Application-aware Data Plane Processing in SDN

Hesham Mekky
University of Minnesota
Minneapolis, MN
hesham@cs.umn.edu

Fang Hao
Bell Labs Alcatel-Lucent
Holmdel, NJ
fang.hao@alcatel-lucent.com

Sarit Mukherjee
Bell Labs Alcatel-Lucent
Holmdel, NJ
sarit.mukherjee@alcatel-lucent.com

Zhi-Li Zhang
University of Minnesota
Minneapolis, MN
zhzhang@cs.umn.edu

T V Lakshman
Bell Labs Alcatel-Lucent
Holmdel, NJ
t.v.lakshman@alcatel-lucent.com

Abstract

A key benefit of Software Defined Networks is fine-grained management of network flows made possible by the execution of flow-specific actions based upon inspection and matching of various packet fields. However, current switches and protocols limit the inspected fields to layer 2-4 headers and hence any customized flow-handling that uses higher-layer information necessitates sending the packets to the controller. This is inefficient and slow, adding several switch-to-controller round-trip delays. This paper proposes an extended SDN architecture that enables fast customized packet-handling even when the information used is not restricted to L2-L4. We describe an implementation of this architecture that keeps most of the processing in the data plane and limits the need to send packets to the controller even when higher-layer information is used in packet-handling. We show how some popular applications can be implemented using this extended architecture and evaluate the performance of one such application using a prototype implementation on Open vSwitch. The results show that the proposed architecture has low overhead, good performance and can take advantage of a flexible scale-out design for application deployment.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations; C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.2.3 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Design, Measurement, Performance

Keywords

Software-Defined Networking; OpenFlow; Open vSwitch; Data Plane

1. INTRODUCTION

Software Defined Networking (SDN) is a new paradigm permitting application-aware management of networks. Two key aspects of SDNs are: (i) a flexible flow-based forwarding abstraction that can be used for programming the data plane(e.g., OpenFlow switches)using an open API, (ii) a logically-centralized control plane abstraction that can be used by network applications, network "apps", to perform network-wide operations without low-level configuration of individual network elements. SDNs are currently being deployed for managing large data center networks that need to be application-aware [8] and for wide-area application-aware traffic engineering [6, 7].

To keep the data plane simple and efficient, the current SDN architecture makes the switches stateless and acts only on layer 2-4 packet-header information even though many applications can benefit from a more extended architecture that permits stateful actions on higher layer information. Also, to avoid delays in flow-processing, pre-installation of flow forwarding rules in the switches is done when possible. When this is not possible, installation of rules right after examining the first packet at the controller (rather than examining multiple packets of a flow) is preferred.

While this architecture and mode of operation can efficiently meet the needs of a variety of networking tasks, like routing or address rewriting, a variety of other ubiquitous functions will benefit from an extended architecture permitting stateful operations and use of information deeper in the flow. The widely used NAT function needs to maintain state for keeping track of available IP addresses and ports. A content-aware request routing (L7 load balancing) app redirects client requests to an appropriate server from a server pool using information about the requested content. This requires inspecting packets beyond L4 headers. Also, this information is not available in the first packet of the request, and so the request routing rule cannot be set up in the switch even after examining the first packet of the flow.

Clearly, stateful processing using higher-layer information is possible using the current SDN architecture – packets belonging to a flow can be forwarded to the controller till the corresponding controller app has enough information to install the needed rules in the switches. This sacrifices efficiency and slows down the system. Alternatively, dedicated middleboxes can be used to perform stateful operations but this limits the flexibility and benefits of SDN.

This paper proposes an extended *application-aware* SDN architecture that generalizes the standard OpenFlow forwarding abstractions to include stateful actions that use L4-L7 information. For efficient implementation despite the more general packet-handling, we need to keep some application logic locally at the switches (instead of limiting the application logic to only the controller). The switches in our system are augmented to have a stateful app processing capability that uses higher layer information in packets. The controller application installs app-specific packet-processing actions in the augmented switches' *app table* (which we define in Section 2.3). This table is similar in spirit to the OpenFlow flow table. However, new flow forwarding rules for incoming packets can be generated locally within the switch by executing the packet processing logic installed by the controller. This architecture has several advantages:

- In most cases, the packets need not go to the controller for determining flow actions. This reduces the switch-to-controller delays involved in flow establishment.
- In case of service chaining, application-aware switches can replace dedicated appliances and allow packets to be treated on their shortest paths, eliminating inefficient traffic detouring. Moreover, the same switch can apply the processing logic of multiple apps, potentially avoiding the need of tracking flows between apps.
- It enables uniform central control on application-aware flow processing with a built-in scale-out data plane.

Examples of new data plane functions that can be implemented using this extended architecture include TCP splicing, NAT, L7 server selection, firewall and so on. We use examples to illustrate how these new data plane functions can be composed and controlled by various network apps running on top of the controller to enable application-aware decision making. Together, they realize *in software* many common application-level flow processing and decision making functions. We present a prototype implementation of an example of application-aware service on top of Open vSwitch (OVS). In our implementation, the switches incorporate the ability to route requests to different servers based on the requested content. They fall back on the controller only when the requested-content to content-server mapping is not locally available. Through experiments, we show that the extra packet processing latency caused by the apps on the switch is low, and the response time to serve uests is fast.

2. SYSTEM ARCHITECTURE

Virtual switches, such as OVS that run in hypervisors, have become commonly available in data centers. They offer an ideal platform for our design since they open up the opportunity of inserting app processing logic in the data path via software extensions. We design a solution that extends OVS to be application-aware while conforming with the existing OpenFlow. We also attempt to make the design modular, with a clean interface to the existing OVS implementation, that allows new apps to be plugged in easily.

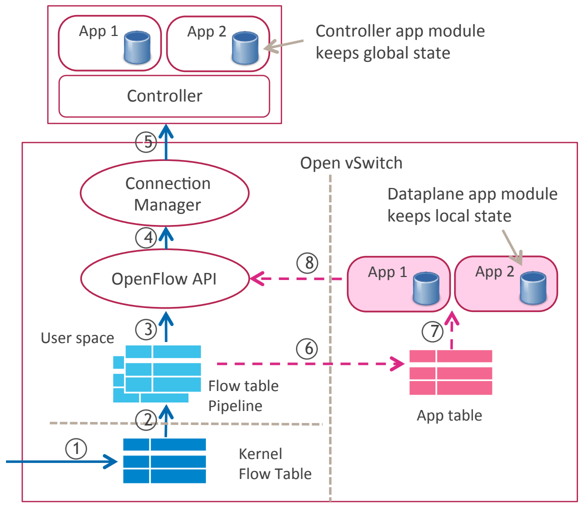

Figure 1: System Architecture

2.1 Existing Open vSwitch (OVS)

Figure 1 shows the high-level architecture of the system. The four main components of the existing OVS implementation are shown on the left, including connection manager, OpenFlow API, user space flow table pipeline and the kernel flow table. The flow table pipeline contains one or more flow tables, each with flow rules that specify how the matching packets should be processed. A packet can be processed by multiple flow tables by following the "goto" instruction in the rules. To speed up packet processing, the kernel flow table caches the flow actions so that active flows can be directly processed in the kernel.

The solid blue arrows (Steps 1-5) show how the packets can be processed by the application at the central controller in the existing OVS implementation. An incoming packet is first matched with the kernel flow table. If there is no match, the packet is sent up to the flow table pipeline in the user space. A *table-miss* rule is contained in each table in the pipeline, which tells what to do when the packet does not match any other rules in the table, e.g., to be sent to another table or to the controller. In the latter case, the OpenFlow API calls the connection manager to encapsulate the packet in a `Packet_In` message and send it out. When the controller receives the `Packet_In` message, one or more applications running on the controller may process the message and install rules in the flow table pipeline via a `Flow_Mod` message so that later packets of the flow can be processed on the switch.

2.2 Design Choices

To make the data plane application-aware, our basic idea is to intercept the packet before sending it to the controller, so that application processing logic can be applied to the packet locally without leaving the switch. We have considered the following three options:

1. Intercept the messages between the controller and OVS at the connection manager, and call application handling logic from there.
2. Intercept the packet after the flow table lookup, but before messages are generated.

3. Add *app actions* directly into the flow table, so that packets can be processed by calling such app actions when they match the rule.

Option 1 can provide nice isolation between applications and the existing OVS code. However, it may lead to significant redundancy in packet processing. For instance, when the application intercepts the Packet_In message from the switch to controller, it needs to decapsulate the already encapsulated message and recover the original data fields. The application also needs to implement its own flow table so that it can look up the policy rules for the packet. Both message decap/encap and flow table operations add extra development cost and performance overhead.

Option 3 is the most efficient since the app actions are treated the same as native OpenFlow actions. But the main issue is that in order to achieve fast processing, app actions should be implemented at both user and kernel space. Since application logic is typically much more complex than simple actions such as drop/forward and it may require keeping persistent application state, this approach requires significant code change in OVS. Such app actions may also be more time consuming and make packet processing less predictable. Nevertheless, this is a viable approach if the action is simple; one example is given in Section 3.

We believe Option 2 is the best choice for most applications since it avoids drawbacks of both 1 and 3. In order to intercept the packets right after the standard flow table matching, we use the last flow table in the pipeline as a special *app table*, shown in Figure 1. All table-miss rules with the action of "output to controller" are modified to "goto app-table", so that all packets that are originally processed by the controller will instead first pass through the app table. Unlike the standard flow table, special *app actions* can be called from the app table to handle the packets. Such app actions are implemented as application functions. The dotted arrows (Steps 6-8) in Figure 1 show how the packets are "detoured" to the app table in the new architecture. Next, we discuss details of the design by using this option.

2.3 App Table and App Actions

The app table is operated in the same way as the standard OpenFlow flow table. Packets are matched with the rules contained in the entries, and the corresponding actions are executed for the corresponding matching rule. If matching fails in the app table, the table-miss rule can be used to send the `Packet_In` message to the controller. The SDN controller installs and removes rules from the switch app table by using the standard OpenFlow `Flow_Mod` command.

The app actions are specified as *vendor actions* in the OpenFlow protocol [3]. The app table rules run only at the OVS user space; they are not installed in the kernel module to avoid slowing down the fast path packet processing of Open vSwitch. An app action may do any combination of the following operations: (1) determine the actions to be taken for the current packet; (2) modify its local persistent state; (3) generate or modify the rule set to be installed in the standard OpenFlow flow tables for processing this flow for future packets; (4) remove flows from standard OpenFlow flow tables; (5) generate a packet out to other switches; and (6) send Packet_In, Flow_Removed, or app update vendor messages to the controller.

Operations (3) to (6) are completed by calling the set of API functions exposed from the OpenFlow API module (Figure 1 Step 8), which include `add_flow`, `del_flow`, `packet_in`, `packet_out`,, `flow_removed`, and `app_update`.

Each app also provides a message callback handler, so that it can be called from OpenFlow API when the controller sends messages to the app. Message exchanges between the controller and local app modules are encapsulated in the OpenFlow vendor messages.

2.4 App Chaining and Execution

Multiple apps can be chained together to implement complex network services. The order of app actions are determined based on the service policies. Each app action may modify the packet header and its metadata, and also generate or modify the instructions and actions that are to be installed in the standard flow table. Such instructions and actions are temporarily stored in the *scratch pad*, an array that is shared by all apps. The first app that processes a flow allocates an entry for the flow in the scratch pad. This entry stores the rule set that is generated by the apps for this flow. The last app action in any app chain is always Install, which installs the rule set into the standard flow tables. Install also frees the allocated flow entry in the scratch pad. The flow's metadata contains the index of the flow in the scratch pad and two flags: *break* and *flow removal*.

An app can prevent later apps in the chain from execution by using *break* flag. For example, the firewall app can decide to drop the packet and set *break=true*, so that the following load balancer app can ignore the packet. Note that all the app actions are still invoked – we do not change the OpenFlow semantics or the OVS implementation, but the apps ignore the packet when they see *break* being set. However, the Install app always executes regardless of the flag, unless the rule set in the scratch pad is empty. Usage of *flow removal* flag is described next.

2.5 Flow Installation and Removal

A flow entry in the standard flow table may be installed by the switch apps or the SDN controller. Ideally, only the installer of each flow should be informed of the flow removal, so that appropriate action can be taken to keep track of the flow state when necessary. To do so, we make a small change in the Flow-Removed message generation function: instead of generating the message to the controller, it generates a fake packet that conforms to the flow definition but bears the *flow removal* flag in its metadata. This packet is then submitted to the app table. Since this packet will match the same app table rule that has generated the flow entry, the corresponding responsible apps will be invoked, which can then perform their own flow removal handling. Unmatched packets will be handled by the table-miss rule. We also change the table-miss action to a special Unmatch app action, which sends regular unmatched packet using `Packet_In` message to the controller, and converts the fake packet to Flow-Removed message and sends it to the controller. The latter case is for handling removal of controller installed flows.

2.6 An Example: Firewall & Load Balancer

We use the following example to illustrate how the new architecture works. Suppose the network policy requires *"any web traffic to server x needs to go through the firewall and then the load balancer"*. The firewall rule specifies that only web traffic is allowed, and maximum number of active TCP connections is 1000. The load balancing rule is to distribute the load to servers s1 and s2 by hashing according to the source IP address.

To implement this policy, the controller inserts the following rule to the app table: `(dst_ip=x, tcp, dport=80: fw,lb,fwd,install)`, where fw, lb, fwd, and install will call Firewall, Load Balancer, Forward, and Install app functions, respectively. When a new flow `(src_ip=a, sport=6000, tcp, dst_ip=x, dport=80)` arrives, it will be delivered from the kernel to the first flow table in the user space. Then the table-miss rule will submit the packet to the app table. Since it matches the flow, the four apps will be called in sequence:

Firewall: It keeps track of the number of active flows by using a static local variable `nFlow`. Suppose current `nFlow < 1000`, then the following flow rule is generated and stored in the scratch pad: `(src_ip=a, sport=6000, tcp, dst_ip=x, dport=80: null)`; and nFlow is incremented. The empty action set indicates the flow is accepted but no action is defined. Firewall also sets "Continue" flag and stores the rule's index in the packet metadata.

Load Balancer: It selects the server's address using hash(`src_ip`). Suppose the hashing result is s1, then the flow rule in the scratch pad is replaced as: `(src_ip=a, sport=6000, tcp, dst_ip=x, dport=80: set dst_ip=s1)`. In addition, it changes the `dst_ip` field of the current packet header from x to s.

Forward: It looks up its routing table to find the output port `pt1` based on s1. It then updates the rule in the scratch pad as: `(src_ip=a, sport=6000, tcp, dst_ip=x, dport=80: set dst_ip=s1, out pt1)`.

Install: It retrieves the rule from the scratch pad and calls API function `add_flow` to install the flow into the flow table pipeline. It also calls `packet_out` to send the current packet out to `pt1`.

As a result, the apps have jointly generated the flow rule according to the network policy without going to the controller. This rule stays in the flow table pipeline and may be also cached in the kernel.

Note that Firewall drops the flow when `nFlow > 1000`. In that case, it sets "Break" flag in metadata and generates the following rule: `(src_ip=a, sport=6000, tcp, dst_ip=x, dport=80: drop)`. This causes Load Balancer and Forward to ignore this packet. At the end, Install will install the drop rule into the flow table pipeline.

When this flow expires, a fake packet with fields `(src_ip=a, sport=6000, tcp, dst_ip=x, dport=80)` is triggered and submitted to the app table, with metadata containing the *flow removal flag.* All apps will be executed so they can update their current local state. In this example, the firewall will decrement `nFlow`.

3. EXPERIMENTAL STUDY

In order to validate the proposed architecture, we build a proof-of-concept prototype of the Content-Aware Server Selection app. We use OVS v1.10.0 and Floodlight as the base for the switch and the controller, respectively, and modify them to instill application awareness. We extend the OVS kernel module to implement TCP seq/ack rewriting (i.e. splicing) action, since it is simple and has to be applied to every packet. More complex but less frequently invoked actions including TCP Handshake and Server Selection are implemented as app actions at the user space.

3.1 Content-aware Server Selection

In this application a client's web request to a virtual IP address, VIP of a server pool, is redirected to an appropriate server dynamically based on the URL of the request. This is done in two steps. First, the datacenter gateway router distributes the client requests to multiple front-end OVSes by using ECMP. Second, the front OVS selects the server based on URL of the request, and forwards the packet through the back-end OVS to the server. On the return path, the packet is forwarded by the back-end OVS directly to the client. This is similar to the layer 4 load balancing solution proposed in [10], except that here we enable server selection based on layer-7 information.

Figure 2 shows a simple mapping of the app onto the proposed architecture with intermediate message flow. For simplicity, it omits the gateway router and first stage ECMP based load distribution, and just shows two Open vSwitches: SW1 as front-end and SW2 as back-end. Standard OpenFlow flow tables are shown in solid lines, and app tables are shown in dashed lines. In our proof-of-conecpt implementation the Server Selection app resolves VIP to one of the servers S1 or S2. Initially the controller adds default rules to submit table-misses to the app table in both SW1 and SW2. In addition, the Server Selection controller app adds flow entries into the app table as shown in the figure.

When a client request arrives at SW1, the app table rule fires and SW1 performs the TCP Handshake with the client to advance to the HTTP GET request. During this phase (steps 1-3), SW1 uses SYN cookies to preserve the connection state, and stops execution of Server Selection by inserting a "break" after packet-out-ing the SYN-ACK packet. Only after the HTTP GET request packet arrives, the execution "continues" to Server Selection which extracts the requested URL and uses that to find the appropriate server from the server pool. If the mapping is not available locally at the switch, it sends the packet to the controller, which resolves and returns the mapping back to SW1 (say S1), as shown in steps 5 and 6 of Figure 2b. Server Selection function writes a forwarding rule for the rest of the connection into the standard flow table of SW1 that rewrites the destination VIP to S1 and forwards the packet towards S1 (see Figure 2b).

When the switch in front of S1 (i.e., SW2) receives the packet, it matches the app table rule. It then invokes the TCP Handshake function that plays back the handshake with S1 based on the header fields of the packet. During this phase, right after receiving the SYN-ACK from S1, it can compute the deltas used for TCP splicing, and therefore it installs the appropriate rewrite rules in the standard flow table of SW2 (see Figure 2c). When S1 replies, SW2 performs TCP splicing to adjust the sequence and acknowledgement gaps for the connection to go through transparently between the client and S1. SW2 is also tasked with SNAT-ing S1's address to the virtual address VIP.

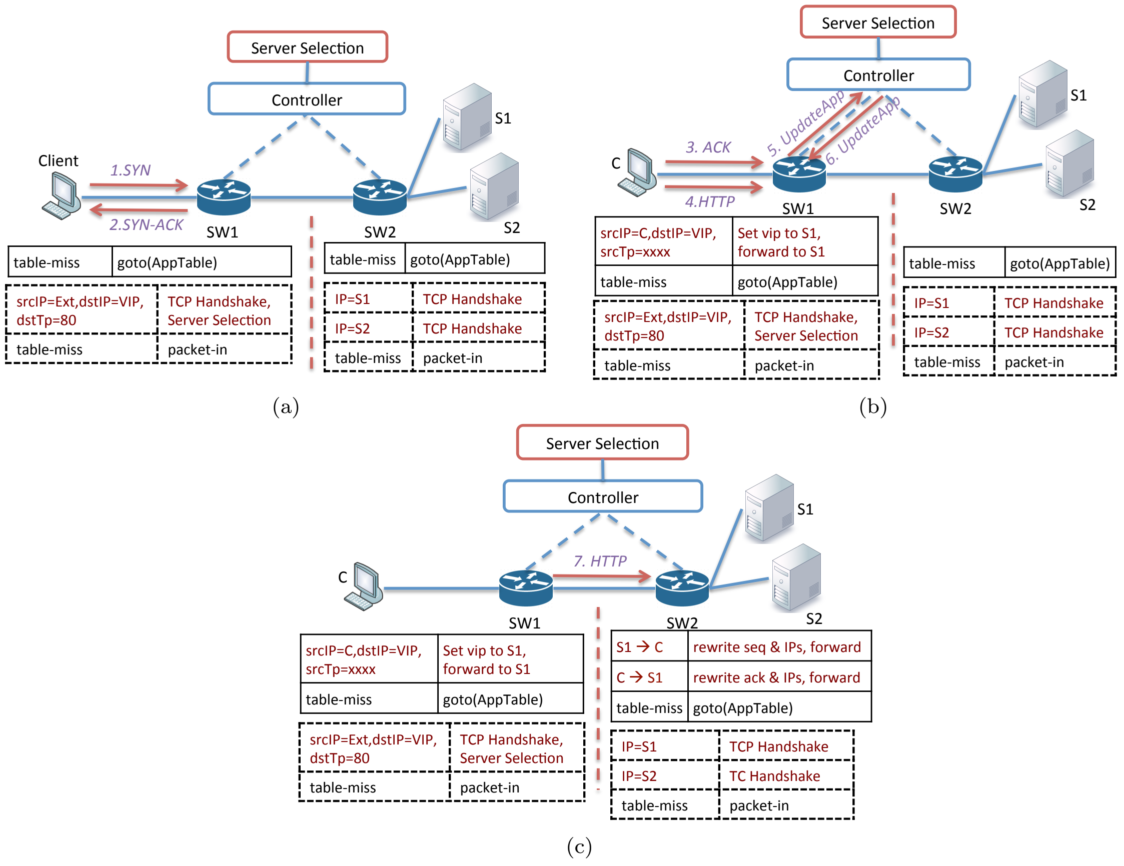
Server Selection
Controller
Client
1.SYN
2.SYN-ACK
SW1
SW2
S1
S2
table-miss | goto(AppTable)
srcIP=Ext,dstIP=VIP, dstTp=80 | TCP Handshake, Server Selection
table-miss | packet-in
IP=S1 | TCP Handshake
IP=S2 | TCP Handshake
(a)
C
3. ACK
4.HTTP
5. UpdateApp
6. UpdateApp
srcIP=C,dstIP=VIP, srcTp=xxxx | Set vip to S1, forward to S1
(b)
7. HTTP
S1 → C | rewrite seq & IPs, forward
C → S1 | rewrite ack & IPs, forward
table-miss | goto(AppTable)
IP=S2 | TC Handshake
(c)

Figure 2: **Content-aware server selection.** *(a) App-rule installation and TCP handshake with client, (b) Flow-rule insertion at SW1, (c) TCP handshake with S1 and flow-rule insertion at SW2*

3.2 Experimental Results

First, we verify that modifying the TCP seq/ack numbers by a delta does not incur any more overhead than normal OpenFlow actions such as setting the TCP source port. The OVS in this setup receives packets on a port, applies the pre-installed rules, and then forwards the packets to the output port. We use 10 Gbps cards to saturate the OVS datapath. We measure the average packet delay under sustained traffic rate and show the results in Figure 3. As shown in the figure, the two new data plane actions ("seq rewrite" and "ack rewrite") incur the same overhead as standard OpenFlow actions, which establishes that adding TCP sequence number translation is not affecting the datapath speed.

In the next experiment, we evaluate the overall data plane performance by comparing webpage download time perceived by the client for two different configurations. The first configuration is the standard OpenFlow operation where both SW1 and SW2 have all the rules pre-installed in the standard flow table without data plane apps enabled. This is used as a baseline for comparison. The second configuration uses the application-aware data plane, where the TCP Handshake and Server Selection functions are executed at the switches (refer to Figure 2). We use *httperf* [9] for the evaluation. As shown in Figure 4, we find that when the client request arrival rate is relatively small, overhead due to application-awareness is very low even with additional rule processing at the user space level by the OVS. As request arrival rate increases, we observe the divergence from the baseline scenario. This is because as request rate increases, more packets get queued for user space processing, leading to longer page download time. Note that our current code is not yet fully optimized. And more importantly, the OVS v1.10.0 that we currently use is single-threaded at the user space level. Hence we believe there is significant potential for throughput improvement in our implementation. We are presently optimizing the code and migrating it to the latest Open vSwitch version that supports multi-threading at the user space level, which will improve the performance.

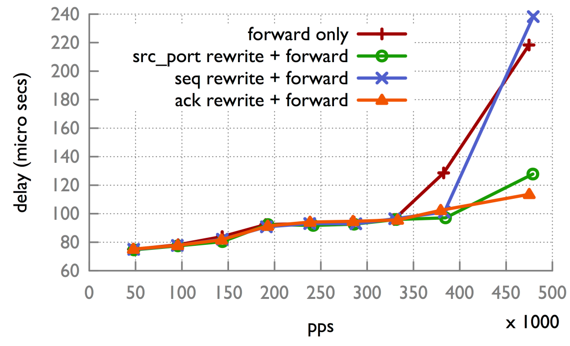

Figure 3: Data plane Performance for TCP splicing

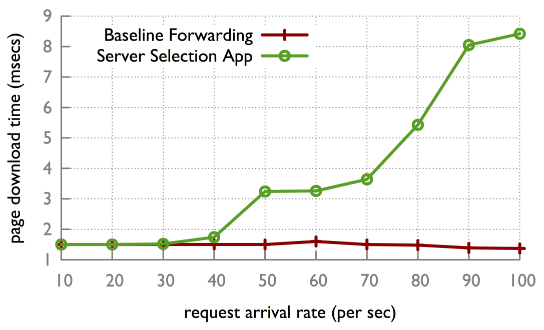

Figure 4: Data plane performance for page downloads

4. RELATED WORK

Network services have traditionally been implemented by steering flows through a chain of middleboxes. SDN enables dynamic middlebox chaining, although tracking the flows that traverse through the middleboxes remains a challenge because the packets may be altered along the way. Novel approaches have been proposed to address this problem by using either statistical inferencing [11] or explicit tags in packet headers [2]. However, inferencing may cause errors and finding unused fields in the packet headers for the tag may not always be possible. In our approach, service chaining is implemented by chaining apps on the same switch, and hence avoiding the need for explicitly tracking flows.

Middlebox virtualization has also been studied extensively [1, 4, 12]. For example, ETTM proposes to implement middlebox functions in special end-host modules [1]. Unlike previous approaches, we propose to integrate service functions into the SDN data plane for a unified network, service control, and ease of management.

Prior work has also been done to use SDN for traffic engineering [6, 7], layer-3 [5, 14] and layer-4 load balancing [10]. Our work builds on existing work to address application awareness in the SDN data plane, and addresses layer-7 load balancing problems.

Recently, AVANT-GUARD has been proposed to protect SDN networks against SYN floods by implementing TCP splicing in SDN switches [13]. Our focus is on a general framework for different types of applications, and can address security applications as well.

5. DISCUSSION AND CONCLUSIONS

This paper is only a first step towards enabling application-aware SDN data plane. There are still many interesting issues to be addressed. First, the architecture should allow data plane apps to be added dynamically without recompiling the entire OVS code or stopping its execution. We are currently experimenting what dynamic loading mechanism to use and what API to provide. Second, we need to explore the potential of this mechanism by investigating more network service applications. Although the current API seems to be able to address common app requirements, it remains to be validated or improved through more use cases. Third, the recent DPDK based OVS implementation opens up new possibilities. Certain applications that require per-packet payload scan (e.g. IDS/IPS) may not fit in our current design since they would invoke app actions at the user space for almost every packet and hence eliminate the performance benefit of the kernel flow table. But this may not be a problem under DPDK since all operations are done at the operating system user space.

6. ACKNOWLEDGMENTS

The authors thank anonymous reviewers for comments that helped improve the paper. This research was supported in part by DTRA grant HDTRA1-09-1-0050, NSF grants CNS-10171647 and CNS-1117536, and a CNS-1346688/Raytheon Subcontract.

7. REFERENCES

[1] C. Dixon, H. Uppal, V. Brajkovic, D. Brandon, T. Anderson, and A. Krishnamurthy. ETTM: A Scalable Fault Tolerant Network Manager. In *Proc. of NSDI*, 2011.

[2] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul. Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *Proc. of NSDI*, 2014.

[3] O. N. Foundation. OpenFlow Switch Specification (Version 1.3.0). June 2012.

[4] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella. Toward Software-defined Middlebox Networking. In *Proc. of HotNets-XI*, 2012.

[5] N. Handigol, S. Seetharaman, M. Flajslik, A. Gember, N. McKeown, G. Parulkar, A. Akella, N. Feamster, R. Clark, A. Krishnamurthy, et al. Aster* x: Load-Balancing Web Traffic over Wide-Area Networks, 2009.

[6] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-driven WAN. In *Proc. of SIGCOMM*, 2013.

[7] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *Proc. of SIGCOMM*, 2013.

[8] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network Virtualization in Multi-tenant Datacenters. In *Porc. of NSDI*, 2014.

[9] D. Mosberger and T. Jin. httperf – A Tool for Measuring Web Server Performan. In *Proc. of the Internet Server Performance Workshop*, 1998.

[10] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud Scale Load Balancing. In *Proc. of SIGCOMM*, 2013.

[11] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying Middlebox Policy Enforcement Using SDN. In *Proc. of SIGCOMM*, 2013.

[12] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of NSDI*, 2012.

[13] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. AVANT-GUARD: Scalable and Vigilant Switch Flow Management in Software-defined Networks. In *Proc. of CCS*, 2013.

[14] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Proc. of USENIX HotICE*, 2011.