Scotch: Elastically Scaling up SDN Control-Plane using vSwitch based Overlay

An Wang
George Mason University
awang10@gmu.edu

Yang Guo
Bell Labs, Alcatel-Lucent
Yang.Guo@alcatel-lucent.com

Fang Hao
Bell Labs, Alcatel-Lucent
Fang.Hao@alcatel-lucent.com

T. V. Lakshman
Bell Labs, Alcatel-Lucent
T.V.Lakshman@alcatel-lucent.com

Songqing Chen
George Mason University
sqchen@cs.gmu.edu

Abstract

Software Defined Networks use logically centralized control due to its benefits in maintaining a global network view and in simplifying programmability. However, the use of centralized controllers can affect network performance if the control path between the switches and their associated controllers becomes a bottleneck. We find from measurements that the software control agents on some of the switches have very limited throughput. This can cause performance degradation if the switch has to handle a high traffic load, as for instance due to flash crowds or DDoS attacks. This degradation can occur even when the data plane capacity is under-utilized. The goal of our paper is to design new mechanisms to enable the network to scale up its ability to handle high control traffic loads. For this purpose, we design, implement, and experimentally evaluate *Scotch*, a solution that elastically scales up the control plane capacity by using a vSwitch based overlay. *Scotch* takes advantage of both the high control plane capacity of a large number of vSwitches and the high data plane capacity of commodity physical switches to increase the SDN network scalability and resiliency under normal (e.g., flash crowds) or abnormal (e.g., DDoS attacks) traffic surge.

Categories and Subject Descriptors

C.2.1 [**Computer Communication Networks**]: Network Architecture and Design

General Terms

Design

Keywords

Network Architecture, SDN, overlay

1. INTRODUCTION

Software Defined Networking has recently emerged as a new networking paradigm of much research and commercial interest. A key aspect of Software Defined Networks (SDN) is the separation of control and forwarding [20]. The control plane is logically centralized and implemented on one or more controllers [13, 18, 3]. The centralized controller configures the flow tables in the various switches that it controls using a de facto standard protocol such as OpenFlow [12]. Each OpenFlow capable switch has a software implemented OpenFlow Agent (OFA) that communicates with the switch's controller over a secure TCP connection. This connection is used by the switch to inform the controller the arrival of new flows, and by the controller to configure the switch's flow table in both reactive (on-demand) and proactive (a priori configured) modes. The reactive mode, which permits fine-grained control of flows, is invoked when a new flow starts and there is no entry in the flow table corresponding to that flow. In this paper, we call the interconnection between a switch and its controller this switch's *control path* or *control channel*.

Since the controller and the switches it controls are separated, it is important that swithes' control paths are not bottlenecks between the switches and their controller. This is particularly important if the switch is configured to operate with a large fraction of reactive flows since each reactive flow requires communication with the controller. If a switch's control path is completely saturated, the switch becomes essentially disconnected from the controller and unable to change its flow tables in response to network conditions or new flow arrivals. The switch may then become unable to handle new flows even though the data plane is uncongested and could have forwarded the flow's packets. A low throughput of a control path can also lead to new Denial-of-Service (DoS) attacks. Malicious users may attempt to saturate the switch to controller channels and stop network operations by effectively making the controllers not reachable.

The load on the control path can be reduced by limiting reactive flows and pre-installing rules for all expected traffic. However, this comes at the expense of fine-grained policy control, visibility, and flexibility in traffic-management, as evidently required in [3, 16]. Alternatively, the control path capacity can be increased by optimizing the OFA implementation. However this is not sufficient to bridge the large gap between control plane and data plane throughputs. A switch's data plane forwarding capacity is typically several orders of magnitude larger than that of its control path. Ideally, one would like to elastically scale control plane capacity. Mechanisms for elastically scaling controller capacity have been proposed [18, 7]. However, they only scale the actual processing capacity of the controller but not the switch-controller control path capacity.

In this paper, we aim to develop new mechanisms that exploit the available high data plane capacities to elastically scale up the achievable throughput of control paths. We propose *Scotch*, an Open vSwitch based overlay that avoids the OFA bottleneck by using the data plane to scale the controller channel capacity. This scaling permits the network to handle much higher reactive flow loads, makes the control plane far more resilient to DDoS attacks by providing a mitigating mechanism, and permits faster failure recovery.

Scotch essentially bridges the gap between control and data plane capacities by building an overlay from each switch to the Open vSwitches that run on host hypervisors (assuming a virtualized environment). The idea is to pool the vSwitch-controller capacities of all the vSwitches and use it to expand the total capacity of control paths. Two significant factors that we exploit are (i) the number of vSwitches greatly exceeds the number of physical switches, and (ii) the control paths of vSwitches running on powerful CPUs with sufficient memory possess higher throughput than that of physical switches whose OFA runs on less powerful CPUs with less memory.

Scotch enables an elastic expansion of control channel capacity by tunnelling new flows to vSwitches over a control overlay. When the control path of a physical switch is overloaded, the default rule at the switch is modified so that new flows will be tunneled to multiple vSwitches, instead of being sent to the central controller through Packet-In messages. Note that when the new flows are tunneled to vSwitches there is no additional load on the OFA since the flows are handled by the switch hardware and stay in the data plane. This results in shifting load from the OFA to edge vSwitches. The vSwitches send the new flow's packets to the controller by using Packet-In messages. Since the vSwitch control agent has much higher throughput than the OFA on a hardware switch, and a large number of vSwitches can be used, the control plane bottleneck at the OFA can be effectively eliminated. Below we use DDoS attacks as an extreme example of traffic that can cause control path overload and study this in detail.

Scotch also uses the overlay to forward the data traffic, and separates the handling of small and large flows. Generally, it is not sufficient to address the performance bottlenecks at any one switch. When one switch is overloaded, it is likely that other switches are overloaded as well. This is particularly so if the overload is caused by an attempted DDoS attack that generates a large number of small flows in an attempt to overload the control plane. If an attacker spoofs packets from multiple sources to a single destination, then even if we spread the new flows arriving at the first hop hardware switch to multiple vSwitches, the switch close to the destination will still be overloaded since rules have to be inserted there for each new flow. To alleviate this problem, *Scotch* forwards new flows on the overlay so that new rules are initially only inserted at the vSwitches and not the hardware switches.

Most flows are likely to be small, and may terminate after a few packets are sent [1]. This is particularly true for flows from attempted DDoS attacks. *Scotch* can use monitoring information at the controller to migrate large flows back to paths that use physical switches. Since the majority of packets belong to a small number of large flows, this approach allows *Scotch* to effectively use the high control plane capacity on vSwitches and the high data plane capacity on hardware switches. The activation of *Scotch* overlay can also trigger the network security tools and solutions. The collected flow information can be fed into the security tools to help pinpoint the root cause of the overloading system. The security tools will hopefully kick in and tame the attacks. Once the control paths become uncongested, the *Scotch* overlay automatically phases out. The SDN network will gradually revert back to the normal working conditions.

The paper is organized as follows. Section 2 describes the related work. Section 3 investigates the performance of SDN networks in a DDoS attack scenario. We use DoS attacks and their mitigation as the extreme stress test for control plane resilience and performance under overload. Section 4 proposes the *Scotch* overlay scheme to scale up the SDN network's control-plane capacity, and Section 5 describes the design of key components of *Scotch*. The experimental results of *Scotch* performance are reported in Section 6. Concluding remarks are in Section 7.

2. RELATED WORK

Security issues in SDN have been studied and examined from two different aspects: to utilize the SDN's centralized control to provide novel security services, e.g., [21, 27], and to secure network itself, e.g., [10, 28]. The SDN security under DDoS attacks is studied in [28]. Two modules are added at the data-plane of the physical switches. One module functions similarly to a SYN proxy to avoid attacking traffic from reaching the controller. The other module enables the traffic statistics collection and active treatment, e.g., blocking harmful traffic. The motivation for our work is to elastically scaling up the control path capacity as the load on the control plane increases, due to either the flash crowds or DDoS attacks. No modification at the physical switches is required.

Centralized network controllers are key SDN components and various controllers have been developed, e.g., [13, 2, 18, 9], among others. In our experiment, we use the *Ryu* [5] OpenFlow controller. *Ryu* supports latest OpenFlow at the time of our experiments, and is the default controller used by Pica8 switch [23], the physical switch used in our experiment. Since Scotch increases the control path capacity, the controller will receive more packets. A single node multi-threaded controller can handle millions of PacketIn/sec [30]. A distributed controller, such as [7], can further scale up capacity. The design of a scalable controller is out of the scope of this paper.

Traffic detouring techniques have been employed to reduce the routing table size, e.g., [33, 17, 25, 26], where traffic is re-routed inside the physical networks. *Scotch* employs an overlay apporach for traffic detouring, which offers better flexibility and elasticity than the in-network traffic detouring approach. In addition, *Scotch* can also help reduce the number of routing entries in the physical switches by routing short flows over the overlay.

DevoFlow [6] has some similarity with our work in the sense that it identifies limited control-plane capacity as an important issue impacting SDN performance. The proposed DevoFlow mechanism maintains a reasonable amount of flow visibility that does not overload the control path.

The authors of [15] investigated the SDN emulation accuracy problem. Their expriments also reveal SDN switches' slow control-path problem, which is consistent with our findings using a different type of physical switch. However, the goal of their work is quite diffent from ours. While they attempt to improve the emulation accuracy, we develop a new scheme to improve SDN's control-path capacity.

3. OPENFLOW CONTROL PLANE BOTTLENECK

3.1 Background

Fig. 1 shows a basic OpenFlow network where all the switches are connected to a central controller via secure TCP connections. Each switch consists of both a data plane and a simple control plane – the OpenFlow Agent (OFA). The data plane hardware is responsible for packet processing and forwarding, while the OFA allows the central controller to interact with the switch to control its behavior. Each switch has one or more flow tables (not shown in the figure), which store rules that determine how each flow should be processed.

When the first packet of a new flow arrives at a switch, the switch looks up the flow table to determine how to process the packet. If the packet does not match any existing rule, it is treated as the first packet of a new flow and is passed to the switch's OFA. The OFA encapsulates the packet into a *Packet-In* message and delivers the message to the central controller (Step 1 in Fig. 1). The Packet-In message contains either the packet header or the entire packet, depending on the configuration, along with other information such as ingress port id, etc. Upon receiving the Packet-In message, the OpenFlow controller determines how the flow should be handled based on policy settings and the global network state. If the flow is admitted, the controller computes the flow path and installs new flow entries at the corresponding switches along the path (Step 2). This is done by sending a flow modification command to the OFA on each switch along the route. The OFA then installs the new forwarding rule into the flow table. The controller may also generate a *Packet-Out* message to explicitly tell the switch where to forward the first packet.

One problem with the current OpenFlow switch implementation is that the OFA typically runs on a low end CPU that has limited processing power. This seems to be a reasonable design choice since one intention of the OpenFlow architecture is to move the control functions out of the switches so that the switches can be simple and of low cost. However, this can significantly limit the control path throughput.

To better understand this limitation, we study a DDoS attack scenario, where a DDoS attacker generates SYN attack packets using spoofed source IP addresses. The switch treats each spoofed packet as a new flow and forwards the packet to the controller. The insufficient processing power of the OFA limits how fast the OFA can forward the packets to the OpenFlow controller, as well as how fast it can insert new flow rules into the flow table. A DDoS attack can cause Packet-In messages to be generated at a much higher rate than what the OFA can handle. This effectively makes the controller unreachable from the switch and causes legitimate traffic to be blocked even though there is no data plane congestion. Note that this blocking of legitimate traffic can occur whenever the control plane is overloaded, e.g., under DDoS attacks or due to flash crowds. We merely use the DDoS attack scenario as an extreme traffic pattern that causes control plane overload.

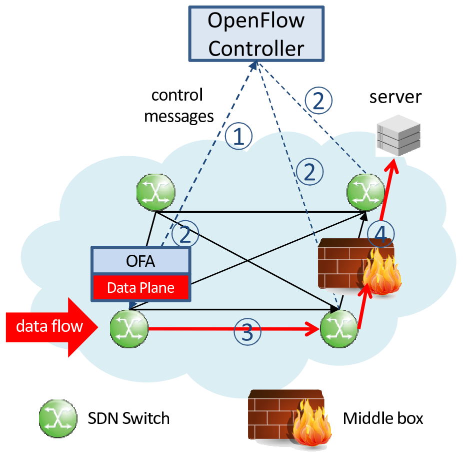


Figure 1: OpenFlow network with the OpenFlow Controller.

3.2 Attack's impact on SDN switch performance

Here, we experimentally evaluate the impact of control plane overload on SDN packet forwarding performance. The overload is caused by an attempted DDoS attack. The testbed setup is shown in Fig. 2. We experiment with two types of hardware switches: Pica8 Pronto 3780 and HP Procurve 6600, with OpenFlow 1.2 and 1.0 support, respectively. For comparison, we also experiment with Open vSwitch, which runs on a host with an Intel Xeon 5650 2.67GHz CPU. We use the *Ryu* OpenFlow controller, since it was one of the few controllers that supported OpenFlow 1.2 at the time of our experiments, a requirement by Pica8 Pronto switch. The experiments are done using one switch at a time. The attacker, the client and the server are all attached to the data ports, and the controller is attached to the management port. We use *hping3* [14] to generate attacking traffic. The Pica8 switch uses 10 Gbps data ports, and the HP switch and vSwitch have 1 Gbps data ports. The management ports for physical switches are 1Gbps.

Both the attacker and the legitimate client attempt to initiate new flows to the server. We simulate the new flows by spoofing each packet's source IP address. Since the OpenFlow controller installs the flow rules at the switch using both the source and destination IP addresses, a spoofed packet is treated as a new flow by the switch. Hence in our experiment, the flow rate, i.e., the number of new flows per second, is equivalent to the packet rate. We set the client's new flow rate at 100 flows/sec, while vary the attacker's attacking rate from 10 and 3800 flows/sec. We collect the network traffic using tcpdump at the client, the attacker, and

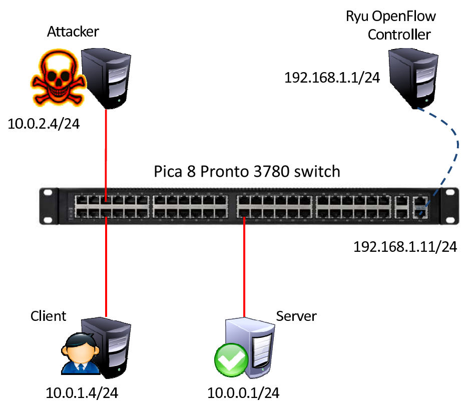


Figure 2: DDoS attack experiment setup.

the server. We define the *client flow failure fraction* to be the fraction of client flows that are not able to pass through the switch and reach the server. The client flow failure fraction is computed using the collected network traces. Fig. 3 plots the client flow failure fraction for different switches as the attacking flow rate increases. We observe that all three switches suffer from the client flow failure as the attacking flow rate increases. Note that even at the peak attacking rate of 3800 flows/sec, and even with the maximum packet size of 1.5 Kbytes, the traffic rate is merely 45.6 Mbps, a small fraction of the data link bandwidth. This indicates that the bottleneck is at the control plane rather than at the data plane. In addition, both Pica8 and HP Procurve physical switches exhibit much higher flow failure fraction than the software-based Open vSwitch, suggesting software-based vSwitch has higher control path capacity than the two physical switches tested in this experiment.

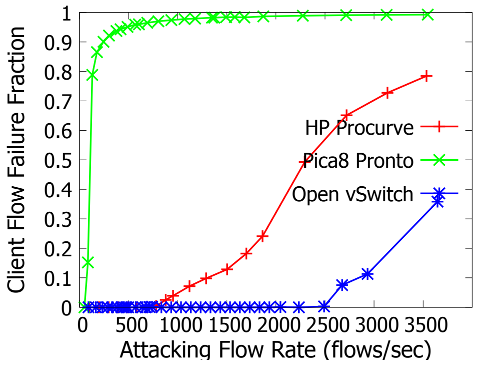


Figure 3: Physical switches and Open vSwitch control plane throughput comparison.

3.3 Profiling the control path bottleneck

We next identify which component along the control path is the actual bottleneck. Recall that a new flow arrival at the SDN switch triggers the following control plane actions: (1) a Packet-In message is sent from OFA to the controller; (2) the controller sends back a new rule to OFA; and (3) OFA inserts the rule into the flow table. The new flow can go through the switch if all the above three steps are completed successfully. Below we use the similar experimental set-up as in the previous experiment (see Fig. 2), with the client generating a new flow per packet towards the server while the attacker is turned off. The network traffic is traced at the server and the OpenFlow controller. We measure the Packet-In message rate (observed at the controller), the flow rule insertion rate (observed at the controller), and the rate at which the new flows successfully pass through the switch and reach the destination (observed at the server).

Fig. 4 plots the Packet-In message rate, flow rule insertion rate (one rule is included in one packet), and the received packet/flow rate at the server. We use the Pica8 switch for this experiment. We observe that all three rates are identical, which suggests that *the OFA's capability in generating Packet-In messages is the bottleneck*. Experiments in Section 6.1 further show that the rule insertion rate that the switch can support is indeed higher than the Packet-In message rate for Pica8 switch.

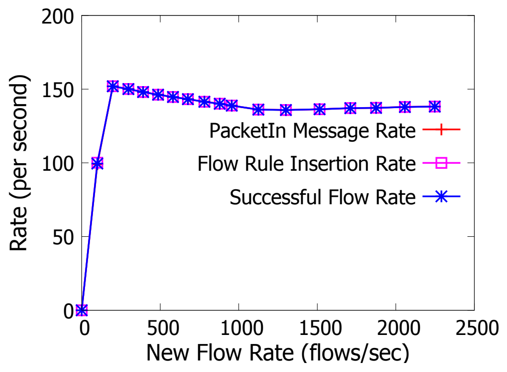


Figure 4: SDN switch control path profiling.

A limited amount of TCAM at a switch can also cause new flows being dropped [19]. A new flow rule won't be installed at the flow table if it becomes full. In the above experiment, OFA's capability in generating Packet-In messages is the bottleneck while the TCAM size is not. However, the solution proposed in this paper is applicable to the TCAM bottleneck scenario as well.

We use Pica8 Pronto switch instead of HP Procurve switch in this experiment and most of the later experiments. Though the Procurve switch has higher OFA throughput (see Fig. 3), we use the Pica8 Pronto switch due to its more advanced OpenFlow data-plane features that it supports, e.g., tunneling, multiple flow table support, etc. Also the Pronto switch can do wire speed packet processing with full OpenFlow support, while the older Procurve switch used in our experiments cannot. We do not intend to compare different hardware switches here, but just to explain the rational for our selection of equipments.

4. SCALING UP SDN CONTROL PLANE USING SCOTCH OVERLAY

We make the following observations based on the experiments as presented in Section 3:

- The control plane at the physical switches has limited capacity. The maximum rate at which the new flows can be set up at the switch is low – it is several orders of magnitude lower than the data plane traffic rate.

- The OpenFlow network running in reactive mode is very vulnerable to DDoS attacks. The operation of SDN switches can be easily disrupted by the increase of control traffic due to DDoS attacks, unless the switch operates very much in proactive mode.

- The vSwitches have higher control plane capacity but lower data plane throughput compared to the physical switches. The higher control plane capacity can be attributed to the more powerful CPUs on the general purpose computers where the vSwitches typically run on.

Our goal here is to elastically scale up the SDN control plane capacity when needed without sacrificing any of the advantages of SDN regarding the controller having high visibility and fine-grained control of all flows in the network. A straightforward method is to use more powerful CPUs for the OFA combined with other design improvements that allow faster access rates between the OFA and line cards. One can also improve the design and implementation of OFA software stack to enable more efficient message processing. With continued research interest in the SDN network control plane [6], these improvements will be likely. However, the significant gap between the control path throughput and data path flow setup requirements will still persist. For example, it may not be economically desirable to dimension the OFA capacity to be based on the maximum possible flow arrival rate given that the peak flow rate may be several orders of magnitude higher than the average flow rate [1]. Another method is to dedicate one port of the physical switch to the overloaded new flows. Whenever the control path is overloaded, the new flows are forwarded to the controller via this dedicated port at the data-plane. However, using a dedicated physical port does not fully solve the problem. The maximum flow rule insertion rate is limited as shown in Section 6.1. The controller cannot install the flow rules fast enough at physical switches when overloaded.

Software based virtual switches (e.g., Open vSwitch) have been widely adopted. vSwitches offer excellent switching speed [4, 8], and high control path throughputs as shown in the previous section. The interesting question is whether we can use vSwitches to improve the control plane capacity of physical switches. *Scotch*, our proposed solution, addresses this question and achieves high control plane throughputs using a vSwitch based overlay.

4.1 Scotch: vSwitch based SDN overlay network

Fig. 5 depicts the architecture of the *Scotch* overlay network. The main component of *Scotch* is a pool of vSwitches that are distributed across the corresponding SDN network, e.g., across different racks in the data center for a data center SDN network, or distributed at different locations for a wide-area SDN network. We select vSwitches at hosts that are lightly loaded and with under-utilized link capacity.

The *Scotch* overlay consists of three major components: (i) *vSwitch mesh* - a fully connected mesh (using tunnels) of vSwitches; (ii) the tunnels that connect the underlying physical switches with the vSwitch mesh; and (iii) the tunnels that connect the end hosts with the vSwitch mesh. The tunnels can be configured using any of the availablel tunneling protocols, such as GRE, MPLS, MAC-in-MAC, etc.; they use the underlying SDN network's data plane.

The tunnels connecting the physical switches with the vSwitch mesh allow a physical switch to forward the new flows to the vSwitches whenever the physical switch becomes overloaded in its control path. The vSwitches can then handle the new flow setup and packet forwarding tasks for these flows. The benefits are two-fold. First, the new flows can continue to be serviced by the SDN network in the face of control path congestion. Second, the SDN controller can continue to observe the new flows, which give us the opportunity to mitigate the possible DDoS attack (more in Section 5.2). The collected flow information can also be fed into network security applications to diagnose the root cause of the control path congestion. For the purpose of load-balancing, a physical switch is connected to a set of vSwitches so that the new flows can be distributed among them. Further details of load-balancing are described in Section 5.1.

The tunnels connecting the end hosts with the vSwitch mesh allow the new flows to be delivered to the end hosts over the *Scotch* overlay. Once a packet is forwarded from a physical switch to a *Scotch* vSwitch, it needs to be forwarded to the destination. This can be done if a path over the underlying SDN network can be set up on demand. But this may not be desirable since it may overload the control paths of physical switches on the path and create other hot spots. Another option is to configure tunnels between the *Scotch* vSwitch and the destinations directly. This, however, would lead to a larger number of tunnels since it requires one tunnel from each *Scotch* vSwitch to each host.

To avoid these problems, we partition hosts based on their locations so that all hosts are covered by one or more nearby *Scotch* vSwitches. For example, in the case of data center SDN network, there may be two *Scotch* vSwitches at each rack. Tunnels are set up to connect the host vSwitches with their local *Scotch* vSwitches.

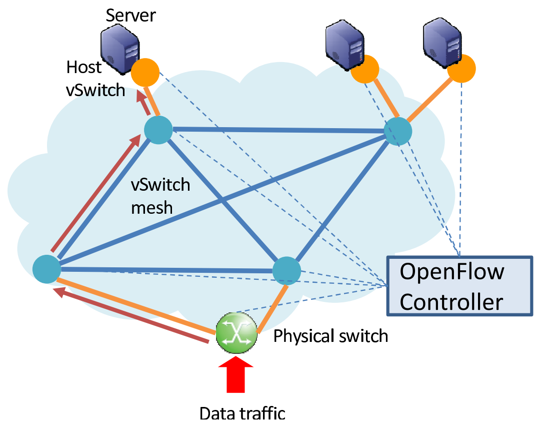


Figure 5: Architecture of Scotch overlay network.

Finally, we choose to form a fully connected vSwitch mesh in order to facilitate the overlay routing. The arrows in Fig. 5 show how the packets are forwarded through the *Scotch* overlay when the *Scotch* is on. Packets are first forwarded from the physical switch to a randomly chosen

Scotch vSwitch. Then they are routed across the mesh to the vSwitch that is closest to its destination. The receiving *Scotch* vSwitch further delivers the packets to the destination host via the tunnel. Finally, the host vSwitch delivers the packet to the destination VM. Since there is a full-mesh tunnel connectivity between *Scotch* vSwitches, a packet traverses three tunnels before reaching its destination.

Next, we use an example to describe how the *Scotch* overlay network scales up the control path throughput, and also illustrate how packets are forwarded if middleboxes are involved.

4.2 An example

Fig. 6 illustrates an example how the scheme works. The OpenFlow controller monitors the rate of Packet-In messages sent by the OFA of each physical switch to determine if the control path is congested. If a control path is deemed to be congested, the new flow packets arriving at the switch are forwarded to one or multiple *Scotch* vSwitches (only one is shown in Fig. 6) (Step 1 in Fig. 6). This allow these packets to leave the physical switch via the data plane instead of being handled by overloaded OFA and going through the congested control path. Details of overload forwarding rule insertion at the physical switch and load balancing across multiple vSwitches are discussed in Section 5.1.

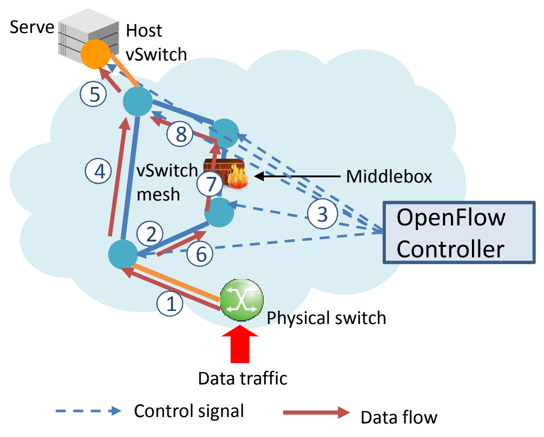


Figure 6: An example showing how Scotch works.

When the packet reaches the vSwitch, the vSwitch treats the packet as a new flow packet. The vSwitch OFA then constructs a Packet-In message and forwards it to the OpenFlow controller (Step 2 in Fig. 6). We configure the vSwitch to forward the entire packet to the controller, so that the controller can have more flexibility in deciding how to forward the packet. The controller can choose to set up the path either from the vSwitch or from the original physical switch. If the path is set up from the vSwitch, the *Scotch* overlay tunnels are used to forward the packet. The controller needs to set up the forwarding rules at the corresponding vSwitches, as done in Step 3 in Fig. 6.

The packets continue to be forwarded on the *Scotch* overlay until they reach the destination (Step 4 and 5 in Fig. 6). If middleboxes need to be passed through, e.g., a firewall needs to be passed through as shown in Fig. 6, the packets are routed through the firewall (Step 6, 7, 8, and 5). Details of maintaining policy consistency is described in Section 5.4.

Packets from the same flow follow the same overlay data path.

5. DESIGN DETAILS

5.1 Load balancing across vSwitches

In general, we want to balance the network traffic among different vSwitches in the overlay to avoid performance bottlenecks. Hence when multiple vSwitches are used to receive packets from a physical switch, we need a mechanism to do load balancing between these vSwitches. In the following, we describe a method that implements load balancing by using the *group table feature* offered in OpenFlow Switch Spec 1.3 [12].

A group table consists of multiple group entries, where each entry contains *group Id*, *group type*, *counters*, and *action buckets*. Group type defines the group semantics. Action buckets contain an ordered list of action buckets, where each action bucket contains a set of actions to be executed and their associated parameters.

To achieve load balancing, we use *select* group type, which chooses one bucket in the action buckets to be executed. The bucket selection algorithm is not defined in the spec and the decision is left to the switch vendors/users. Given that ECMP load balancing is well accepted for router implementations, it is conceivable that using a hash function based on the flow id may be a likely choice for many vendors. We define one action bucket for each tunnel that connects the physical switch with a vSwitch (see Fig. 5). The *action* of this bucket is to forward the packet to the corresponding vSwitch using the pre-set tunnel.

5.2 Flow management at OpenFlow controller

Identifying the flows at the controller. In order to manage the flows at the central controller, we first need to make sure that the Packet-In message arriving at the controller from the *Scotch* vSwitch carries the same information as that coming directly from the physical switches. This is mostly true since the Packet-In messages contain similar information in both cases. But there are two exceptions. First, when the packet comes from a vSwitch, it does not contain the original physical switch id. This can be easily addressed by maintaining a table to map the tunnel id to the physical switch id, so that the controller can infer the physical switch id based on the tunnel id contained in the Packet-In meta data. Second, the packet from vSwitch also does not contain the original ingress port id at the physical switch. We propose to use a second label to solve this problem. In the case of MPLS, an inner MPLS label is pushed into the packet header based on the ingress port. In the case of GRE, the GRE key is set according to the ingress port. Note that since the packets need to be load balanced across different vSwitches, two flow tables are needed at the physical switch: the first table contains the rule for setting the ingress port; and the second table contains the rule for load balancing. The vSwitch strips off the inner MPLS label or the GRE key, attaches the information on the Packet-In message, and send them to the controller. The controller maintains the flow's first-hop physical switch id and the ingress port id at the *Flow Info Database*. Such information will be used for large flow migration as described in Section 5.3.

Flow grouping and differentiation. Next we describe how the OpenFlow controller manages the new flows. When a new flow arrives, the controller has three choices: (1) forwarding the flow over the physical SDN network, starting from the first physical switch encountered by the flow; (2) forwarding the flow using the vSwitch overlay network, starting from the vSwitch which forwards the first packet using the Packet-In message; and (3) dropping the flow when the load is too high, especially if the flow is identified as a DDoS attack flow. In general, we can classify the flows into different groups and enforce fair sharing of the SDN network across groups. For example, we can group the flows according to which customer it belongs to, so that we can achieve fair sharing among different customers. In the following, we give an example of providing fair access to the SDN network for the flows arriving from different ingress ports of the same switch. This is motivated by the observation that if a DDoS attack comes from one or a few ports, we can limit its impact to those ports only.

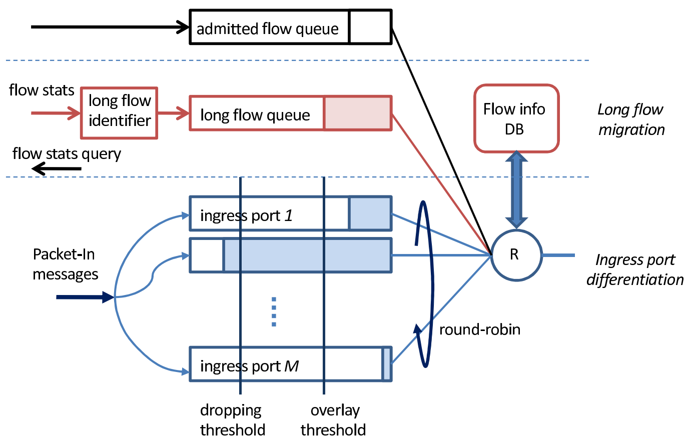


Figure 7: *Scotch* Flow management at the OpenFlow controller for one SDN switch: ingress-port differentiation and large flow migration from the *Scotch* overlay to the SDN network.

For the new flows from the same physical switch, the OpenFlow controller maintains one queue per ingress port (see *Ingress port differentiation* at the lower part of Fig. 7). The service rate for the queue is R, the maximum rate at which the OpenFlow controller can install rules at the physical switch without insertion failure or packet loss in the data plane. We will investigate how to choose the proper value of R in Section 6. The controller serves the different queues in a round-robin fashion so as to share the available service rate evenly among ingress ports. If a queue size grows to be larger than the *overlay threshold*, we assume that the new flows at this queue are beyond the control plane capacity on the physical switch network. Hence the controller will route the flows surpassing the threshold over the *Scotch* overlay by installing forwarding rules at corresponding vSwitches. If a queue size continues to build up, and exceeds the *dropping threshold*, then neither the physical network nor the *Scotch* overlay is able to carry these flows. The Packet-In messages beyond the *dropping threshold* will simply be dropped from the queue.

Note that the focus of this paper is to provide a mechanism to mitigate the impact of SDN control path congestion, which may be caused by flash crowds or DDoS attacks. Although our scheme offers high visibility to new flows and the opportunity and mechanism to monitor and handle flows, we do not address DDoS attack detection and diagnosis problems. Existing network security tools or solutions can be readily integrated into our framework, e.g., as a new application at the SDN controller, to take advantage of the visibility and flexibility offered by *Scotch*.

5.3 Migrating large flows out of the overlay network

Although vSwitch overlay can scale up the control path capacity, it is not desirable to only forward flows by using vSwitches since the vSwitch data plane has a much lower throughput than that of physical switches. In addition, the forwarding path on the overlay network is longer than the path on the physical network. In this section, we discuss how to take advantage of the high data plane capacity of the underlying physical network.

Measurement studies have shown that the majority of link capacity is consumed by a small fraction of large flows [1]. Hence our idea is to identify the large flows in the network and migrate the large flows out of the Scotch overlay. Since there are few large flows in the network, such migration should not incur major control plane overhead.

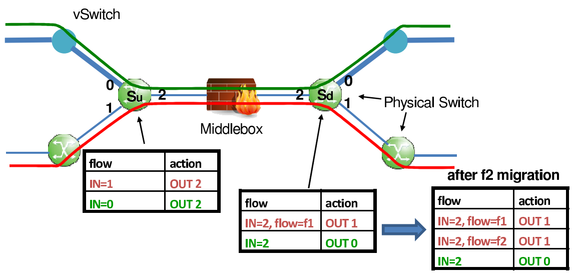
The middle part of Fig. 7 illustrates the operations that the controller conducts for large flow migration. The controller sends the flow-stats query messages to the vSwitches, and collects the flow stats including packet counts. The *large flow identifier* selects the flows with high packet counts, and puts the large flow migration requests into the large flow migration queue. The controller then queries the *Flow Info Database* to look up the flow's first hop physical switch. The controller then computes the path and checks the message rate of all switches on the path to make sure their control plane is not overloaded. It then sets up the path from the physical switch to the destination. This is done by inserting the flow forwarding rules into the *admitted flow queue* of the corresponding switches (top part of Fig.. 7). The rules will be installed on the switches when the inserted rules are pulled out of the queue by the controller. Once the fowarding rules are installed along the path, the flow will be moved to the new path, and remain at the physical SDN network for the rest of time. Note that the forwarding rule on the first hop switch is added at last so that packets are forwarded on the new path only after all switches on the path are ready.

The OpenFlow controller gives the highest priority to the *admitted flow queue*, followed by the *large flow queue*. Ingress-port differentiation queues receive the lowest priority. Such a priority order causes small flows to be forwarded on physical paths only after all large flows are accommodated.

5.4 Maintaining policy consistency

When we migrate a flow from *Scotch* overlay to the underlying physical network, we need to make sure that both routing paths satisfy the same policy constraints. The most common policy constraints are middlebox traversal, where the flow has to be routed across a sequence of middleboxes according to a specific order.

A naive approach is to compute the new path of physical switches without considering the existing vSwitch path. For example, if a flow is routed first through a firewall FW_1 and then a load balancer LB_1 on the vSwitch paths, we may compute a new path that uses a different set of firewall FW_2 and load balancer LB_2. This approach in general does not work since the middleboxes often maintain flow states. When a flow is routed to a new middlebox in the middle of the connection, the new middlebox may either reject the flow or handle the flow differently due to lack of pre-established context. Although it is possible to transfer flow states between old and new middleboxes, this requires middlebox specific changes and may lead to significant development cost and performance penalty.



flow	action
IN=1	OUT 2
IN=0	OUT 2

flow	action
IN=2, flow=f1	OUT 1
IN=2	OUT 0

flow	action
IN=2, flow=f1	OUT 1
IN=2, flow=f2	OUT 1
IN=2	OUT 0

Figure 8: Maintain policy consistency in *Scotch*.

In order to avoid the middlebox state synchronization problem, our design enforces the flow to go through the same set of middleboxes in both the vSwitch and physical switch paths. Figure 8 illustrates how this is done. In this example we assume a typical configuration where a pair of physical switches, S_U and S_D, are connected to the input and output of the middlebox (firewall), respectively. But the solution also works for other configurations, as we will discuss at the end of this section.

The green line on the top shows the overlay path. The vSwitches in the overlay mesh connect to the physical switches S_U and S_D with tunnels. In the case that the physical switches cannot support tunnels to all vSwitches in the mesh, a few dedicated vSwitches in the mesh that are close to the middleboxes can serve as dedicated tunnel aggregation points. The upstream physical switch, S_U, decapsulates the tuneled packet before forwarding the packet to the middlebox to ensure that the middlebox sees the original packet without the tunnel header. Similarly, the downstream physical switch, S_D, encapsulates the packet again so that the packet can be forwarded on the tunnel. The flow rules at physical switches S_U and S_D (green entries in the flow tables) enforce the flows on the overlay path to go through the firewall and stay on the overlay.

The red line at the bottom shows the path for flows that are not routed on the overlay. They can be either elephant flows that are selected to migrate, or the flows that are set up while the physical switches have sufficient control plane capacity. The flow rules for forwarding such flows are shown in red in the flow tables. The red (physical path) rules have higher priority than the green (overlay) rules. Each individual flow forwarded on the physical path requires a red rule, while all flows on the overlay path share the same green rule. In other words, flows without individual rule entries are forwarded on the overlay path by default. This is important for scalability: when the control plane is overloaded, all flows can be forwarded on the overlay path without incurring per-flow setup overhead on the physical switches.

Fig. 8 shows an example of how flows are forwarded. Initially, flow f_1 is routed over the underlying physical network while other flows are routed over the *Scotch* overlay. When an elephant flow, say f_2, needs to be migrated to the physical path at the bottom, the controller adds an additional forwarding rule to make S_D forward flow f_2 to the physical network.

We next examine the impact of different middlebox connection types. In the data center network, sometimes the middleboxes are "attached" to a physical switch. This happens, for instance, when the middlebox is integrated with the physical switch or router. This is essentially combining the S_U and S_D in Figure 8. Since the rules on both switches are independent of each other, we can simply combine the rules on S_U and S_D and install them on the "attaching" switch. Virtual middleboxes that run on Virtual Machines may also be combined. In this case a vSwitch can run on the hypervisor of the middlebox host and execute the functions of S_U and S_D.

5.5 Withdrawal from Scotch overlay

As the DDoS attack stops or the flash crowd goes away, the switch control path becomes uncongested and hence the *Scotch* overlay becomes unnecessary. We then stop forwarding new flows to the overlay at the uncongested switch, while keeping existing flows uninterrupted.

The controller detects such control plane condition change by monitoring the new flow arrival rate at physical switches. If the arrival rate falls below a threshold, the OpenFlow controller starts the *withdrawal process*. The withdrawal process consists of three steps. First, for the flows that are currently being routed over the *Scotch* overlay, the controller inserts rules at the switch to continuously forward these flows to the *Scotch* overlay. Since the large flows should have been already migrated to the physical network, most of these flows are likely to be small flows and may terminate shortly. Second, the controller removes the default flow forwarding rule that was inserted initially when *Scotch* was activated (Section 5.1). The new flow packets will be forwarded to the OpenFlow controller directly via OFA. Third, if any remaining small flows routed on the overlay become large flows, they can still be migrated to the physical path following the same migration procedure.

Note that the *Scotch* overlay is for the entire network, so other congested switches may continue to use *Scotch* overlay.

5.6 Configuration and maintenance

To configure the *Scotch* overlay, we first need to select host vSwitches based on the planned control path capacity, physical network topology, host and middlebox locations, and so on. Extra backup vSwitches are added to provide necessary fault tolerance. We then configure the Scotch overlay by setting up tunnels between various entities: between physical switches and vSwitches for load distribution, between each pair of mesh vSwitches for forwarding, and between mesh and host vSwitches for delivery. vSwitch offers reasonably high data throughput [4]. Recent advancements in packet processing at general purpose computers, such as the systems based on the Intel DPDK library, can further boost the vSwitch forwarding speed [8] significantly. In addition, vSwitch has low overhead to support software tunneling. According to [31], it is possible to do tunneling in software with performance and overhead comparable to non encapsulated traffic, and to support hundreds of thousands of tunnel end points. In terms of configuration overhead, although the

Scotch overlay can be large, configuration is done largely offline so it should not affect operation efficiency.

A vSwitch may fail or stop functioning properly. Hence we need to detect such failures in order to avoid service interruption. vSwitch has a build-in heartbeat module that periodically sends the ECHO REQUEST message to the OpenFlow controller, which responds with the ECHO RESPONSE message. The heartbeat period can be adjusted by changing configuration parameter. The heartbeat message enables the OpenFlow controller to detect the failure of a vSwitch. In fact, several OpenFlow controllers, e.g. Floodlight [11], already include the vSwtich failure detection module. Once a controller detects the failure, the controller can replace the failed vSwitch with the backup in the action buckets installed in the physical switch as described in Section 5.1. The flows that are originally routed through the failed vSwitch will then be handled by the backup vSwitch, which treats the affected flows as new flows. When recovered, the failed vSwitch can join back *Scotch* as a new or backup vSwitch.

We may also need to add new vSwitches to increase the *Scotch* overlay capacity or replace the departed vSwitches. A new vSwitch becomes part of the overlay after it is connected with other vSwitches or physical switches, depending on its role, and is registered with the *Scotch* overlay controller. We do not expect frequent vSwitch additions or failures.

6. EVALUATION

We implement the *Scotch* overlay management as an application on the *Ryu* OpenFlow controller. We also construct a basic Scotch overlay with multiple vSwitches, and form an overlay together with end-hosts and physical switches using MPLS tunnels. Note that attackers, servers, clients, and vSwitches can be anywhere as long as a tunnel can be set up between the physical switch and them. We use experiments to demonstrate the benefits of ingress port differentiation and large flow migration. We also show the growth in the *Scotch* overlay's capacity with addition of new vSwitches into the overlay. We further investigate the extra delay incurred by the Scotch overlay traffic relay. Finally, we conduct the trace driven experiment that demonstrates the benefits of *Scotch* to the application performance in a realistic network environment.

6.1 Maximum flow rule insertion rate

As shown in Fig. 7, the maximum rate at which the OpenFlow controller installs the new rules into the switch, R, is an important design parameter. The larger the R, the better, so that more traffic can be routed over the physical network. However, the value of R needs to be set properly so that all the new flow rule insertion requests can be successfully handled at the switch. We first measure the maximum flow rule insertion rate allowed by the Pica8 switch. We let the *Ryu* controller generate flow rules at a constant rate and send them to the Pica8 switch. The switch OFA installs the rules into the flow table. The generated rules are all different, and the time-out period of a rule is set to be 10 seconds. Throughout the experiment, there is no data traffic passing through the switch.

The *Ryu* controller periodically queries the switch to get the number of rules currently installed in the flow table. We set the query interval sufficiently long (30 seconds), to minimize the impact on the OFA's rule insertion performance. Denote by N_k the number of rules in the flow table at the k-th query, and K the total number of queries. Denote by T the rule time-out period. The successful insertion rates can be estimated as $\sum N_k/(K \cdot T)$.

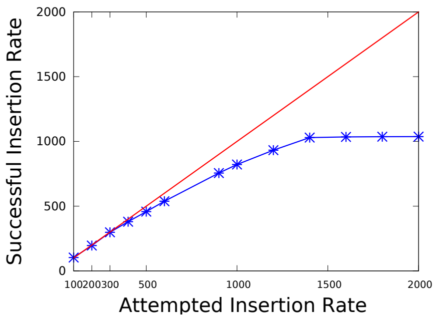


Figure 9: Maximum flow rule insertion rate at the Pica8 switch.

Fig. 9 plots the successful flow rule insertion rate with varying attempted insertion rate. The Pica8 switch is able to handle up to 200 rules/second without loss. After that, some rule requests are not installed into the flow table, and the successful insertion rate flattens out at about 1000 rules/second. In *Scotch*, the OpenFlow controller should only insert the flow rules at a rate that does not cause installation failure.

6.2 Interaction of switch data plane and control path

During the maximum flow rule insertion rate experiment, the Pica8 switch does not route any data traffic. In reality, while the OFA writes the flow rules into the flow table, the switch also does flow table lookups to route incoming packets. These two activities interact with each other. We conduct an experiment where the OpenFlow controller attempts to insert the flow rule at certain rates while the switch routes the data traffic with rates of 500, 1000, and 2000 packets/second. We measure the data-plane packet loss rate at the receiving host.

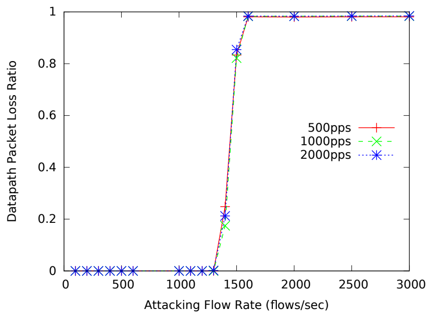


Figure 10: Interaction of the data path and the control path at the Pica8 switch.

Fig. 10 depicts the packet loss ratio with varying flow rule insertion rates. The curve exhibits an interesting *turning point* at a rule insertion rate of 1300 rules/second. The data path loss rate exceeds 90% when the rule insertion rate is greater than 1300 packets/second. This clearly demonstrates the interaction between the data and control-paths. We conjecture that the interaction is caused by the contention for the flow table access, which is confirmed by the vendor [22].

The results from this experiment and the previous experiment help us with setting the right new flow insertion rate at the OpenFlow controller. For Pica8 switch, the flow rule insertion rate is lower. Thus it governs the new flow insertation rate at the controller. Fortunately we only need to do one set of such experiments for every type of physical switches.

6.3 Effect of ingress port differentiation

Here we evaluate the benefits of ingress port differentiation. The experiment setup is the same as in Fig. 2 with the vSwitches added to form the *Scotch* overlay. An attacker generates the attacking flows, while the client generates the normal flows. The normal new flow rate is set at 30 flows/second, while we vary the attacker's new flow rate. Since we are only interested in switch's control-path performance, we generate one packet for each flow with different IP addresses. For this experiment, we turn off the large flow migration functionality and focus only on the ingress port differentiation feature. We set the queue service rate to be 70 flows/second to make sure the rule insertion will be successful and that it will not affect the data path throughput.

Fig. 12(a) depicts the flow failure rate for the attacking traffic and the normal traffic. The *Scotch* overlay management application maintains two queues and each queue receives at least 35 flows/second for rule insertion. All normal flows can be installed successfully. Attack traffic only uses the leftover capacity from the server, with some of the attack traffic being dropped at the OpenFlow controller. Ingress port differentiation clearly segregates the attack traffic from normal traffic arriving at a different ingress port.

6.4 Benefits of large flow migration and adding additional vSwitches

In this experiment, we examine the effect of large flow migration. To focus on the performance of large flow migration, we turn off the ingress port differentiation in the OpenFlow controller application. All flows arriving at the OpenFlow controller will be routed over the *Scotch* overlay. The large flow will be migrated to the underlying physical network. We set the large flow detection packet count to be 10 packets.

The attacker sends out the attacking traffic (one packet per flow) at a constant rate of 200 flows/second. The client establishes two large flows to the server at time 200 and 400 seconds, respectively. Fig. 12(b) depicts the data-path traffic rate going through the vSwitch and the Pica8 switch, respectively. Since the traffic is forwarded to the vSwitch by the Pica8 switch, the data-path traffic rate going through the Pica8 switch is equal to the total traffic rate. The data-path traffic rate going through the vSwitch is the traffic rate routed over the *Scotch* overlay.

The data-plane traffic rate is 200 packets/second at both the Pica8 switch and the Open vSwtich at the beginning of the experiment. This is because the *Scotch* overlay is on and all traffic passes through both the Pica8 physical switch and the vSwitch. The flow size is small (one packet per flow) so none of the attacking flow is migrated to the physical switch.

At time 200 seconds, a large flow of 80 packets/second starts. We see a small bump in the data-path traffic rate of the vSwitch since the large flow is initially routed through the vSwitch. Once the packet count reaches 10 packets, the large flow detection threshold set in the Scotch application at the controller, the large flow is migrated to the Pica8 switch. The traffic rate at the vSwitch comes back to 200 packets/second. The same happens at time 400 seconds when the second large flow of 60 packets/second arrives. Except for a short time period during which the vSwitch traffic rate experiences a bump, the data-path traffic rate at the vSwitch remains at 200 packets/second. The experiment clearly shows that *Scotch* overlay carries the small attack flows that require large control-path handling but relatively a small data-path throughput.

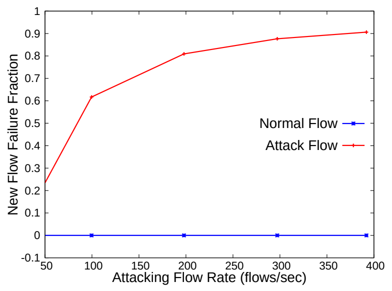
Next we investigate the benefits of additional vSwitches to the *Scotch* overlay. The experimental setup is similar to the large flow migration experiment except that we add one more vSwitch as a *Scotch* node. Fig. 12(c) depicts the data-path traffic rate at the Pica8 switch and two vSwitches, respectively. Here the data-path traffic rate is 100 packets/seconds since the attack traffic is carried by two vSwitches. The large flow arriving at time 200 seconds is first routed through vSwitch 1 and then migrated to Pica8 switch. The second large flow arriving at time 400 seconds is routed through vSwitch 2 at first and then migrated to Pica8 switch. The two vSwitches, however, split the attacking traffic. The *Scotch* overlay's data-path and control-path overall capacities are doubled by adding one more vSwitch. In general, we can scale up the *Scotch* overlay's data-path and control-path capacity by adding more vSwitches.

6.5 Delay of vSwitch relay in Scotch

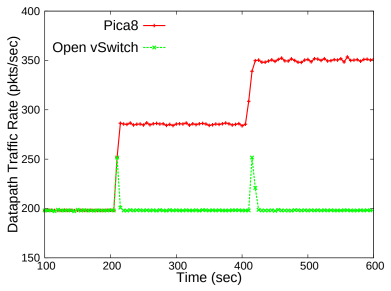
As in any Peer-to-Peer network, *Scotch* overlay incurs both extra bandwidth and delay overhead. There have been extensive studies on the bandwidth overhead caused by the extra traffic of the overlay network, e.g., [32]. Such overhead is in general tolerable. Also in our case, *Scotch* is turned on only if the control path is congested. The alternative would be to drop the new flows, which is clearly less preferrable.

We conduct experiments to evaluate the extra delay overhead caused by *Scotch* overlay. Without considering the middleboxes, when *Scotch* is activated, packets are first sent to a randomly selected vSwitch for load balancing purpose. Packets are then forward to the vSwitch close to the destination host. The later vSwitch finally forwards the traffic to the destination. The extra delay comprises of both propagation and processing delays incurred at both the vSwitches and the extra physical switches along the tunnels. Since the load-balancing vSwitch is randomly selected, the propagation delay on average is doubled when packets are forwarded on the *Scotch* overlay.

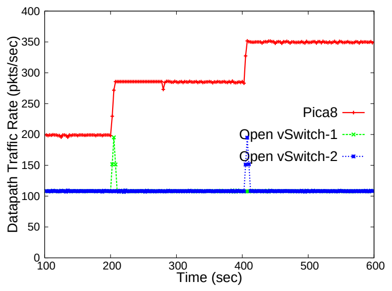
We conduct two delay measurements with and without *Scotch* overlay. In the first experiment, a client periodically sends ping messages to a server via the Pica8 physical switch. In the second experiment, the ping message is detoured to two vSwitches in sequence before reaching the server; each vSwitch represents a vSwitch in the *Scotch* overlay. There is no direct link between the vSwitches; instead they are connected via the Pica8 switch. Hence the packet has to traverse through the physical switch multiple times when it is forwarded on the overlay. Note that we ignore the host vSwitch in both experiments; including it would add a small



(a) Ingress port differentiation

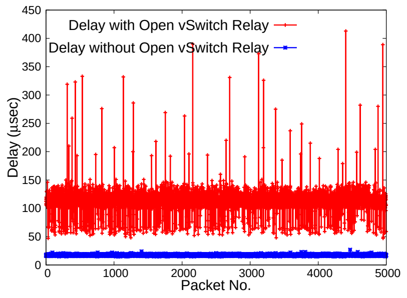


(b) Large flow migration

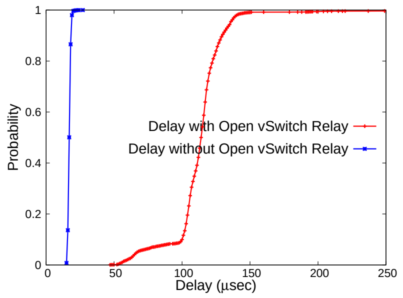


(c) Multiple vSwitches

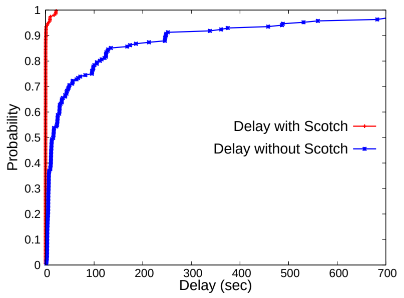
Figure 11: (a) New flow failure fraction with ingress port differentiation (b) Data-path traffic rate at the vSwitch and the Pica8 switch with large flow migration (c) Data-path traffic rate at two vSwitches and Pica8 switch with large flow migration.



(a) Measured delay



(b) CDF of delay



(c) File downloading

Figure 12: (a)(b) Delay with and without vSwitch relay (c) File downloading delay with and without Scotch.

additional delay for both cases but not affecting the comparison. For the convenience of measurement, we run both the client and the server on the same physical host in order to avoid clock synchronization problem.

Fig. 12(a)(b) shows 5,000 measured delay samples and the CDF from the experiments. The delay without overlay is very small, around 17 microseconds. The delay with overlay is on average 113 microseconds. The delay with the vSwitch relay is more volatile, with the standard deviation of 22 microsecond. This indicates that packet processing done by a vSwitch software has larger variance compared to a hardware switch. However, given that the overall delay is still very small, well below 1ms, we believe this satisfies the requirement of most applications in the data-center scenario. If *Scotch* is employed for a wide-area network, the extra switch processing delay should be negligible compared to the propagation delay.

6.6 Case study with data center traffic trace

Finally we conduct a case study with real data center traffic traces [24]. We select the packet trace collected from a university data-center (EDU1 in [1]). The packet trace is collected at a switch inside this data-center. We use Tcpreplay [29] to playback the first 30 minutes of the trace, and send the traffic toward a sink node via the Pica8 switch. To study the benefit of *Scotch* on applications, we set an Apache server that serves out a small file of one Kbytes. A client periodically (every ten seconds) attempts to fetch the file. Both the file download traffic and background traffic go through the same physical switch. We measure the file downloading time for both with and without *Scotch*, and report the results in Fig. 12(c).

As reported in the study [1] (Fig. 3(a)), the number flow rate is slightly greater than 200 flows/sec, which is right above the loss-free control path capacity of the Pica8 switch. Without the help of *Scotch*, 3% of file retrieval fails. For the successful file retrievals, the average downloading time is 71.4 second, with the standard deviation of 133.9 seconds. In contrast, with the *Scotch* overlay, the control path capacity is greatly improved and the client always manages to retrieve the file successfully. The average downloading time is shortened to 0.8 second, with the standard deviation of 3.3 seconds. This result shows that *Scotch* improves the file downloading performance significantly. Note that without *Scotch*, the worst downloading time is 711 seconds. Looking at the tcpdump trace, we notice that due to the control path congestion, it takes multiple attempts to successfully install a flow rule into the switch. Since the expiration time interval for a flow rule is 10 seconds, a flow rule may be timed out before a TCP connection is succesfully set up. This causes the application to make multiple retransmissions before the download succeeds.

7. CONCLUSION

To mitigate the bottleneck of control path of SDN under the surge of control traffic (e.g., due to flash crowds or DDoS attacks), we present *Scotch* that can elastically improve the control plane capability of SDN by using an Openflow vSwitch overlay network that primarily carries small flows. *Scotch* exploits both the high control plane throughput of vSwitches and the high data plane throughput of hardware switches. It enables the control plane throughput to scale linearly with the number of vSwitches used.

While achieving the high control plane capacity, *Scotch* still preserves high visibility of new flows and flexibility of fine-grained flow control at the central controller. We experimentally evaluate the performance of *Scotch* and show its effectiveness in elastically scaling control plane capacity well beyond what is possible with current switches.

8. ACKNOWLEDGMENTS

This study was done while An Wang was a summer intern at Bell Labs, Alcatel-Lucent. It was supported in part by NSF under grants CNS-0746649 and CNS-1117300. The authors thank anonymous reviewers for their constructive comments. Yang Guo would like to thank Muralidharan Kodialam for helpful discussions.

9. REFERENCES

[1] T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *IMC*, 2010.

[2] Z. Cai, A. L. Cox, and T. S. E. Ng. Maestro: Balancing fairness, latency and throughput in the openflow control plane. In *Rice University Technical Report TR11-07*, 2011.

[3] M. Casado, M. J. Freedman, and S. Shenker. Ethane: Taking Control of the Enterprise. In *ACM SIGCOMM*, 2007.

[4] G. Catalli. Open vSwitch: Performance improvement and porting to FreeBSD. In *CHANGE & OFELIA Summer school*, 2011.

[5] Ryu: Component-based software defined networking framework. http://osrg.github.io/ryu/.

[6] A. Curtis et al. Devoflow: Scaling flow management for high-performance networks. *Proc. of SIGCOMM*, 2011.

[7] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an Elastic Distributed SDN Controller. In *HotSDN*, 2013.

[8] Packet Processing - Intel DPDK vSwitch - OVS. https://01.org/packet-processing/intel-ovdk.

[9] D. Erickson. The Beacon OpenFlow Controller. In *HotSDN*. ACM, 2013.

[10] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: an api for application control of sdns. In *SIGCOMM*, 2013.

[11] Floodlight. http://floodlight.openflowhub.org.

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

[13] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. Mckeown, and S. Shenker. NOX: Towards an Operating System for Networks. In *SIGCOMM CCR*, 2008.

[14] hping3. http://linux.die.net/man/8/hping3.

[15] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-Fidelity Switch Models for Software-Defined Network Emulation. In *HotSDN*, 2013.

[16] X. Jin, L. E. Li, L. Vanbever, and J. Rexford. Softcell: Scalable and flexible cellular core network architecture. In *ACM CoNEXT*, 2013.

[17] C. Kim, M. Caesar, and J. Rexford. Floodless in seattle: A scalable ethernet architecture for large enterprises. In *SIGCOMM*, 2008.

[18] T. Koponen et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[19] K. Krishna Puttaswamy Naga, F. Hao, and T.V. Lakshman. Securing software defined networks via flow deflection, 812383-US-NP.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, et al. Openflow: enabling innovation in campus networks. *SIGCOMM CCR*, 2008.

[21] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, 2009.

[22] Personal Communication with Pica8.

[23] Pica8: Open networks for software-defined networking. http://www.pica8.com/.

[24] Packet trace at a switch in a data-center. http://pages.cs.wisc.edu/ tbenson/IMC10_Data.html.

[25] S. Ray, R. Guerin, and R. Sofia. A distributed hash table based address resolution scheme for large-scale ethernet networks. In *ICC*, 2007.

[26] A. Shaikh and A. Greenberg. Making routers last longer with ViAggre. In *NSDI*, 2009.

[27] S. Shin, P. Porras, V. Yegneswaran, M. Fong, G. Gu, and M. Tyson. Fresco: Modular composable security services for software-defined networks. In *NDSS*, 2013.

[28] S. Shin, V. Yegneswaran, P. Porras, and G. Gu. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[29] Tcpreplay. http://tcpreplay.synfin.net/.

[30] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood. On Controller Performance in Software-Defined Networks. In *HotICE*, 2012.

[31] The overhead of software tunneling. http://networkheresy.com/2012/06/08/the-overhead-of-software-tunneling/.

[32] Yang-hua Chu, S. Rao, S. Seshan, and H. Zhang. A case for end system multicast. In *IEEE Journal on Selected Areas in Communications*, Oct. 2002.

[33] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with DIFANE. In *SIGCOMM*, 2010.