). Policy is specified with features not locations, enabling features to be located anywhere. FAPI enables communication between enterprise and FP control planes to share policy and configure features.

We have built a prototype implementation of this architecture called Jingling. Our prototype system incorporates a nation-wide backbone network and FPs located in six sites around the United States.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Distributed Networks, Network Communications*

Keywords

Jingling, outsourcing, network services, middleboxes, SDN

1. INTRODUCTION

Network administrators expect much more from their networks than data transport alone. Today's networks are a blend of "plumbing" to transport packets along paths, and "services" to provide additional in-network features, such as web caching, protection from malicious traffic, network address translation, and load balancing.

Network switches and routers excel at data transport. Focusing on transport alone simplifies hardware, keeping down cost, power and complexity. A number of commercial switching chips are available with a capacity of almost a terabit per second [3, 13], and are used for wiring closets, top-of-rack switches in data centers, and in access and WAN routers. There seems little doubt that cheap and abundant switching capacity (particularly in wireline networks) is here to stay.

1.1 Adding network functionality

While it is clear where to place basic switching, it is much less clear where to place richer functionality. Three approaches are commonly used to add functionality to networks, with varying degrees of success:

Embed in end-hosts : Support for encryption, personal firewalls, tunneling, and anti-spam filters are sometimes provided by the OS or applications. This works well for personalized and standalone services, but is hard to support and keep up-to-date, particularly in large enterprises.

Deploy middleboxes : These act as "bumps-in-the-wire" that process all packets passing through them. Each new feature is typically provided by a new dedicated box. Middleboxes have the benefit of being added only where needed (rather than paying for them in every switch along the path). But we have to decide where to place them: typically, they need to be added at a choke point in the path of all packets [20].

Add to switches and routers : Network equipment vendors frequently add new features to switches and routers. It is common for switches and routers to support thousands of embedded features and protocols (there are over 6,000 RFCs to consider), and to be based on tens of millions of lines of source code. Adding advanced features directly to switches reduces the total number of boxes. Making a subset of boxes feature-rich makes network planning harder: we have to know where to place the feature-rich boxes when laying out the network. Making all boxes feature-rich adds to the overall cost, power consumption, and fragility of the network.

Software Defined Networks (SDN) provide *programmatic* control over traffic flow. The middlebox placement problem can be eliminated with SDN: traffic of interest can be explicitly steered through a middlebox placed anywhere in the network. Policy is specified and stored in a logically centralized control plane and enforced when pushed down to the programmable dataplane devices. This redirection capability was explored in early SDN systems [5, 10] and demonstrated in commercial systems [19] targeting data centers and enterprise networks (commonly termed "service insertion").

1.2 Outsourcing

Common amongst these solutions is their location: they are all situated *within* the network. We pose the question: *Can in-network features be "outsourced"?* That is, can we add functionality to a network by placing the function *outside* the network? For example, how can an enterprise direct all incoming HTTP traffic through an external HTTP virus scanning service? Alternatively, could a network detect malicious traffic using an external intrusion detection system (IDS) service, or spread load across web servers using an external load balancing service?

There are many benefits to outsourcing functionality to a remote *Feature Provider* (FP). First, it allows an enterprise network to be much simpler and focuses resources on building good "plumbing", without having to worry about the purchase, placement and support of middleboxes and feature-laden switches. Second, it allows an external FP to specialize in providing good service to many customers. Third, it encourages improvement of service through competition between FPs.

Enterprises already outsource infrastructure such as web hosting, e-mail, and databases for the same reasons. Outsourcing network services could remove 10s to 1000s of devices from typical networks [17], reducing capital and operational expenditure.

On the face of it, this may sound like a marketing plan to save Internet service providers from obsolescence by giving them a means to take control of our enterprise networks. Our goal is in fact quite different: we wish to allow anyone to provide a service, without regard to location. We wish to allow features to be placed anywhere, and to be called up from anywhere else. New functionality might be placed in a traditional service provider, or provided by a dedicated third party company. A global enterprise might place complex services at its headquarters, or inside a public cloud (e.g. Amazon EC2 [1]). An enterprise should be able to deploy a network service, without having to worry about where it is placed.

We are not the first to propose that network services be inserted arbitrarily. Service Insertion Architecture [16] describes a way for services to be inserted along the path. In principle, policy-based routing can be used to route traffic to anywhere in the network [6]. And several researchers have recommended waypoint services where traffic is brought to the service boxes [5, 10, 15].

All of these proposals have the advantage of crisply distinguishing a clean, simple data transport network ("dumb and minimal", as the Internet pioneers envisaged), from the more complex, dedicated devices providing added functionality. But existing methods assume that the "plumbing" and the "services" are under the control of the same organization. The network administrator is assumed to know precisely where all middleboxes are placed, and where traffic should be routed, and they must carefully manage the routing tables so that traffic is carried to the right box. Whenever the topology changes—and it frequently does because a link or switch fails, new VLANs are added, or new switches are added—then switches and routers must be manually reconfigured to re-route traffic. This approach is time-consuming, and fraught with risks.

Instead, we believe the network administrator should be able to express a policy such as: *All of my incoming SMTP traffic should first pass through Company A's spam filter service*, regardless of the state of the data transport network, and regardless of the location of the spam filter. The enterprise network administrator should be able to choose another service provider by merely replacing their name in the policy. Mapping of policy onto the underlying network should be performed by an intelligent control plane.

1.3 Outsourcing requirements

To successfully outsource network functionality, we need the following to hold true:

Policy should be expressed via service names, not locations. Network administrators may not have control over where a particular service resides outside their network. Policy absent of location information does not need updating when a service is moved, for example to bring it closer to the user. It also allows the service to scale, by allowing the service provider to choose from among multiple copies of a service.

Service should be established via a well-defined open API. The network administrator must be able to clearly identify the service being requested, and (optionally) the service provider. The network administrator must also be able to clearly identify the set of traffic requiring special processing. This in turn must be used to automatically configure the network. If redirection is local (i.e. if the traffic is to be redirected only once it reaches the administrator's network), the rules are applied locally. If redirection is to be applied remotely—for example, by an Internet service provider—then the local administrator needs a way to characterize and communicate how traffic is to be redirected on his/her behalf.

Functionality should be abstracted as a service. Network operators should focus on *what* functions they want and not *how* those functions are implemented. A service details a contract to process data in a given manner—multiple implementations that adhere to the same contract provide the same service.

As an example, we can use these three properties to implement the virus scanning example cited earlier. The service is invoked using the policy statement: "*All HTTP traffic* $\rightarrow$ *virus scan*", without reference to the location, or the mechanism used to perform the scanning. The network administrator and the virus scanning provider use an open API to negotiate the use and configuration of the service.

We built a prototype system that we call Jingling[1]. Jingling allows arbitrary functionality to be interposed in a packet flow, regardless of location; Jingling is designed so that remote service providers can offer arbitrary network services to enterprises.

Our Jingling prototype includes:

- A policy language and Feature API for service negotiation.

[1]Jingling (精灵) means sprite in Chinese.

- An enterprise network controller to map policies into network settings.
- A resource manager that allocates and manages resources.

The resulting system requires no end-host changes and is based on a simple, minimal forwarding plane. Services may be located anywhere, implemented in hardware or software, and may be fixed-function or programmable. Although the system was designed to enable outsourcing of functionality, the same system can be used to support deployment of services within the enterprise.

Our system is inspired by several branches of existing work. Mechanisms for separating network policy from the network infrastructure and using centralized control are found in many SDN proposals [4, 5, 8, 14]. Approaches to use indirection to solve middlebox problems are found in [9–11, 18, 20] and commercial products [7, 16].

An alternate outsourcing system [17] was proposed independently and in parallel to this work. There the authors motivate the case for outsourcing with a survey of middlebox usage in networks. Our work was motivated by the programmatic control afforded by SDN and focuses very much on the need for an definition of the Feature API (FAPI).

2. JINGLING ARCHITECTURE

2.1 Overview

At its core, Jingling is an architecture for adding functionality or features to a network. Jingling takes the unorthodox approach of placing features *outside* the traditional network boundary.

A high-level overview of the Jingling model is shown in Figure 1. The model involves three stakeholders: the enterprise, the Internet Service Provider (ISP), and the *Feature Provider* (FP). The enterprise requires one or more features which the FP provides. The enterprise and FP are unlikely to be located adjacent one another and thus must communicate via an ISP. One ISP and one FP are shown for simplicity but multiple of each may be involved.

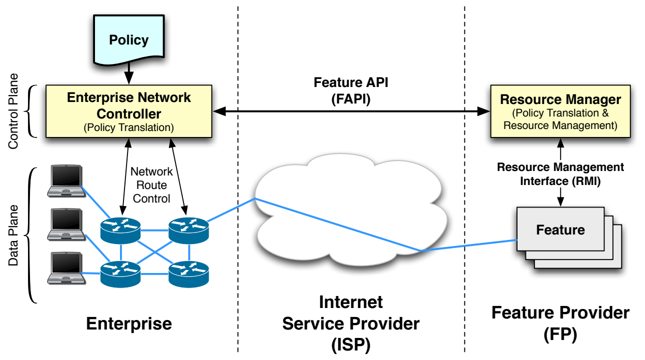

Figure 1: Overview of the Jingling architecture. An enterprise utilizes one or more features provided by a Feature Provider (FP). The enterprise and FP communicate via a Feature API (FAPI) to configure features and specify which feature(s) to apply to different traffic.

The enterprise network administrator specifies policy that determines the features to apply to particular traffic. The policy is installed into the enterprise control plane or *Enterprise Network Controller* (ENC), shown in Figure 2. The ENC has two key elements. A *translation element* maps policy into network configuration, utilizing information about the configured features and current topology. A *topology monitoring element* updates network configuration when the topology changes due to link or switch addition, removal, or failure. The ENC also maintains databases of topology, feature, and policy information.

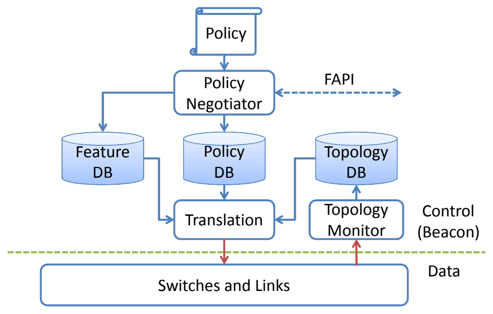

Figure 2: Enterprise Network Controller (ENC): translates policy and monitors topology within the enterprise.

The *Resource Manager* (RM) is the FP counterpart to the ENC, and is shown in Figure 3. The RM configures the FP's resources to create processing pipelines that implement features. The RM's structure is similar to the ENC. Policy from the ENC is stored in the RM's policy database and translated into network configuration. Currently configured feature information is store in a feature database. A *management element* manages the creation, deletion, and movement of feature instances. Network configuration must be updated whenever feature instances or the network state change.

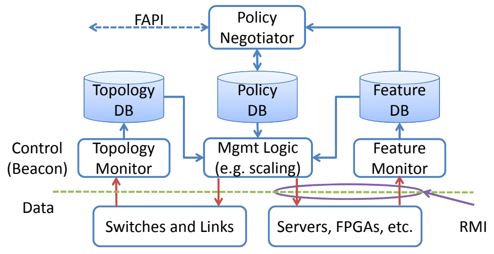

Figure 3: Resource Manager (RM): manages/monitors services and translates policy within the Feature Provider.

Communication between the ENC and RM is performed via a *Feature API* (FAPI). FAPI is used to communicate policy information and to configure each feature.

Required in this model is the ability to route traffic to and from each feature. Several feasible choices (e.g. Policy-Based Routing, Tunneling, OpenFlow [14]) of routing mechanism exist. Regardless of the mechanism that is used, an intelligent control plane translates the high-level feature-based policy into the low-level mechanism to route traffic appropriately.

2.2 Policy

Policy determines the features to apply to traffic. Policy must be specified in terms of *which* features to use but not *where* those features are located. Support for scaling and mobility within Jingling rely on the absence of location information; specifying only the feature allows the control plane to map and remap traffic to elements providing a feature as it deems necessary.

Policy consists of a set of policy rules conforming to the following basic structure:

$$\textit{traffic pattern} \rightarrow \textit{features}$$

where *traffic pattern* specifies the subset of traffic that the rule applies to, and *features* is an ordered list of features to apply to that traffic. For example, a policy rule that directs all HTTP traffic within an IP subnet through a caching service and a firewall service is expressed as:
`[ip_dst=192.168.0.0/24, tp_dst=80 → caching, firewall]`.

2.3 Feature API (FAPI)

The Feature API allows the enterprise to communicate with the FP to 1) specify policy to apply to traffic, and 2) configure features. FAPI also allows communication with Jingling-aware ISPs.

FAPI provides a uniform interface through which entities communicate. Implementation details of each entity are hidden behind this interface, allowing an enterprise to communicate with any FP. Table 1 presents the core FAPI calls.

Name	Description
installPolicy(p)	Install policy rules p
removePolicy(p)	Remove policy rules p
writeState(f, k, v)	Write state (k, v) into feature f
readState(f, k)	Read state k from feature f

Table 1: Feature API for control communication between enterprise and FP.

Policy specification is performed via *installPolicy(p)* and *removePolicy(p)*. These two calls are invoked on the FP control plane by the enterprise control plane in response to policy changes by the network administrator. Some features may allow configuration by the enterprise to modify the feature behavior. The set of configurable parameters is feature specific; an HTTP content filtering feature may allow the selection of the content type to filter, while an HTTP cache may not allow any configuration at all. Feature configuration is performed via *writeState(f, k, v)* and *readState(f, k)*. These calls present configuration as a set of key-value pairs (k, v) that are read and written for each feature.

2.4 Resource Management Interface (RMI)

The Resource Management Interface allows an RM to manage resources under its control. RMI presents a uniform interface across resources to the control plane. Table 2 shows the core RMI calls.

RMI uses the concepts *feature*, *resource*, and *instance*. A feature details a contract to process data in a given manner; a resource is a physical entity capable of providing one or more features; an instance is an implementation of a feature on a resource. Resources can be commodity PCs, programmable hardware, or fixed-function middleboxes; the middlebox permanently provides one fixed instance of a feature.

Resources must implement appropriate logic to process each RMI call. A *createInstance* call to a software node should start a new software process; the same call to a programmable hardware node should download the appropriate bitstream.

2.5 Traffic redirection

A mechanism is required to redirect traffic through FPs. Traffic originating within the enterprise is redirected by appropriate routing rules within the enterprise network. The FP may rewrite source addresses to ensure that return data is delivered directly to the FP. Traffic originating outside the enterprise is redirected at the enterprise edge (simple, but adds routing delay) or within the internet. Advertising addresses allocated by the FP instead of those of the enterprise[2] effectively redirects traffic as it enters the internet.

Traffic must transit one or more ISPs between the enterprise and FP. Such traffic can be transported via encapsulation or by establishing paths through the ISP using FAPI (Jingling-aware ISPs only).

3. IMPLEMENTATION AND RESULTS

To evaluate the merits of outsourcing network functionality, we built the nationwide testbed shown in Figure 4. The "enterprise" is a small set of servers located at Stanford University. Three FPs were created across the US. Two of the FPs are single-site, one each at Stanford and Utah (Emulab). The third FP models a multi-site provider, with locations in Los Angeles, Houston, New York, and Washington D.C.; the servers are physically located in Internet2 PoPs. Two features were deployed: a web cache and an intrusion prevention system.

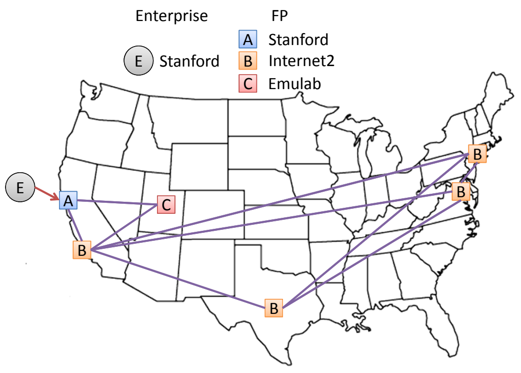

Figure 4: The Jingling Prototype

3.1 Implementation

Our enterprise consists of several unmodified PCs within our local network at Stanford. Each FP is composed of a small number of x86-based servers, most of which contain a NetFPGA. Features are implemented via software (either virtual machines or processes on non-virtualized hosts) or firmware on the NetFPGAs. These components are stitched

[2]Enterprises who have moved their IT infrastructure, such as web and mail servers, to the cloud already advertise IP addresses owned by their service providers.

Name	Description
Control Plane $\rightarrow$ Resource	
createInstance(f)	Create a new instance of feature f
destroyInstance(i)	Destroy feature instance i
moveInstance(i, l)	Move instance i to location l
writeState(i, k, v)	Write state (k, v) to instance i
readState(i, k)	Read state k from instance i
Resource $\rightarrow$ Control Plane	
instanceJoin(i)	Instance i is available
instanceLeave(i)	Instance i is leaving
instanceMoved(i, l_{old}, l_{new})	Instance i has moved from l_{old} to l_{new}
instanceEvent(i, e)	Event e occurred at i

Table 2: Resource Management Interface for FP resource control

together to form a complete pipeline. OpenFlow networks within each FP interconnect servers, and all are Internet connected.

The prototype ENC and RM are implemented as extensions to Beacon [2], a Java-based, multi-threaded, modular OpenFlow controller. The Jingling bundle contains approximately 2,500 lines of Java. A small RMI client daemon, written in Python, runs on each server. The client can start and stop processes/VMs, and read the state of features. FAPI and RMI method calls were encoded using JSON.

Traffic between the enterprise and FPs is forwarded over GRE tunnels. Outbound connections from the enterprise are rewritten by the FP to enable return traffic to be routed directly to the FP. Inbound connections from the internet must be rerouted to the FP by the enterprise.

3.2 Results

3.2.1 *Baseline Functionality*

A simple test verified Jingling's baseline functionality of mapping policy onto the underling network. The test utilized the enterprise network, the Emulab FP, and the Internet2 FP. Two features were used: a web cache hosted in Internet2, and an Intrusion Prevention System (IPS) hosted in Emulab.

The test operated as follows. A client within the enterprise repeatedly requested a fixed page from a distant public HTTP server, while the policy was modified according to Table 3. Initially no policy was installed, allowing direct access to the HTTP server. The web cache was inserted, reducing access time for the page. An IPS that blocks HTTP traffic was inserted *after* the cache, but the client is served the page from the cache. Finally, the IPS and cache order were reversed to force traffic through the IPS before the cache, thereby preventing the client retrieving the page.

Step	FAPI Calls	Latency
0	—	High
1	*installPolicy*(`tp_dst=http`$\rightarrow$`[cache]`)	Low
2	*installPolicy*(`tp_dst=http`$\rightarrow$`[cache,IPS]`) *writeState*(`IPS, rules, 'http deny'`)	Low
3	*installPolicy*(`tp_dst=http`$\rightarrow$`[IPS,cache]`) *writeState*(`IPS, rules, 'http deny'`)	∞ (Blocked)

Table 3: Policy changes to demonstrate baseline functionality

3.2.2 *Resource Management Tasks*

We implemented three advanced resource management tasks that use the RMI.

Scaling: Scaling was tested by generating load on a caching service within the Stanford FP. Instances were created/destroyed by the RM in response to traffic load, with each instance hosted as a process on a separate machine. Figure 5 shows the number of instances closely tracking aggregate load.

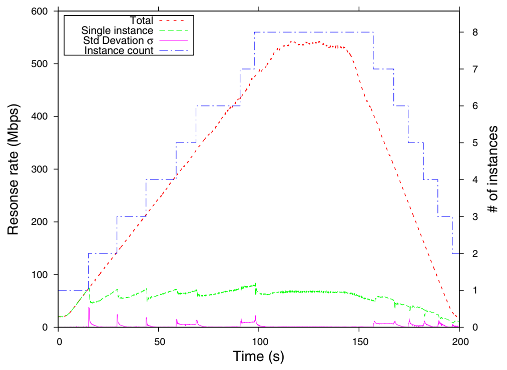

Figure 5: Automatic feature scaling in response to load. The standard deviation (σ) across instances reflects fairness. The trigger rate at which new instances are created can be read off the instance count line using the left-hand axis.

Fast Failure Recovery: In this test, the RM detected and recovered from failures. It monitored the health of each resource via the RMI interface with periodic heartbeats. Failure recovery was performed by switching to a hot-standby or creating a copy of instances hosted on the failed resource. In a setup similar to the scaling experiment, the RM switched to a hot-standby instance in 190ms.

Live Migration: An FP may migrate a service closer to users to reduce response times. In this test, a client generated HTTP requests to a fixed URL, and the requests were redirected through a caching service within Internet2. The cache started in New York, but the RM detected the client's location and moved the cache to Los Angeles to be closer to the client. Response time was reduced by approximately 140ms. The underlying VM migration mechanism resulted in approximately 300ms of downtime during migration.

4. CONCLUSION AND FUTURE WORK

The Jingling architecture enables network functionality to be *outsourced* to external Feature Providers (FPs). Policy is expressed via service names, not locations, and an intelligent control plane maps policy onto the underlying network. A Feature API (FAPI) enables communication of policy between the enterprise and FP. A prototype was built that demonstrates the core Jingling functionality.

There are several areas that we believe deserve more focus before a production deployment is built:

Performance: Jingling incurs latency penalties due to increased path lengths (as with other indirection-based systems [10, 18]). The penalty should be small if the feature provider is located near the original traffic path, and may even be negative [12]. Content delivery networks provide a model for deployment; they have a large number of geographically-distributed sites to minimize distance from users. Measurement should be performed to estimate the impact of different deployment scenarios.

Traffic redirection: Several redirection mechanisms are proposed in § 2.5. We implemented one set of mechanisms for use in our Jingling prototype; the cost and benefit of each mechanism should be quantified to enable selection of the best mechanism.

Security and trust: There are two primary concerns for an enterprise using an FP: is the FP providing the agreed service, and is the FP (and only the FP) using the data appropriately? Verifying that a service is provided could be achieved by monitoring and probing the FP, either by the enterprise or a third-party. Ensuring appropriate use of data is a harder problem. Eavesdropping by third-parties can be prevented using existing mechanisms (eg. encrypted tunnels); trust of an FP could be based upon reputation or a mechanism such as external auditing. Users of cloud compute services face similar concerns but this has not prevented their widespread use.

5. REFERENCES

[1] Amazon EC2. `https://aws.amazon.com/ec2/`.

[2] Beacon. `http://www.beaconcontroller.net/`.

[3] Broadcom BCM88600-Series. `http://www.broadcom.com/products/Switching/Carrier-and-Service-Provider/BCM88600-Series`.

[4] Caesar, M., Caldwell, D., Feamster, N., Rexford, J., Shaikh, A., and van der Merwe, J. Design and implementation of a routing control platform. In *Proc. of USENIX NSDI '05* (Berkeley, CA, USA, 2005), NSDI'05, USENIX Association, pp. 15–28.

[5] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: taking control of the enterprise. *SIGCOMM Comput. Commun. Rev. 37* (August 2007), 1–12.

[6] Cisco: Manipulating Routing Updates. `http://ptgmedia.pearsoncmg.com/imprint_downloads/cisco/bookreg/2237xxd.pdf`.

[7] Cisco Unified Network Services: Overcome Obstacles to Cloud-Ready Deployments. `http://www.cisco.com/en/US/solutions/collateral/ns340/ns517/ns224/ns836/ns976/white_paper_C11-64521.html`.

[8] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev. 35* (October 2005), 41–54.

[9] Greenhalgh, A., Huici, F., Hoerdt, M., Papadimitriou, P., Handley, M., and Mathy, L. Flow processing and the rise of commodity network hardware. *SIGCOMM Comput. Commun. Rev. 39* (March 2009), 20–26.

[10] Joseph, D. A., Tavakoli, A., and Stoica, I. A policy-aware switching layer for data centers. *SIGCOMM Comput. Commun. Rev. 38*, 4 (Aug. 2008), 51–62.

[11] Lee, J., Tourrilhes, J., Sharma, P., and Banerjee, S. No more middlebox: integrate processing into network. *SIGCOMM Comput. Commun. Rev. 40* (August 2010), 459–460.

[12] Lumezanu, C., Baden, R., Spring, N., and Bhattacharjee, B. Triangle inequality variations in the internet. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference* (New York, NY, USA, 2009), IMC '09, ACM, pp. 177–183.

[13] Marvell Prestera CX8297. `http://investor.marvell.com/phoenix.zhtml?c=120802&p=irol-newsArticle&ID=1561826`.

[14] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev. 38* (March 2008), 69–74.

[15] Ng, T. S. E., Stoica, I., and Zhang, H. A waypoint service approach to connect heterogeneous internet address spaces. In *Proc. USENIX Technical Conference* (Berkeley, CA, USA, 2001), USENIX Association, pp. 319–332.

[16] Quinn, P., Durazoo, K., Harvey, A. G., Gannu, S., Cheng, D., Baker, F., Pruss, R. M., Greene, B. R., Rajendran, S., and Gleichauf, R. Service insertion architecture. Patent Application, 07 2008. US 2008/0177896 A1.

[17] Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., and Sekar, V. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proc. of SIGCOMM* (2012).

[18] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. *SIGCOMM Comput. Comm. Rev. 32*, 4 (Aug. 2002), 73–86.

[19] 2011 Open Networking Summit Demonstrations. `http://opennetsummit.org/demonstrations.html`.

[20] Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., and Shenker, S. Middleboxes no longer considered harmful. In *Proc. of OSDI '04* (Berkeley, CA, USA, 2004), USENIX Association, pp. 215–230.