StEERING: A Software-Defined Networking for Inline Service Chaining

Ying Zhang, Neda Beheshti, Ludovic Beliveau, Geoffrey Lefebvre†, Ravi Manghirmalani, Ramesh Mishra,
Ritun Patney†, Meral Shirazipour, Ramesh Subrahmaniam, Catherine Truchan, Mallik Tatipamula†
Ericsson Research

***Abstract*—Network operators are faced with the challenge of deploying and managing middleboxes (also called inline services) such as firewalls within their broadband access, datacenter or enterprise networks. Due to the lack of available protocols to route traffic through middleboxes, operators still rely on error-prone and complex low-level configurations to coerce traffic through the desired set of middleboxes. Built upon the recent software-defined networking (SDN) architecture and OpenFlow protocol, this paper proposes StEERING, short for SDN inlinE sERvices and forwardING. It is a scalable framework for dynamically routing traffic through any sequence of middleboxes. With simple centralized configuration, StEERING can explicitly steer different types of flows through the desired set of middleboxes, scaling at the level of per-subscriber and per-application policies. With its capability to support flexible routing, we further propose an algorithm to select the best locations for placing services, such that the performance is optimized. Overall, StEERING allows network operators to monetize their middlebox deployment in new ways by allowing subscribers flexibly to select available network services.**

I. Introduction

Network appliances such as firewalls, content filters, intrusion detection systems (IDS), deep packet inspection (DPI), network address translation (NAT), content caches, load-balancers, wide area network (WAN) accelerators, multimedia transcoders, logging/metering/charging/advanced charging applications, etc. are generally referred to as middleboxes or inline services because end users are often unaware of their existence in their traffic's path. Middleboxes are inevitably deployed in broadband access networks (fixed or mobile), in enterprise networks, and more recently in data centers and cloud environments. While this topic area has received a significant amount of attention in recent years [1], [2], [3], [4], [5], [6], [7], there is still no satisfactory solution for the deployment and management of middleboxes in broadband access networks. In fact, given the forecasts of traffic growth and reduction of average revenue per user (ARPU), network operators need to address the problem of reducing the costs associated with the deployment of middleboxes as well as new ways to monetize some of these network based services. For example, it is desirable that upon detection of a long lived video flow, DPI could be dynamically bypassed to save processing resources. Another example is that a transcoding appliance can be bypassed once the content type and codec have been identified. Scalable and dynamic steering capabilities could also enable new ways to monetize the middlebox deployment, where subscribers can buy network based services.

However, there are currently challenges associated with configuring the network to bypass certain middleboxes for given traffic flows. Because of the lack of efficient protocols for this problem, operators often need to use low level and complex configuration to achieve certain degree of control. Thus, today's practice is either to direct excess traffic through unnecessary middleboxes, or to set up an unwieldy set of tunnels.

This paper describes StEERING (SDN inlinE sERvices and forwardING), a novel framework based on the concepts of Software-Defined Networking (SDN) that addresses these challenges by empowering network operators with the ability to *steer* traffic at the granularity of subscriber and traffic types, using simple policies which are defined and propagated from a centralized point of control. Moreover, the ability to offer a per-subscriber selection of inline services leads to the creation of new offerings and hence new ways for operators to monetize their networks. StEERING does that by reusing existing OpenFlow 1.1 [8] specification, in a novel fashion without the need for any new extensions. The design of StEERING aims to meet the following requirements:

- ***Efficiency:*** traffic should traverse middleboxes in the sequence specified by the network operators and should not unnecessarily traverse middleboxes. Great CAPEX savings could be achieved if traffic could be selectively steered away (bypassed) from specific inline services.
- ***Flexibility:*** the framework should support subscriber, application, and operator specific policies simultaneously, all stemming from a single control point. Adding or removing new services should be easily done by the network operator.
- ***Scalability:*** the framework should support a large number of rules and scale as the number of subscribers/applications grows.
- ***Openness:*** it should be possible to deploy any type of middlebox in the network, independently of its vendor in order to avoid vendor lock-in backward compatibility with already deployed middleboxes. Our method can be applied to both physical and virtualized middle boxes.

Concretely, StEERING consists of a set of OpenFlow 1.1 switches interposed between middleboxes and their interconnections (Ethernet switches and IP routers). The OpenFlow switches are programmed and managed by a centralized controller which provides a single interface for the deployment of fine-grained traffic steering policies. StEERING provides scalability in three distinct ways. First, in order to reduce the amount of state required at each switch, we use multiple tables to transform the flat policy space into a multi-dimensional space. Therefore, the rules required can scale linearly with the number of subscribers and applications, instead of the cross-product of subscribers and applications. Second, we define the

†These authors are no longer at Ericsson Research.

978-1-4799-1270-4/13/$31.00 ©2013 IEEE

set of services to traverse for each flow as one type of metadata, which is used to communicate information between tables. Thus, every table can operate on the service set independently, *e.g.,* adding/removing services. This design enables the simple integration of different types of policies. Third, StEERING pushes to the perimeter of the service delivery network expensive forwarding operations such as classification and header rewriting. These operations need to be done only once within the network. We also perform intelligent lookup to improve the switch's performance.

We make the following three key contributions in this paper:

- We propose a scalable and flexible architecture to reduce the state required at each switch, by leveraging new features available in OpenFlow 1.1 (the proposal is applicable to later versions of the protocol). This design enables the simple and efficient integration of different types of traffic steering policies.
- Given the flexible steering capability supported by the above design, we further address the issue of where to connect the inline services to the network. To this end, our main contribution is to formulate the problem and analyze its importance. We further use a heuristic solution and demonstrate that it can significantly improve user performance. The efficient solution can also be be periodically run to determine the locations for virtualized appliances.
- We demonstrate the deployment feasibility of our approach by implementing the StEERING forwarding plane on a network processor which has already proven its efficiency in commercial routers [9]. We also evaluate the scalability using real broadband network traffic.

The novelty of our design is reusing existing SDN concepts, *e.g.,* OpenFlow 1.1 multiple tables and metadata,to solve the inline service chaining problem in a scalable fashion. The rest of this paper is organized as follows. We introduce the problems and summarize limitations of existing approaches in §II. §III presents an overview of the StEERING architecture and §IV describes each component in details. §V describes our algorithm for determining the service locations. In §VI we describe our prototype and evaluation results. Finally, we summarize the related work in §VII and conclude in §VIII.

II. Problem Statement

The StEERING design presented in this paper is a general solution that can be applied to any network with middlebox services. Broadband (fixed and mobile) networks will however specifically benefit from the presented design, because of its scalability – i.e., its ability to handle per-subscriber and per-application traffic for a large number of subscribers and applications. For this reason, we focus the use-cases and examples on broadband networks.

The network-wide view of StEERING is shown in Figure 1. The broadband access network depicted here consists of three areas: the access network (with either mobile or fixed access technologies), the aggregation network, and the core network (hosting packet gateways and policy control servers). A set of middleboxes resides in the core network, between

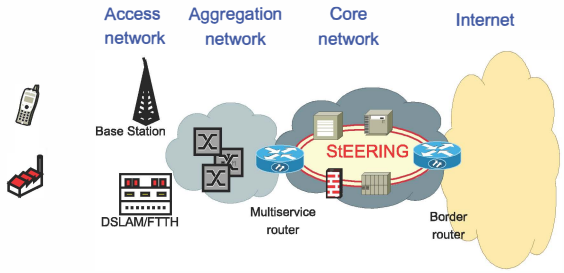

Fig. 1. Network view of StEERING

	Steering	Efficiency	Flexibility	Openness	Scalability
Single box	✓	✓	✗	✗	✗
Static chains	✓	✗	✗	✓	✗
PBR	✓	✗	✗	✓	✗
Pswitch	✓	✓	✗	✓	✗
StEERING	✓	✓	✓	✓	✓

TABLE I. Summary of existing approaches

the ingress routers and the border routers that connect to the Internet. These service-providing middleboxes, together with the switches that interconnect them, are referred to as the *service delivery network*. We define *upstream* to be the direction *from a subscriber to the core network*. Conversely, *downstream* traffic is *from the core to a subscriber*. Note that we allow the existence of multiple ingress and egress points, but the traffic only goes in two directions, *i.e.,* from subscribers to the Internet and vice versa. We do not consider communications between subscribers in this work.

Service chaining is required if the traffic needs to go through more than one inline services. Moreover, if more than one chain of services is possible, then the operator needs to configure the networking infrastructure to direct the right traffic through the right inline service path. In this paper, *traffic steering* refers to leading the traffic through the right inline service path. Unfortunately, in today's networks, this process is inflexible and prone to mis-configurations.

Limitations of existing solutions: The operators today still do not have protocols or tools available to perform flexible traffic steering. We summarize the existing approaches used in operational networks below. Their pros and cons are discussed in Table I, according to the requirements listed in §I.

- Single box running multiple services: This approach consolidates all inline services into a single box [10]. The operator adds new services by adding additional service cards to its router or gateway. This approach makes the integration of third party service appliances difficult. It also suffers from a scalability issue as the number of services and the aggregated bandwidth is limited by the router's capacity.
- Statically configured service chains: It configures one or more static service chains where each service is configured to send traffic to the next service in its chain. This approach is not flexible as it cannot dynamically reconfigure the set of services easily. It requires a large amount of service-specific configuration and is error prone. It certainly cannot support the steering at the granularity of per subscriber.

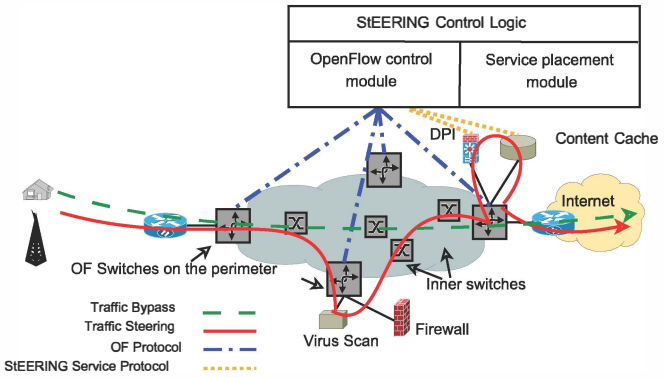

Fig. 2. System architecture

- Policy-based routing: A third approach is to use a router using policy-based routing (PBR) to route traffic to different services. Each time the traffic is returned to the centralized router and then sent to the next service. Clearly, this approach suffers from scalability issues as traffic is forced through the router after every service. The router must be able to handle N times the incoming traffic line rate to support a chain with $N-1$ services.
- Policy-aware switching layer: Joseph *et al.* [1] propose a policy-aware switching layer for data centers which explicitly forwards traffic through different sequences of middleboxes. Their method satisfies the efficiency requirement but fails to meet the requirements of flexibility and scalability. Each policy needs to be translated into a set of low level forwarding rules on all the relevant switches. There is no explicit way to configure application and subscriber related rules separately which need to be manually consolidated into a set of low level rules. Moreover, it requires to install one rule for each new flow. Therefore, it is hard to scale with the number of subscriber/application combinations.

Location of the services: Even if we have the perfect steering mechanism to route traffic through different middleboxes as the subscribers' policies define, there is still an orthogonal problem at the network planning stage, *i.e.,* where do we connect these services so that the users' performance is the best. Apparently, traffic steering will result in prolonged delay in user performance, due to the detour to the services. Thus, how these services are connected to the network clearly has an impact on the user performance.

III. System Overview

StEERING is built upon the software-defined networking (SDN) concept. SDN is a new network architecture that introduces programmability, centralized intelligence and abstractions from the underlying network infrastructure [11]. In SDN, the control plane applications that implement network functionalities (e.g., routing and switching) can evolve separately from the forwarding plane. OpenFlow (OF) [8] is an open standard protocol proposed for the interface between the control and forwarding planes in SDN.

The components of StEERING are shown in Figure 2. There are two types of switches in the network. The *Perimeter OF Switches* are placed on the perimeter of the service delivery network. These switches will classify the incoming traffic and steer it towards the next service in the chain. These are the switches to which services or gateway nodes are connected. The *Inner Switches* will forward the traffic using efficient L2 switching. These switches are only connected to other switches and may or *may not* be OF switches.

There are two modules of StEERING control logic. The first module is a standard OpenFlow controller, which is responsible for setting up the table entries in the OF switches, via OpenFlow protocol (blue dashed lines). The second module of the control logic is an algorithm that periodically runs to determine the best locations for inline services. The openflow rules are used to perform two-step process for steering. The first step classifies incoming packets and assigns them a service path based on predefined subscriber, application, and ordering policies. The second step forwards packets to a next service based on its current position along the assigned service path. For example, the solid red line in Figure 2 shows a service path for the upstream traffic through Virus Scan, DPI and Content Cache. The green line, on the other hand, allows the traffic to bypass all the services.

In most cases, policies can be determined by fields in the packet headers. Alternatively, it can also be determined by the packet payload, such as URLs. In such cases, the policy can be decided on the fly via a DPI box, which will inform the controller with the packet content. Thus, in our architecture we also propose an interface between the inline services and the OF controller, as the yellow dotted lines in Figure 2. For instance, once DPI has recognized or resolved a flow, it can send a notification message to the OF controller.

The service policy for each flow is determined by the operator but can also be specified by the subscriber, *i.e.,* customers of the ISPs, which can be either end users or enterprise customers. Therefore a subset of the services traversed by the traffic can be selected by the subscriber, perhaps on a pay per service basis.

IV. Basic Design Components

The data plane (forwarding) of StEERING can be easily configured and scaled as the number of subscriber/application combination grows. The controller programs switches with the rules on how to forward each packet. Forwarding decisions are made based on Layer 2 to Layer 4 contents of packets as well as the ingress port. The key challenge to achieve scalability is to avoid exponential growth (rule explosion) of the forwarding rules installed in each switch. We make three design choices to reduce the amount of state on each switch: defining port types to indicate directions, using multiple tables to decompose multi-dimensional policies, and introducing a new metadata type to encode service paths.

A. Represent directions with port types

We define two types of ports on perimeter switches: *node* ports and *transit* ports. Node ports are connected to services and gateway nodes (BNG, GGSN, routers). Transit ports are connected to other perimeter switches or to inner network switches.

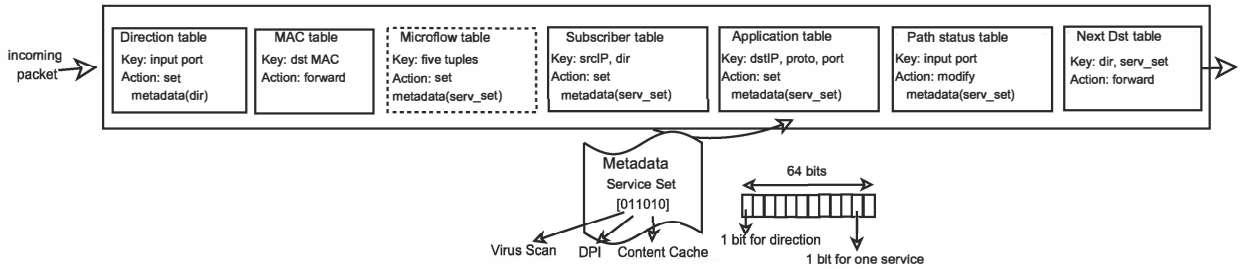

Fig. 4. Multiple tables and metadata

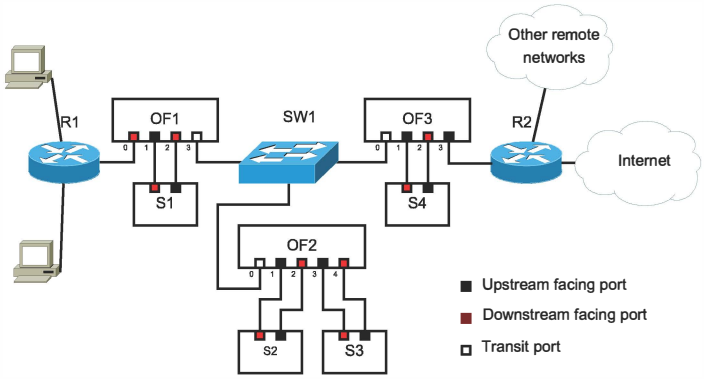

Fig. 3. Illustration of port direction

Figure 3 shows an example of a Service Delivery Network based on the StEERING architecture. Switches OF1, OF2, and OF3 are perimeter switches. Switch SW1 is an inner switch. The black/red ports on the switches are node ports and the white ports are transit ports.

Incoming traffic, either coming in from a gateway node or coming back from a service, always enters the Service Delivery Network via a perimeter switch and through a node port. Packets coming in through node ports are steered towards the next node (service or gateway) in their assigned service paths. Packets arriving on transit ports are simply forwarded using their destination MAC addresses.

All packets traversing the steering network are considered to be traveling either upstream or downstream. Each node port in the perimeter switches is either facing upstream or downstream. In Figure 3, downstream-facing ports are colored red and upstream-facing ports are colored black. All packets that arrive on a downstream-facing port are traveling upstream, and vice versa. Packets arriving on transit ports may be traveling in either direction. In this case, the direction is known based on the destination MAC address, which will correspond to either an upstream-facing or downstream-facing service or router port.

B. Realize policies with multiple forwarding tables

In theory, a single TCAM-like table could be used to specify the required functionality, as in OpenFlow 1.0 or pswitch. However, this would not be a scalable solution because it would involve the cross-product of subscribers, applications, and ports within the same table. *Using indirection and multiple tables, we separate this into multiple steps, resulting in linear scaling of each table.*

Six tables are mandatory. The matching keys and the associated actions of these tables are shown in Figure 4.

Direction table: Ingress port is the key, which identifies the direction of the packet (upstream vs. downstream) and the type of the port the packet was received on (node vs. transit port). The action is to set the a metadata field called *dir*. We will explain more about the metadata fields in Section IV-D.

MAC table: Destination MAC address is the key. Based on the contents of this table, the packet will either be transmitted to a directly connected service or router on a node port, forwarded out to another transit port, or dropped.

Subscriber table: Subscriber's identifier (*e.g.,* IP address) and the direction bit are the keys. This table contains the information of the default service set for the a given direction for each subscriber. This can be an LPM (longest-prefix match) table. If there is a miss in this table, the default action is to drop the packet. With a hit, a metadata field called *serv_set* will be set, which records the list of services this flow should traverse. Again, it will be further explained in Section IV-D

Application table: In this context, application refers to the remote communication endpoint, *e.g.,* web servers, as identified by the IP address, protocol, and port number. This table is used to modify the subscriber's default service set according to any static L3-L4 application policies. Based on this information, specific services can be excluded from the service set or added to it. Its action is also to set the *serv_set* metadata field. Note that that action of this table can add or remove certain bits that are set in the previous table.

The Path Status table: According to the bits set in the service set, the path status table determines which services have already been traversed and what is the next service in the chain. This is important because a packet may traverse the same perimeter switch multiple times, and it should be treated differently each time. The ingress port is sufficient to provide this information. If this table is reached, it means that the packet has arrived on a node port, connected directly to a service or router. The ingress port then tells us which service was just applied, if any, and it also tells us the direction. There is a global ordering of services in each direction (they may or may not be the exact reverse of each other). Based on the direction and the previous service, the service set is modified to exclude the previous service and all other services that precede it.

Next Destination table: As the last table, it uses the direction and the service set as a key. This is a TCAM-like table, with arbitrary bitmasks and rule priorities. Based on the direction bit, it essentially scans the bits in the service set according to the global service ordering in that direction. The first or highest-priority service it finds will be the next destination. If the service set is empty, the next destination will be either the

upstream or downstream router, depending on the direction bit. The next destination may be connected to the current switch or another one. If the destination is connected to a different switch, then the destination MAC address is set to the value corresponding to that service or router and the packet is transmitted out through an appropriate transit port. If the destination is directly connected, then the MAC addresses are updated as needed and the packet is transmitted out to the corresponding node port.

C. Handle dynamics with the Microflow Table

Microflow table is added to handle dynamically generated rules. Usually operators have policies statically determined by the subscriber and application identities. These rules can statically be pre-programmed on the Subscriber table and the Application table. But in real time, operators may want to add policies dynamically, or add more specific policies, or higher priority policies. Moreover, policies may be added according to the results of another middlebox, *e.g.,* DPI. We add the Microflow table to accommodate such dynamics.

If there is a hit in the direction table, the next table to be consulted will be the Microflow table. The key for this table is the direction bit together with the 5-tuples (source and destination IP address, IP protocol field, and TCP/UDP source and destination port) of the packet. The table contains exact-match entries used for selective complex steering of specific TCP/UDP flows. If there is a hit in this table, the next two lookups will be skipped. Thus, the rules in the Microflow table have higher priority than the rules in the Subscriber and Application tables.

D. Encode service chaining with metadata

Metadata is used in OpenFlow 1.1. to communicate the information among different tables and associated actions [8]. Intermediate results from one table are communicated to other tables using some metadata, which can be used as part of a subsequent lookup key or be further modified later.

We introduce two new types of metadata, the direction bit that represents the direction of the flow and the set of inline services to be applied for the flow under process, called *service set.* This service set is encoded as a bit vector, one bit for each possible service. More sophisticated encodings can be used to enable more advanced features such as load balancing over multiple service instances. OpenFlow 1.1 supports 64-bit metadata field [8]. Our design requires one bit for the direction and leaves up to 63 bits for encoding the service set, allowing a maximum of 63 distinct services. The format of the metadata is shown in Figure 4, together with an example. The metadata field can be applied with arbitrary mask and is updated as below:

$$new\_metadata = (old\_metadata \& \neg mask) | (value \& mask).$$

The $value$ are the bits to be set and the mask is used to select these bits, which can be arbitrary 64 bit numbers.

The first bit indicates whether it's upstream (0) or downstream (1). The following N-bit vector defines the service set, encoding for N number of services. The encoding in this example specifies that this packet should traverse the VS, DPI, and content cache. In the datapath, this metadata is set, then

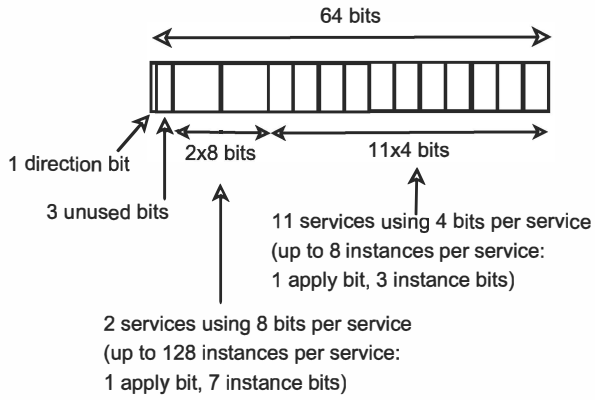

Fig. 5. Encode Metadata to handle load balancing

modified, and finally used to search for the next service to be applied to this packet.

1) Load balancing across multiple service instances: If a single middlebox is insufficient to deliver the required throughput, multiple instances of that service can be connected to the steering network. We use the metadata fields to distribute traffic without the need for a separate load balancer.

The controller can re-define the format of the bits in $serv\_set$ metadata for load balancing purposes. The current encoding can be seen as using one bit to represent a service (indicating whether or not it should be applied), and there is only one instance of each service. This encoding can be extended such that the services can additionally have an instance identifier included in the metadata as shown in Figure 5. If n bits are allocated to each service for such an identifier, then up to 2^n instances can be represented. With a fixed number of bits available, this is a trade-off between the maximum number of services and the number of instances per service. Each service can use a different number of bits for the instance identifier.

E. Example

We finally illustrate the whole system through an example using the same topology as in Figure 3. A subscriber Bob has a subscriber policy: service {S1,S3} for upstream traffic, *i.e.,* traffic sent from the subscriber to the core network should go through first $S1$ then $S3$. On the other hand, the operator wants to impose two application policies: for YouTube traffic {+S2,-S3}, and for VoIP traffic {-S1}.

Table II shows the table entries of the perimeter switch OF2. The only table that is not represented here is the Microflow table, as it is not derived from the static configuration. Below, an example is given on how the metadata field is matched or updated. This example is based on a 5-bit metadata field (one direction bit, 4 service bits) with the most significant bit representing the direction. Let us walk through this example of Bob watching YouTube (Bob at 1.2.3.4 and the YouTube server at 4.3.2.1/32 on port 80). The metadata is defined as a five bit vector $[DS_1S_2S_3S_4]b$ for upstream, where D means the direction bit and the rest of the bits represent services, with the letter b indicating the value in binary format. We pick two tables as examples to show how the metadata is modified.

The Direction table sets determines the direction based on the ingress port. Value 1 represents the upstream direction and

Direction Table

Ingress port	Action
1	dir=down
2	dir=up
3	dir=down
4	dir=up

MAC Table

Dst MAC	Action
S2 MAC	set srcMAC, output to port 1
S3 MAC	set srcMAC, output to port 3

Subscriber Table

Direction	IP Addr	Action
Up	Bob	$serv\_set = \{S1, S3\}$

Application Table

Dir.	IP Addr	Proto	Port	Action
Up	YouTube server	TCP	80	$serv\_set = serv\_set - \{S3\} + \{S2\}$
Up	VoIP server	*	5060	$serv\_set$-=$\{S1\}$

Path Status Table

Ingress Port	Action
1	$serv\_set$-=$\{S2\}$
3	$serv\_set$-=$\{S2, S3\}$

Next Destination Table

Dir.	Service set	Action	
Up	$S2 \in serv\_set$	set dstMAC=S2's MAC	set srcMAC=port1's MAC, output
Up	$S3 \in serv\_set$	set dstMAC=S3's MAC	set srcMAC=port 3's MAC, output

TABLE II. EXAMPLE OF TABLE CONTENTS ON OF2

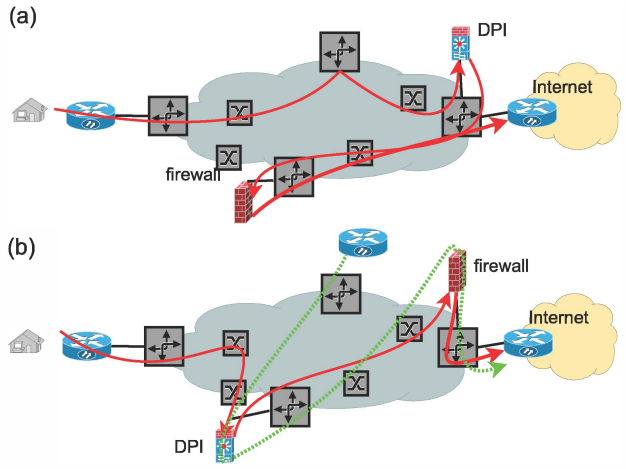

Fig. 6. Motivating example of service placement

0 represents the downstream direction. Assuming that a packet coming from port 2 matches the second rule, the metadata field would be updated according to Equation 1. For example, to set the first bit to 1, it uses mask $10000b$ to select the first bit to operate on the old metadata 0 and set its value to 1 using $10000b$. Thus, metadata is:
$(0\&01111b)|(10000b\&10000b) = 10000b$.
Note that in this table we will only consider the direction information, regardless of the subscribers. Next, we perform a lookup in the Subscriber table using the subscriber information. Using both the direction bit (Up) and the Subscriber IP address, the first rule is selected, *i.e.,* service S1 and S3 to be added. The old metadata is $10000b$. Since it is the first time to set the service set, all the services are selected using mask $01111b$ and then only S1 and S3 bits are set to 1 using $00110b$. The metadata is
$(10000b\&10000b)|(01010b\&01111b) = 11010b$.
We use these two examples to illustrate how the metadata is set and used to transfer information between tables.

V. SERVICE PLACEMENT MODULE

Given the ability to perform flexible routing across services demonstrated above, we do not require the services to be placed in a specific order. Rather, they can be placed anywhere in the network. However, despite this freedom, we discover that certain placement strategies are better than others when affecting user performance. In this section, we discuss the inline service placement problem and provide one heuristic to placing services in the network, with the objective of minimizing the average time it takes for the subscribers' traffic to go through all required services. Note that we do not aim at finding an optimal algorithm for service placement in this work. Our contribution is to formulate the problem and demonstrate its importance.

In what follows, we first explain how different placement strategies affect performance. Then, we formulate the problem as a graph problem, and propose a greedy algorithm that minimizes the average latency of paths that subscribers' traffic traverse in the network.

A. Motivating example

We motivate the need for an intelligent (non-random) service placement scheme through an example. In Figure 6 (a), we assume that all the traffic enters the network from the left router and exit the network on the right. The ordering of the inline services to be traversed is determined according to subscribers' policies: 90% of the subscribers require their traffic to first traverse the DPI box and then traverse the firewall. This service order is specified in the subscribers' service contract. The remaining 10% of traffic requires going through FW first before DPI. In this case, the majority of traffic needs to first go through the network to reach the DPI box, and then go all the way through the network again to reach FW, and finally, to reach the outbound router, shown in the red solid curve (We assume that a shortest-path routing scheme is used for routing traffic between any pairs of services).

Figure 6 (b) shows an alternative service placement strategy, which moves the DPI closer to the ingress point; hence, makes the path of majority of the flows shorter.

However, the problem of determining the best locations for all the services is not always this trivial. For example, in Figure 6, if there are another ingress point and egress point, and if the subscribers require a different ordering of the inline services (the green dashed curve), then it will be more complicated to find the optimal solution.

B. Problem definition and a greedy solution

We formulate the service placement problem as an optimization problem of minimizing the delay or distance to be traversed by all subscribers' traffic. Let $G = (V, E)$ denote the network graph, with node set V representing switches, and edge set E representing the links. Graph G is an undirected

symmetric graph with weighted edges, *i.e.,* if $(u, v) \in E$, then $(v, u) \in E$. Each edge is associated with a delay value $d_{u,v}$. It can be simply $d_{u,v} = d_{v,u} = 1$, meaning that the delay is approximated as the hop count.

We denote the set of inline services as $S = \{s_1, s_2, \ldots s_h\}$, where each service s_i is one type of service to be placed in the network. Each subscriber has its own requirement on the order of inline services to traverse. For example, subscriber i's requirement is $r_i = ing_i, s_1, s_3, s_5, s_2, egr_i$. Here ing_i and egr_i are the ingress and egress locations of the subscriber's traffic.

The objective is to find a subset M ($|M| = |S| = h$) of the locations among all candidates $N = |V|$, and place the services in these selected locations so that the total delay for all the users are minimized. Given a service placement L, for each service sequence r_i, the total delay is calculated as the sum of the (shortest-path) delay between all consecutive services, plus the delay between the ingress point and the first service and the delay between the last service and the egress point.

If the placement problem can be solved offline, an exhaustive search can be used to find the optimum solution. But if there is a need to periodically change services locations in the real time, *e.g.,* in a data center environment with virtualized appliances, or if the network size is very large, then an efficient approximation algorithm is needed. In the following, we propose a heuristic algorithm to solve the placement problem. Though we are aware that it may not produce optimized results, our results show that it can already outperform a naive approach.

In our heuristic algorithm, we form a service dependency graph between all services. Two services are dependent if they appear consecutively in the service chain requirement of some subscribers, and the degree of this dependency is determined based on the amount of traffic (or the number of subscribers) with such service chain requirement. In a greedy approach, we pick the service with the highest dependency degree, find the best location for that service, and remove that service from the service list. We do this iteratively until all services are placed in the network. Due to limited space, we omit the details and the pseudo code of our algorithm, but some results will be presented in Section VI.

VI. Implementation and Evaluation

We implement StEERING in a testbed as shown in Figure 7. Our control plane is built using the NOX OpenFlow controller [12] and the data plane is implemented using programmable EZchip NP-4 network processors [13]. We deploy four middlebox services: two Linux IPTables based firewalls, a commercial proprietary DPI box, and the Apache Traffic Server [14] as the content filter. An Authentication, Authorization and Accounting (AAA) server connects to the controller using an external Ethernet switch for out-of-band connectivity. We use FreeRADIUS [15] as the AAA server, providing subscriber information to the controller. The controller is also connected to the Broadband Network Gateway (BNG) server so that it can program the rules as the subscribers login. We have demonstrated this prototype in a major mobile computing conference. In this section, we first describe the details of each component followed by evaluation results.

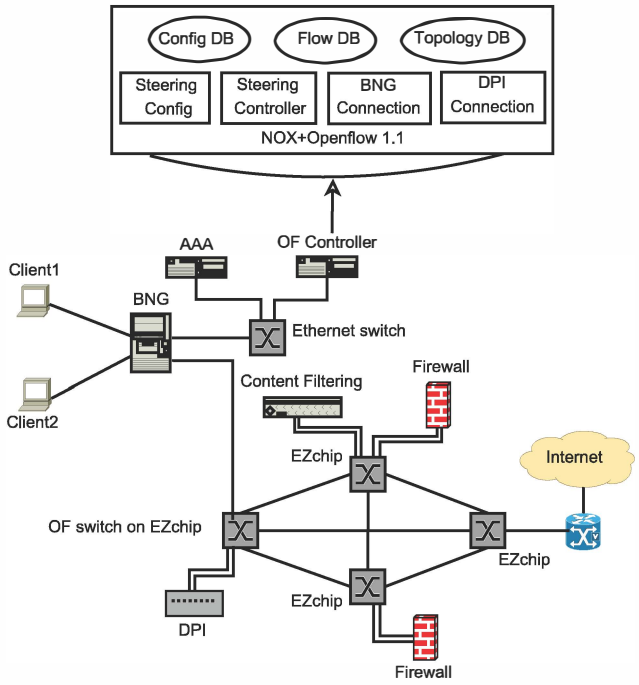

Fig. 7. Prototype diagram

A. Control plane

We build the control plane on top of the NOX OpenFlow controller [12]. NOX provides a programmable interface for network management and control applications. It programs OpenFlow switches via OpenFlow protocol messages. We implemented four modules that run as application on top of NOX, as shown in Figure 7.

The *Steering Config Module* maintains databases of the network topology, the service policies as well as the global ordering between services.
The *Steering Controller Module* translates traffic steering policies into rules that are pushed to the perimeter switches using the OpenFlow protocol. It runs Dijkstra algorithm to determine the shortest paths between any two endpoints. On the one hand, it receives requests from other modules to install new policies. On the other hand, it collects flow statistics which are input to the service placement module.
The *BNG Connection Module* keeps track of events related to subscribers. It registers itself with the BNG to receive notifications whenever a subscriber logs in. These notifications include the subscriber's IP address and its service policy. Upon receiving a notification, this module interacts with the config module to install rules on the Subscriber table.
Finally, the *DPI Connection Module* is responsible for complex traffic steering. It listens for flow notification messages sent by services such as DPI and reacts by pushing new rules that will dynamically "re-steer" a flow to bypasss certain services. These rules are computed by associating the flow described in the flow notification message (*i.e.,* 5-tuple) with the originating subscriber as well as its assigned service chain.

Rule timeout: To reduce the number of rules on the switch, when receiving an explicit logout message from the AAA, the BNG Connection module reports an event to the DPIConnection module to purge all entries from the micro-

flow table and subscriber table associated with that subscriber.

B. Forwarding plane

The decision to develop our *own* implementation of an OpenFlow 1.1 switch was based on the fact that there are no hardware solutions on the market today that support multiple tables. The forwarding plane is built on a hardware platform equipped with an EZchip NP-4 network processor [9]. This processor has a programmable pipelined architecture capable of processing packets at a line rate of 100 Gbps. NP-4 are specifically optimized to process packets and provide more flexibility than a networking ASIC while providing substantially higher performance than general purpose processors from Intel or AMD.

In addition to the NP-4, the hardware platform used to develop our prototype is equipped with a host processor (PowerPC) used to run initialization, control and management tasks. The host processor runs a small footprint distribution of Linux on which we installed a version of Open vSwitch (OVS) [16] that we modified to support OpenFlow 1.1. We use Open vSwitch as an middleman between the OpenFlow controller and the NP-4 processor. It communicates with the controller using the OpenFlow protocol and we develop a hardware abstraction layer that translate Open vSwitch functions to populate tables and handling packets in/out into system calls to the NP-4.

The EZchip NP-4 consists of Task Optimized Processors (TOPs), each one dedicated to a specific task. TOPs can be seen as processing cores where multiple packets are processed in parallel. The NP-4 pipeline contains four types of TOPs:

- TOPparse: Parse and classify the packets. At this stage, keys are built based on the packet headers and sent to the next stage. More than one key can be derived from the packet.
- TOPsearch: Perform lookups in the corresponding data structures in memory. The NP-4 version of EZchip contain two search stage identified as TOPsearch I and TOPsearch II. In particular, the lookup for MAC table, Application, Subscriber, and Microflow table are performed in parallel in the TOPsearch I phase, as they can operate on the metadata field independently. The Path Status and Next Destination table are looked up in TOPsearch II.
- TOPresolve: Analyze the lookup results and determine what to do next with the packet.
- TOPmodify: Apply the modifications to the packet (e.g. push header, set IP address, etc.) and mark how to output the packet if applicable.

The NP-4 is also equipped with a learning TOP (TOPlearn) and an integrated traffic manager that provides end-to-end dynamic traffic control. We did not use these two features in our implementation.

C. Interfaces

There are two types of interfaces. The first type is the OpenFlow 1.1 protocol between the StEERING Controller (StC) and the OF switches. The second type is the

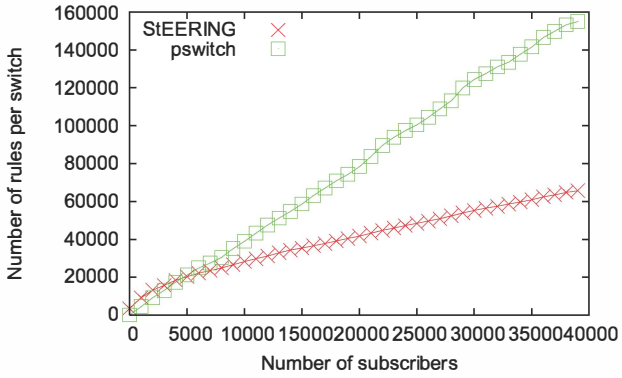

Fig. 8. Number of rules as subscriber increases

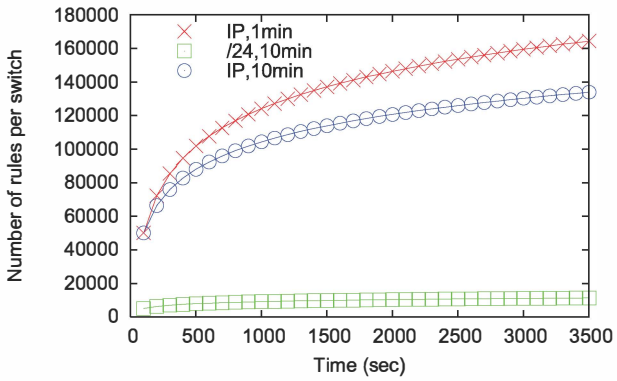

Fig. 9. Table size increases over time

controller-service protocol, called the StEERING Service Protocol (StSP). It is used for services such as DPI to send flow notifications to the StC. The notification will include information to uniquely identify the flow (typically the 5-tuples) as well as the traffic type (e.g. SIP, HTTP URL, etc.) known to both the controller and service. Based on the traffic type and the configured application and service policies, the controller will determine which service path should be applied to this recognized flow and sends the appropriate flow modifications to the perimeter OF switches to re-steer the traffic accordingly. Subsequent packets of this flow will be steered through the Service Delivery Network according to the new rules. Similar to the OpenFlow interface between the controller and switches, the StSP interface will run directly over TCP or be encrypted using TLS (Transport Layer Security).

D. Solution Validation

In this section, we provide preliminary benchmark of our system as well as scalability analysis.

1) Switch Scalability: The number of rules installed is limited by the switch capacity. The rules in StEERING increases linearly as the sum of subscribers and applications, while the pswitch approach grows as the product of subscribers and applications. We show the difference between the two approaches using a simple simulation. In the topology in Figure 7, we define 100 applications, each application j request traversing m_j number of services, $m_j \in [1, 4]$. And we assume that each subscriber i sends flows for n_i applications, $n \in [1, 10]$. At the same time, each subscriber requests to

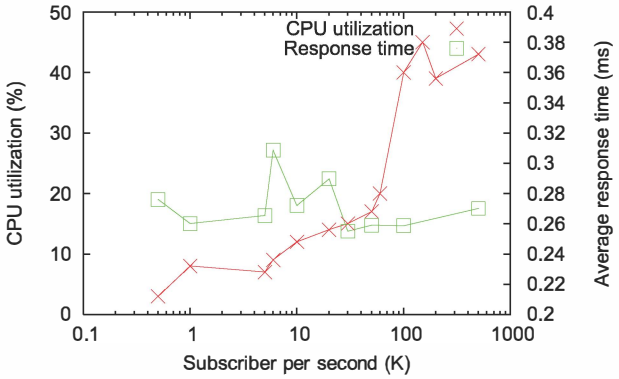

Fig. 10. Controller utilization and Response time

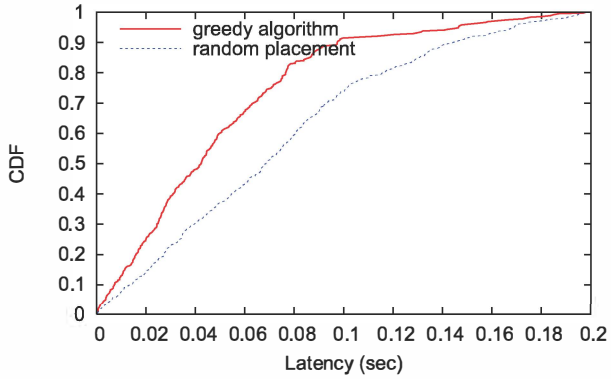

Fig. 11. Delay distribution with 3 services deployed

traverse 2 services. Then we simulate when a subscriber i arrives with an application j, the number of rules installed in StEERING and in a system like pswitch [1]. Figure 8 shows that the slope of our method is much smaller than the pswitch approach. The advantage comes from the separation of subscriber policies from application policies, allowing the independent scaling of rules in each dimension.

Next, we show that StEERING can support flows in the real network. We validate this by replaying the unsampled packet traces collected from a European municipal network. It contains around 200K subscribers and 24 hours of data. We use the data as input and emulate the process of rule setting up and tearing down. We assume that if the subscriber is not active for n second, the rule is removed from the switch. We try $n = 1min$ and $n = 10min$ timeout threshold and show different ways of defining flows in Figure 9. We can see that as the time evolves, the increasing of table size slows down. A larger timeout interval results in fewer rules and faster convergence. As expected, grouping IP addresses to /24 prefixes helps reduce the rules significantly as well. The number of rules can be supported by our commodity EZchip based switches.

2) Forwarding performance: Using a traffic generator, we were able to measure the performance of our prototype at full line rate of 20 Gbps full-duplex with minimum packet size of 64 bytes, which represents 29.8 Mpps. This throughput is, in fact, limited by the capability of our current traffic generator. In the future, we plan to test it with a more powerful generator. This experiment was executed with hardware lookup tables populated to 10,000 entries for the Microflow, Subscriber and Application tables. We test large number of entries in these tables, as the number of subscribers, applications, and dynamic flows can be large in real networks. For other tables, it is unlikely to grow very large, *e.g.,* number of services or ports. As expected, the throughput performance is independent of the number of entries per tables, *i.e.,* the throughput is constantly 20 Gbps as the entries increase. The EZchip hardware platform provide deterministic characteristics for each packet being processed if the number of clock cycles required is kept within the cycle budget available. Our implementation is built in such a way that every packet will necessitate the same amount of processing and memory accesses.

3) Controller performance: We built an application on top of the NOX controller. In this section we show that the

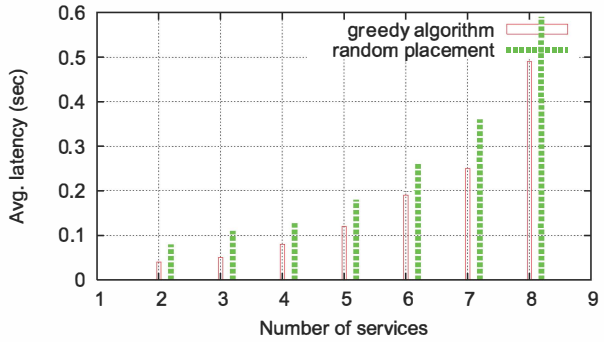

Fig. 12. Average delay as number of services increases

StEERING component does not introduce much overhead. We profile the CPU utilization as the subscriber incoming rate increases. In the left y-axis of Figure 10, we vary the subscriber incoming rate and measure the average CPU utilization of the controller. Since the controller needs to program the policies for each new subscriber, its CPU utilization increases with the subscriber rate, but after 100K subscribers per second, the utilization becomes stable around 45%. After this, the network becomes the bottleneck so the computation resource is stable.

Another metric of controller performance is the response time, *i.e.,* the time elapsed from when a request (flowmod message) comes in until the time when rules are sent out. We show the average response time across all subscribers as the rate increases in the y-axis of Figure 10. Clearly, the response time is not significantly affected by the incoming rate.

4) Evaluation of service placement algorithm: We evaluate the service placement algorithm in Section V using the Abilene network topology [17] and the traffic from the packet traces mentioned above. We first divide the source addresses to 11 groups, assigning each group to one PoP in Abilene network topology as the ingress point of this group of sources. Similarly, we divide all the destination addresses to 11 groups and assign each group to a PoP, as their egress points. We select the top 10 applications that account for most traffic, and then assign n number of services for each application. Given this input, we use the greedy algorithm to compute the locations for these n services, as well as a random deployment. In Figure 11, we show the delay distribution for all the flows. For 50% of the flows, our algorithm can achieve 40ms latency while

the random scheme needs 80ms. We also vary the number of services in Figure 12. The advantage of the greedy algorithm is more significant with more services deployed. It should be noted that the greedy algorithm is used to efficiently solve the problem online, however if resource is not a limitation, other algorithms should be investigated.

VII. Related Work

Our work contributes to and draws inspiration from a rich corpus of work in middlebox management, Software-Defined Networking (SDN) and cellular network management. Studies have measured the end-to-end impact of middleboxes [18] and interactions with transport protocols [19]. Reference [20] proposes a network management plane. Sekar *et al.* [4], [21] propose software-centric middlebox implementations running on general-purpose hardware platforms with open APIs. CoMb [3] is a new middlebox architecture for consolidating middlebox deployments. Sherry*et al.* proposes a method to deploy middleboxes in the cloud [2]. There are increasing interests on middleboxes in cellular networks [5]. Our work is orthogonal to these work as we propose a method of routing traffic through deployed middleboxes.

Our work builds on top of the existing proposals of SDN [22], [11], [8]. It is an application to demonstrate SDN's support on flexible routing. There have been many applications building upon SDN. For example, Flowvisor is designed to provide network virtualization [23] and Wang *et al.* proposes a scheme for load balancing of web servers [24]. In our problem, policy dependencies that arise in the context of inline service routing create new challenges for management and optimization unique to our context. A large body of work on improving the SDN programmability is orthogonal to our work [25], [26], [27], and thus can be integrated into StEERING in the future.

VIII. Conclusion

We presented StEERING for managing traffic through middleboxes in the mobile and broadband network. Built on top of SDN, StEERING can support efficient forwarding and scale to a large number of subscribers and applications. We demonstrated the functionalities and performance through a prototype with both software and programmable hardware supports. Using simulations, we showed the significant latency reduction that results from implementing our approximation algorithm for placing services in the network.

Acknowledgments

The authors would like to thank Frank Ruhl and Daniel Kirkham of Telstra's Innovation and Strategy team. The authors are also grateful to Erik Rubow for his input into the early stages of this work.

References

[1] D. A. Joseph, A. Tavakoli, and I. Stoica, "A policy-aware switching layer for data centers," in *Proc. ACM SIGCOMM*, pp. 51–62, 2008.

[2] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: network processing as a cloud service," in *Proc. ACM SIGCOMM*, pp. 13–24, 2012.

[3] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, "Design and implementation of a consolidated middlebox architecture," in *Proc. Symposium on Networked Systems Design and Implementation*, 2012.

[4] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi, "The middlebox manifesto: enabling innovation in middlebox deployment," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, 2011.

[5] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An untold story of middleboxes in cellular networks," in *Proceedings of the ACM SIGCOMM 2011 conference*, SIGCOMM '11, 2011.

[6] J. Sherry, "Future architectures for middlebox processing services on the internet and in the cloud," Master's thesis, EECS Department, University of California, Berkeley, Dec 2012.

[7] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Towards software-defined middlebox networking," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, 2012.

[8] "OpenFlow 1.1." http://www.openflow.org/wk/index.php/OpenFlow_v1.1.

[9] EZchip, "Ezchip np-4 product brief," 2011. http://www.ezchip.com/Images/pdf/NP-4_Short_Brief_online.pdf.

[10] Ericsson, "Smart Services Router." http://www.ericsson.com/ourportfolio/products/ericsson-ssr-8000-family.

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

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, July 2008.

[13] E. Technologies, "Network processor designs for nextgeneration networking equipment," Dec 1999.

[14] "Apache Traffic Server." http://trafficserver.apache.org.

[15] FreeRADIUS, "Freeradius: The world's most popular radius server." freeradius.org.

[16] "Open vswitch." http://www.openvswitch.org.

[17] http://www.internet2.edu/network/.

[18] M. Allman, "On the performance of middleboxes," in *Proc. ACM SIGCOMM IMC*, 2003.

[19] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend tcp?," in *Proc. ACM SIGCOMM IMC*, 2011.

[20] H. Ballani and P. Francis, "Conman: a step towards network manageability," in *Proc. ACM SIGCOMM*, 2007.

[21] Z. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, "Simple-fying middlebox policy enforcement using sdn," in *Proceedings of the ACM SIGCOMM 2013 conference*, SIGCOMM '13, 2013.

[22] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4d approach to network control and management," *SIGCOMM Comput. Commun. Rev.*, vol. 35, October 2005.

[23] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[24] R. Wang, D. Butnariu, and J. Rexford, "Openflow-based server load balancing gone wild," in *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services*, Hot-ICE'11, 2011.

[25] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," in *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, 2011.

[26] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *Proceedings of the ACM SIGCOMM 2010 conference*, SIGCOMM '10, pp. 351–362, 2010.

[27] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent updates for software-defined networks: change you can believe in!," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, 2011.