Gotta Tell You Switches Only Once: Toward Bandwidth-Efficient Flow Setup for SDN

Kai Bu*
College of Computer Science and Technology, Zhejiang University
Email: kaibu@zju.edu.cn (*corresponding author)

***Abstract*—Software-Defined Networking (SDN) has greatly enriched the flexibility of network management. It introduces a central controller to take over most network functions that otherwise reside in distributed forwarding devices. Such a centralized design, however, tends to make control channel a bottleneck due to bandwidth fatigue. Existing work saves control channel bandwidth at the expense of losing visibility of most or mice flows. This paper proposes *FastLane*, a framework for bandwidth-efficient SDN flow setup without sacrificing global flow visibility. FastLane advocates that the controller inform only a flow's ingress switch of the flow's forwarding path while switches themselves cooperate to complete flow setup. This way, FastLane keeps minimum traffic for flow setup in control channel and leaves the rest to data plane. The analytical results validate FastLane's higher bandwidth efficiency over traditional SDN, especially for relatively long forwarding paths. For a 3-switch path, FastLane can already save more than a half of bandwidth. The saved bandwidth can embrace more flow setups and thus reduces flow latency.**

I. Introduction

With flexible network management brought by Software-Defined Networking (SDN) come also potential bottlenecks. SDN significantly improves network management flexibility through consolidating network functions from forwarding devices to a central controller [1]. The centralized design, however, makes the controller a potential bottleneck [2]. Another potential bottleneck lies in limited control channel bandwidth switches can support. An HP ProCurve 5406zl switch [3], for example, supports only 17 Mbps control channel bandwidth, which can hardly satisfy low-latency flow setup and fine-grained statistics collection at the same time for large networks [4]. Furthermore, Ternary Content Addressable Memory (TCAM) for switches installing non-exact match rules is expensive, power-hungry, and usually scarce on switches [2]. Most commercial switches can accommodate at most thousands of rules [4], [5]. This leads to switches frequently querying the controller for rules and further saturates control channel (Figure 1(a)). Solutions against these bottlenecks are critical for SDN to make the most of its innovation. In this paper, we primarily target control channel bottleneck.

Existing solutions mitigate control channel bottleneck but cost the controller of visibility over most or mice flows. Typical such solutions, DIFANE [2] and DevoFlow [4], economize control channel usage by devolving controller workload to data plane. DIFANE introduces authority switches to relieve the controller from cumbersome flow setup. The controller preinstalls rules on authority switches, which will later accommodate rule queries from normal switches. Flow setup traffic then stays between authority switches and normal switches, rarely invoking the controller. The controller can, however, hardly maintain global flow visibility [4]. DevoFlow, on the other hand, proposes rule cloning technique and enables switches to locally handle flow setup. DevoFlow mitigates control channel bottleneck while focusing more on tracking elephant flows rather than mice flows [4].

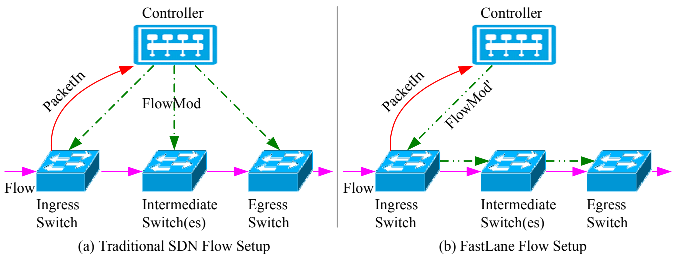

Fig. 1. Comparison of flow setup by (a) traditional SDN and (b) FastLane. FastLane keeps minimum traffic for flow setup in control channel and leaves the rest to data plane.

In this paper, we propose *FastLane* for bandwidth-efficient flow setup without sacrificing flow visibility. Figure 1 sketches the key difference of FastLane from traditional SDN. As in Figure 1(b), FastLane completes flow setup by requiring that the controller inform only a flow's ingress switch of the flow's processing rules. Switches on the forwarding path then cooperate to install the informed rules, instead of being individually contacted by the controller as traditional SDN requires (Figure 1(a)). This way, FastLane strives for keeping minimum traffic for flow setup in control channel and leaving the rest to data plane. Since FastLane preserves reactive flow setup, it keeps flow visibility intact as traditional SDN does. Our analysis shows that FastLane can achieve not only bandwidth-efficient but also faster flow setup for SDN. In line with DIFANE [2] and DevoFlow [4], FastLane imposes minor software modifications on the controller and switches.

We highlight the paper's contributions to mitigating SDN control channel bottleneck as follows.

- Propose a bandwidth-efficient yet flow-visibility–intact flow setup framework with the controller communicating with only a flow's ingress switch.
- Present FastLane design to implement the bandwidth-efficient flow setup framework.
- Demonstrate FastLane's higher bandwidth efficiency and lower flow latency over traditional SDN through analysis.

II. SDN and Why a Flow Could Be Slow

In this section, we first introduce SDN basics and then review its performance bottlenecks and countermeasures.

A. SDN Basics

SDN simplifies network management with a centralized controller taking over management functions that traditionally reside in switches [1]. The controller hosts various network management applications such as access control, routing, and load balancing. Applications translate high-level policies to rules, which are further pushed to switches by the controller. A policy dictates how an administrator would like to regulate network traffic while through the form of a rule can switches interpret the policy. For example, an access control policy might require users from computer science department (in the 10.10.0.0/16 subnet) not connect finance office. The corresponding rule to be installed on the ingress switch of finance office subnet then is [src=10.10.*.*, drop]. The ingress switch will drop any incoming packet with 10.10 as the prefix of its source IP address.

As a finer-grained version of Figure 1(a), Figure 2 illustrates how SDN steers a flow of packets. Specifically, we consider how SDN implements the example policy that directs packets from the 10.20.0.0/16 subnet through a three-switch network. To implement the policy, rules to be installed on switches sw1, sw2, and sw3 are respectively [src=10.20.*.*, fwd(2)], [src=10.20.*.*, fwd(3)], and [src=10.20.*.*, out]. Switches have limited TCAM space for storing rules and thus usually install rules reactively [2]. As in Figure 2, without pre-installing rule [src=10.20.*.*, fwd(2)], switch sw1 is agnostic to how to process the first packet (pkt1) from the 10.20.0.0/16 subnet (step 1a). Switch sw1 needs to buffer packet pkt1, encapsulates its packet header in a PacketIn message (step 1b), and forwards PacketIn to the controller (step 1c). The controller then populates the corresponding rules to switches through FlowMod messages (step 1d). Switches then install received rules to data plane (step 1e) and correctly direct packet pkt1 along the designated forwarding path (steps 1f, 1g, and 1h). When the second packet (pkt2) from the 10.20.0.0/16 subnet arrives, switches direct it following the matching rules (steps 2a, 2b, 2c, and 2d) without involving the controller. When the number of rules on a switch increases, a packet might match multiple rules. The controller therefore also needs to assign each rule a priority. Switches process packets following the matching rule with the highest priority [2]. Besides, accompanying a rule are counters that track flow statistics such as amount and duration [4].

B. SDN Bottlenecks and Countermeasures

The flexibility of network management brought by SDN, however, comes at the cost of potential bottlenecks [6]. Such bottlenecks may happen to the controller, control channel for controller-switch communication, and switches [2], [4], [5].

Controller bottleneck. The centralized design of the controller makes it vulnerable to single-point failure. In SDN, the controller takes over most network management applications

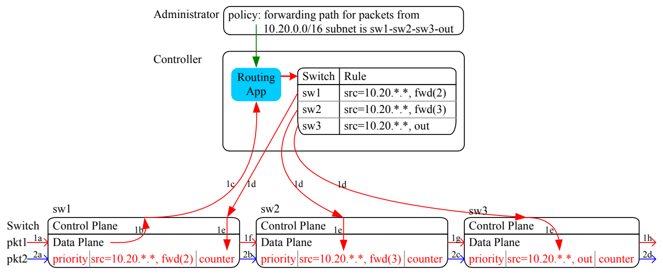

Fig. 2. SDN flow setup and forwarding example.

previously embedded in forwarding devices. Among many others, typical such applications measure network performance [7], balance traffic load [8], schedule network updates [9], and verify configuration correctness [10]. To support these applications, the controller also needs to log network-wide configuration specifics (e.g., rules on each switch as in Figure 2). Although every application independently strives for minimum overhead, together they make the controller resource-hungry. Furthermore, since the controller installs flow rules reactively, its another heavy workload is to welcome the flooding PacketIn messages and populates corresponding rules to switches (e.g., steps 1c and 1d in Figure 2). Even worse, this process might be exploited for DDoS attacks [11], [12].

Existing countermeasures against controller bottleneck either design a distributed controller [13]–[15] or devolve controller workload to switches [2]. Since controller bottleneck is due mainly to centralization, the most intuitive solution is switch to distributed design. Several distributed controller designs such as HyperFlow [13], Onix [14], and ElastiCon [15] have addressed various key challenges like controller synchronization and state migration. Despite the benefits of distributed controller design, solutions for unloading a single controller would still be necessary. DIFANE [2], to this end, proposes devolving controller workload to switches. Specifically, it introduces authority switches that are pre-fed with rules matching all flows. Then authority switches rather than the controller populate rules to switches upon arrival of new flows. While shifting flow management to switches, DIFANE deprives the controller of global visibility of flow states [4].

Control channel bottleneck. The centralized controller design further makes control channel a heavily used medium. Two main types of control channel traffic are for managing flow rules and gathering flow statistics. *First*, most traffic for managing flow rules is by flow setup (e.g., PacketIn and FlowMod messages as steps 1c and 1d in Figure 2). DevoFlow reports that [4] a one-way flow setup generates $94 + 144N$ bytes for an N-switch path. This leads to one control packet for every two or three packets entering a three-switch path and thus a lot of control channel traffic [4]. Messages for rule deletion or other update operations further saturate the control channel [9]. *Second*, collecting flow statistics may cost more bandwidth than flow setup does. Timely and accurate flow statistics are necessary for effective flow schedulers [16]. Statistics collection and flow setup compete for the control channel bandwidth. Collecting finer-grained statistics lowers flow setup rate. As DevoFlow measured, an HP ProCurve 5406zl switch

[3] can support 17 Mbps control channel bandwidth, which is even not sufficient for collecting statistics for 16,000 exact-match rules and 1,500 wild-card rules twice per second [4].

Devolving controller workload to switches mitigates not only controller bottleneck but also control channel bottleneck [2], [4]. Unlike DIFANE [2] that introduces authority switches to take over most of the controller's flow management tasks, DevoFlow [4] enables switches to locally handle certain flow setups and suggests efficient statistics collection methods. Their bandwidth efficiency gains come with the cost of losing global visibility of most or mice flows [4].

Switch bottleneck. Limited internal bandwidth and CPU capacity confine the traffic processing capability of switches [4]. Take again the 5406zl switch for example. The bandwidth between the line-card ASIC and the management CPU is about 80 Mbps (e.g., steps 1b and 1e in Figure 2) while the ASIC has a raw bandwidth of 300 Gbps [4]. Another indirect cause for switch bottleneck is its limited TCAM space for installing rules. The 5406zl switch supports about 1,500 rules [4], which are far from enough for fine-grained flow management. Switches, therefore, need to frequently invoke the controller for fetching uninstalled rules. This process (e.g., steps 1b-1e in Figure 2) consumes not only switch internal bandwidth and switch CPU but also control channel bandwidth and controller CPU.

While strengthening switch internal bandwidth and CPU may have to wait for break-through innovations, existing research efforts start economizing TCAM usage. SwitchReduce proposes compressing same-action rules [5]. A SwitchReduce rule can match all packets with the same forwarding action, regardless of whether their packet headers are identical. CacheFlow augments switches with additional rule caches [17]. The on-switch TCAM installs popular rules to handle most packets with line-speed forwarding. Less popular rules are installed in the augmented cache, from where rather than the controller switches may fetch rules for unmatched flows.

III. Steering Flows to FastLane

In this section, we present FastLane, an SDN flow setup framework that mitigates control channel bottleneck without sacrificing flow visibility. FastLane advocates that the controller inform only a flow's ingress switch of the flow's forwarding path (Figure 1(b)). The path information is then passed along among the ingress switch and en-route switches (i.e., intermediate switch(es) and egress switch). Not directly installing rules to en-route switches by the controller saves control channel bandwidth, especially when the forwarding path is relatively long. The saved bandwidth can benefit more flow setups, finer-grained statistics collection, or other control traffic.

A. Motivation

FastLane strives for saving control channel bandwidth without sacrificing flow visibility. Observing traditional SDN flow setup a bit deeper (Figure 1(a)), we find that the controller can maintain a global flow visibility if each unmatched flow's ingress switch generates a PacketIn message. For successful flow setup, the controller need also issue FlowMod messages. However, since switches are designed for fast forwarding, we argue that the controller can inform only the ingress switch of the forwarding path (in a FlowMod' message) and let switches themselves coordinate to pass along the FlowMod' message and correctly install rules (Figure 1(b)). This way, FastLane keeps the minimum traffic for flow setup in the control channel while leaving others to data plane. Take the 3-switch case in Figure 1 for example. Traditional SDN flow setup uses four packets (1 PacketIn + 3 FlowMod) while FastLane only two (1 PacketIn + 1 FlowMod'). The network load of PacketIn and of FlowMod is respectively 94 bytes and 144 bytes [4]. For simplicity, we temporarily assume that FlowMod' and FlowMod have comparative amount of network load. (More generalized analysis is available in Section IV-A.) FastLane thus saves $1 - \frac{94+144}{94+144\times3} = 55\%$ of control channel bandwidth, being 2.2x bandwidth-efficient than traditional SDN.

B. FastLane Design

Pre-thought. To achieve the FastLane design as illustrated in Figure 1(b), we need to address three key questions.

- Q1: How does the controller inform the ingress switch of the forwarding path with FlowMod'?
- Q2: How does the ingress switch install the associated rule to switch data plane and forward the FlowMod' to next-hop switch?
- Q3: How does an en-route switch install the associated rule to switch data plane and forward the FlowMod' to next-hop switch?

For addressing the preceding questions, FastLane makes minor software modifications to the controller and switches. Controller software modification is necessary for addressing Q1. The modification enables the controller to encapsulate the entire path instead of only the next hop into FlowMod'. Switch software modification helps address Q2 and Q3. Since FlowMod' is no longer in the format of a flow rule, the ingress switch should adapt to extracting the associated rule from FlowMod' and then forwarding FlowMod' to next hop. Furthermore, when an en-route switch receives FlowMod', it needs to locally handle rule installation and FlowMod' forwarding without re-directing FlowMod' to the controller.

FastLane flow setup redesigns the process of installing rules to switches toward saving control channel bandwidth. On the other hand, it does not modify the process of how an ingress switch directs PacketIn to the controller and how the controller compiles the forwarding path. We detail FastLane design along with how we address the aforementioned three questions.

Solution of Q1: The controller informs only the ingress switch of the forwarding path with FlowMod' message. FlowMod' is a modified version of FlowMod. The Action field is replaced with the FwdPath field while the Priority field is replaced with the PriorityPath field that comprises a sequence of priorities corresponding to each switch on the path. We capture the difference of FlowMod' from FlowMod in Formula 1 and Formula 2, where $||$ represents concatenation.

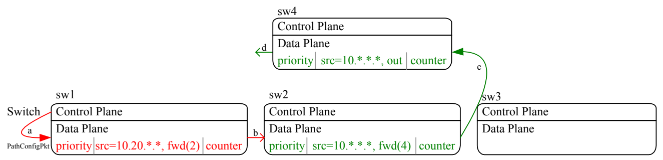

Fig. 3. Example of ineffective forwarding-path configuration. The goal is to configure the sw1-sw-sw3-out path for a flow from the 10.20.*.*/16 subnet. Simply copying the header of a src=10.20.*.* packet, PathConfigPkt is wrongly forwarded to sw4 instead of sw2's control plane.

$$\text{FlowMod} = \text{Priority} \quad || \text{ Header} \quad || \text{ Action}; \tag{1}$$

$$\text{FlowMod'} = \text{PriorityPath} \quad || \text{ Header} \quad || \text{ FwdPath}. \tag{2}$$

When Action for the ingress switch is "drop", FlowMod' is faced with a one-switch forwarding-path and specialized to FlowMod. But more often, FwdPath in FlowMod' consists of more than one switch.

Solution of Q2: The ingress switch 1) extracts the rule indicated by FlowMod', 2) installs the rule, and 3) constructs a data packet called PathConfigPkt with FlowMod' as payload and then feeds PathConfigPkt to switch data plane. To achieve Steps 1 and 2, the ingress switch extracts next-hop switch $NextHop$ and its associated priority P from FwdPath and PriorityPath, respectively. It then constructs rule R:

$$R = P \text{ || Header || } NextHop, \tag{3}$$

and installs R to switch data plane. The ingress switch now knows how to handle packets in the flow led by the packet that triggered PacketIn.

For Step 3, configuring PathConfigPkt's header is relatively complicated. Simply copying the header of R-matching packets is not effective. Since any switch could be an ingress switch that initiates installing certain rules, a switch can hardly guarantee a full knowledge of rules on its next-hop switch. Upon receiving PathConfigPkt, if a switch already has rules matching it, PathConfigPkt will be accordingly forwarded without being directed to switch control plane. This likely violates the forwarding path dictated by the controller. Figure 3 shows such an example. The example intends to forward packets from the 10.20.0.0/16 subnet via path sw1-sw2-sw3-out while switch sw2 already caches a rule matching packets from the 10.0.0.0/24 subnet. With header copied from a packet with, for example, source IP address 10.20.0.0, PathConfigPkt will be wrongly forwarded to sw4 by sw2.

Solution of Q2-Step3 and Q3: The controller maps each switch to an index and pre-installs on each switch a highest-priority to-switch-CPU rule defined in Formula 4.

$$\text{ToSwitchRule=HighestPriority||SwitchIndex||to switch CPU.} \tag{4}$$

To support ToSwitchRule, the packet header needs sufficient unused bits as SwitchIndex. Such bits are usually readily available and have been leveraged for SwitchReduce compressing rules [5] and FlowTags enforcing policies [18]. ToSwitchRule directs any packet with SwitchIndex-bits assigned to switch control plane.

We now associate the motivation of ToSwitchRule with how it helps constructing PathConfigPkt for *Q2-Step3*. If PathConfigPkt can match ToSwitchRule, it can be correctly directed to

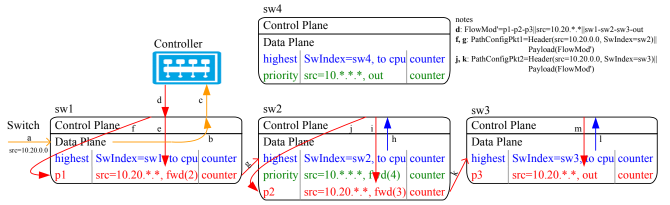
Controller
sw4
Control Plane
Data Plane
highest | SwIndex=sw4, to cpu | counter
priority | src=10.*.*.*, out | counter
notes
d: FlowMod'=p1-p2-p3||src=10.20.*.*||sw1-sw2-sw3-out
f, g: PathConfigPkt1=Header(src=10.20.0.0, SwIndex=sw2)|| Payload(FlowMod')
j, k: PathConfigPkt2=Header(src=10.20.0.0, SwIndex=sw3)|| Payload(FlowMod')
d
c
sw1
Switch
a
src=10.20.0.0
Control Plane
Data Plane
f
e
b
highest | SwIndex=sw1, to cpu | counter
p1 | src=10.20.*.*, fwd(2) | counter
g
sw2
Control Plane
Data Plane
j
i
h
highest | SwIndex=sw2, to cpu | counter
priority | src=10.*.*.*, fwd(4) | counter
p2 | src=10.20.*.*, fwd(3) | counter
k
sw3
Control Plane
Data Plane
m
l
highest | SwIndex=sw3, to cpu | counter
p3 | src=10.20.*.*, out | counter

Fig. 4. FastLane flow setup example. The goal is to configure the sw1-sw2-sw3-out path for a flow from the 10.20.*.*/16 subnet. SwIndex stands for SwitchIndex.

switch control plane for keeping the flow setup process alive. For PathConfigPkt to match ToSwitchRule on a switch, its header's SwitchIndex-bits should be set as the switch's index. For example, if switch sw2 in Figure 3 caches a ToSwitchRule and PathConfigPkt from switch sw1 is with SwitchIndex=sw2, the PathConfigPkt can be correctly directed to sw2's control plane rather than sw4. Furthermore, for PathConfigPkt to be correctly forwarded, it also needs to match the rule extracted from its carrying FlowMod'. As aforementioned, this requires that PathConfigPkt's header follow the packet that triggers FlowMod'. Putting foregoing pieces together, we construct PathConfigPkt by Formula 5.

$$\begin{aligned}\text{PathConfigPkt} = \text{Pkt(FlowMod').Header[SwitchIndex=NextHop] ||}\\ \text{Payload[FlowMod'].}\end{aligned} \tag{5}$$

Pkt(FlowMod').Header[SwitchIndex=NextHop] in the first line composes PathConfigPkt's header. Pkt(FlowMod') denotes the packet that triggers PacketIn and FlowMod'. Pkt(FlowMod').Header[SwitchIndex=NextHop] corresponds to Pkt(FlowMod')'s header with SwitchIndex-bits set as NextHop. Payload[FlowMod'] in the second line uses FlowMod' as the payload of PathConfigPkt. Other en-route switches can reuse PathConfigPkt's header with SwitchIndex-field modified. The egress switch of a flow need only install the rule extracted from received PathConfigPkt without re-constructing or forwarding the PathConfigPkt.

FastLane practice. We now exercise FastLane with a flow setup instance as shown in Figure 4. Same as Figure 3, Figure 4 aims to configure the sw1-sw2-sw3-out path for packets from the 10.20.0.0/16 subnet given that sw2 caches a rule for forwarding packets from the 10.0.0.0/24 subnet to sw4. Different from Figure 3, Figure 4 pre-installs on each switch a ToSwitchRule by Formula 4. FastLane correctly finishes flow setup for packets from the 10.20.0.0/16 subnet, requiring that the controller communicate with only ingress switch sw1. We walk through how FastLane achieves bandwidth-efficient flow setup in Figure 4 step-wise as follows.

Ingress switch: sw1. Upon arrival of the packet with source IP address 10.20.0.0 (step a), switch sw1 finds no rule for handling it and thus queries the controller using a PacketIn message (steps b and c). The controller compiles the forwarding path and informs it to sw1 in a FlowMod' message (step d) as shown in Formula 6:

$$\text{FlowMod'=p1-p2-p3 || src=10.20.*.* || sw1-sw2-sw3-out.} \tag{6}$$

FlowMod' in Formula 6 requires that packets from the 10.20.0.0/16 subnet traverse through sw1, sw2, and sw3.

Corresponding rules on sw1, sw2, and sw3 are assigned with priorities of p1, p2, and p3, respectively. Upon receiving FlowMod', switch sw1 first extracts specified header src=10.20.*.*, $NextHop$=sw2, and corresponding priority P=p1, and then constructs a rule R by Formula 3. Switch sw1 then installs the rule on its data plane (step e). Now switch sw1 can correctly forward src=10.20.*.*-packets to sw2 per the controller's instruction. It needs to pass on the forwarding instruction to switches sw2 and sw3. To this end, switch sw1 constructs PathConfigPkt1 by Formula 5:

$$\text{PathConfigPkt1} = \text{Header(src=10.20.0.0, SwithIndex=sw2)} \;||\; \text{Payload(FlowMod')}.$$

Switch sw1 then feeds PathConfigPkt1 to its data plane (step f). PathConfigPkt1 matches the src=10.20.*.*-rule and is forwarded to switch sw2 (step g).

Intermediate switch: sw2. Upon receiving PathConfigPkt1, switch sw2 matches it with the highest-priority SwitchIndex=sw2-rule and directs it to control plane (step h). Switch sw2 extracts specified header src=10.20.*.*, $NextHop$=sw3, and corresponding priority P=p2, and then constructs a rule R by Formula 3. Rule R is then installed to sw2's data plane (step i). Switch sw2 further constructs PathConfigPkt2 using a copy of PathConfigPkt1 with SwitchIndex modified to sw3 (step j).

$$\text{PathConfigPkt2} = \text{Header(src=10.20.0.0, SwithIndex=sw3)} \;||\; \text{Payload(FlowMod')}.$$

Switch sw2 feeds PathConfigPkt2 to data plane and forwards it to sw3 according to the src=10.20.*.*-rule (step k).

Egress switch: sw3. As the egress switch of src=10.20.*.*-flow, sw3 need only extract src=10.20.*.*-rule from PathConfigPkt2 and install it. It need not further re-construct or forward PathConfigPkt. Since PathConfigPkt2 matches the highest-priority ToSwitchRule, switch sw3 directs it to control plane (step l). Switch sw3 then extracts specified header src=10.20.*.*, $NextHop$=out, and corresponding priority P=p3, and constructs a rule R by Formula 3. Finally, switch sw3 installs rule R to data plane (step m) and completes flow setup for packets from the 10.20.0.0/16 subnet.

In summary, FastLane achieves bandwidth-efficient flow setup by keeping minimum control channel traffic between the controller and the ingress switch. The controller no longer informs all switches on the forwarding path of intended rules. It is switches that pass on the rules and install them. Take the example in Figure 4 for instance. FastLane imposes only two packets on control channel whereas traditional SDN in Figure 1(a) requires four. Furthermore, FastLane preserves global flow visibility as traditional SDN does because it still directs each unmatching flow to the controller.

IV. Performance

In this section, we analyze FastLane's performance in terms of bandwidth efficiency and flow latency. The performance analysis aims to answer the following two questions.

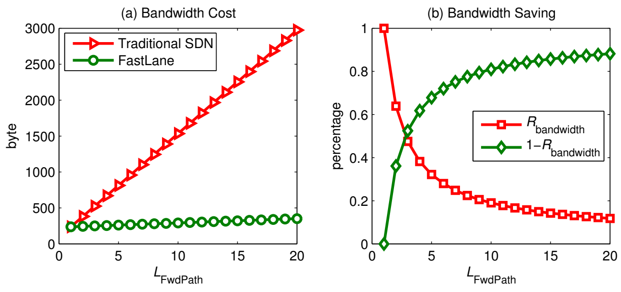

Fig. 5. Comparison of bandwidth efficiency per flow setup by traditional SDN and FastLane.

- Q1: How much bandwidth can FastLane save for SDN flow setup?
- Q2: How does FastLane affect flow latency?

A. Bandwidth Efficiency

We analyze how much bandwidth FastLane saves per flow setup. In comparison with traditional SDN flow setup, FastLane encapsulates priorities and actions for all switches on the forwarding path in one FlowMod' message (Formula 2). This way, FastLane eliminates redundant control channel traffic induced by Header and some other payloads in multiple FlowMod messages. Let L_{PacketIn}, L_{FlowMod}, and $L_{\text{FlowMod}'}$ denote the network load of PacketIn, FlowMod, and FlowMod', respectively. Given a forwarding path with L_{FwdPath} switches, traditional SDN generates a volume $L_{\text{PacketIn}} + L_{\text{FlowMod}} \times L_{\text{FwdPath}}$ of control channel traffic [4]. The traffic generated by FastLane is between only the controller and the ingress switch; its volume is $L_{\text{PacketIn}} + L_{\text{FlowMod}'}$. Let L_{Priority} and L_{Action} denote the network load of Priority field and Action field, respectively, in FlowMod and FlowMod'. The ratio $R_{\text{bandwidth}}$ of FastLane's bandwidth consumption to traditional SDN's is as follows.

$$\begin{aligned}R_{\text{bandwidth}} &= \frac{L_{\text{PacketIn}} + L_{\text{FlowMod}'}}{L_{\text{PacketIn}} + L_{\text{FlowMod}} \times L_{\text{FwdPath}}} \\ &= \frac{L_{\text{constant}} + (L_{\text{Priority}} + L_{\text{Action}})(L_{\text{FwdPath}} - 1)}{L_{\text{constant}} + L_{\text{FlowMod}} \times (L_{\text{FwdPath}} - 1)},\end{aligned}$$

where we have $L_{\text{constant}} = L_{\text{PacketIn}} + L_{\text{FlowMod}}$. The lower $R_{\text{bandwidth}}$ is, the more bandwidth FastLane saves per flow setup. $R_{\text{bandwidth}}$ is upper bounded by $\frac{L_{\text{Priority}}+L_{\text{Action}}}{L_{\text{FlowMod}}}$.

Figure 5 compares FastLane's bandwidth efficiency with that of traditional SDN. The configuration adopts 94-byte PacketIn, 144-byte FlowMod [4], 2-byte Priority field, and 4-byte Action field [19]. For each flow setup, traditional SDN generates $(94 + 144L_{\text{FwdPath}})$-byte control channel traffic while FastLane generates $(94{+}144{+}(2{+}4){\times}(L_{\text{FwdPath}}{-}1))$-byte. Both bandwidth consumption of traditional SDN and FastLane is linear with the length L_{FwdPath} of forwarding path. As shown in Figure 5(a), FastLane costs less bandwidth than does traditional SDN. The bandwidth-cost gap widens as L_{FwdPath} increases. The ratio $R_{\text{bandwidth}}$ of FastLane's bandwidth cost to traditional SDN's thus decreases with L_{FwdPath}, as shown in Figure 5(b). When $L_{\text{FwdPath}} = 3$, which is a commonly short path [4], FastLane already saves more than a half of bandwidth for SDN completing a flow setup (i.e., when $1 - R_{\text{bandwidth}} > R_{\text{bandwidth}}$ in Figure 5(b)).

B. Flow Latency

We now analyze how FastLane affects average flow latency, that is, the average time per flow setup.

First, we sketch the analysis of average flow latency for traditional SDN flow setup. Let $L^{\mathrm{avg}}_{\mathrm{FwdPath}}$ denote the average length of forwarding paths. Toward saturating the control channel, each switch corresponds to one PacketIn and $L^{\mathrm{avg}}_{\mathrm{FwdPath}}$ FlowMod for $L^{\mathrm{avg}}_{\mathrm{FwdPath}}$ flow setups on average. Let T_{PacketIn} and T_{FlowMod} denote the time cost induced by PacketIn and FlowMod, respectively. The average time for traditional SDN to complete a flow setup approximates as follows.

$$\begin{aligned} T_{\mathrm{traditional}} &= \frac{T_{\mathrm{PacketIn}} + T_{\mathrm{FlowMod}} \times L^{\mathrm{avg}}_{\mathrm{FwdPath}}}{L^{\mathrm{avg}}_{\mathrm{FwdPath}}} \\ &= \frac{T_{\mathrm{PacketIn}}}{L^{\mathrm{avg}}_{\mathrm{FwdPath}}} + T_{\mathrm{FlowMod}}. \end{aligned}$$

Second, we outline the analysis of FastLane's average flow setup time. For ease of comparison with traditional SDN, we consider the same amount of control channel bandwidth per switch (i.e., $L_{\mathrm{PacketIn}} + L_{\mathrm{FlowMod}} \times L^{\mathrm{avg}}_{\mathrm{FwdPath}}$) for FastLane flow setup. The number $N^{\mathrm{FastLane}}_{\mathrm{flow}}$ of flow setups supported by FastLane is as follows.

$$\begin{aligned} N^{\mathrm{FastLane}}_{\mathrm{flow}} &= L^{\mathrm{avg}}_{\mathrm{FwdPath}} \times \frac{L_{\mathrm{PacketIn}} + L_{\mathrm{FlowMod}} \times L^{\mathrm{avg}}_{\mathrm{FwdPath}}}{L_{\mathrm{PacketIn}} + L_{\mathrm{FlowMod'}}} \\ &= \frac{L^{\mathrm{avg}}_{\mathrm{FwdPath}}}{R_{\mathrm{bandwidth}}}. \end{aligned}$$

Let T_{SwRTT} denote the round trip time between switch control plane and data plane. The average time for FastLane to complete a flow setup can be estimated as follows.

$$\begin{aligned} T_{\mathrm{FastLane}} &= \frac{T_{\mathrm{traditional}} + (N^{\mathrm{FastLane}}_{\mathrm{flow}} - 1) T_{\mathrm{SwRTT}}}{N^{\mathrm{FastLane}}_{\mathrm{flow}}} \\ &= \frac{R_{\mathrm{bandwidth}}}{L^{\mathrm{avg}}_{\mathrm{FwdPath}}} \times T_{\mathrm{traditional}} + \left(1 - \frac{R_{\mathrm{bandwidth}}}{L^{\mathrm{avg}}_{\mathrm{FwdPath}}}\right) T_{\mathrm{SwRTT}}. \end{aligned}$$

According to the measurements of $T_{\mathrm{PacketIn}} + T_{\mathrm{FlowMod}} = 2$ ms and $T_{\mathrm{SwRTT}} = 0.5$ ms [4], we conclude that FastLane imposes no more latency to each flow on average than traditional SDN does because of $T_{\mathrm{FastLane}} \leq T_{\mathrm{traditional}}$. The equality happens when $L^{\mathrm{avg}}_{\mathrm{FwdPath}} = 1$, that is, when FastLane resembles traditional SDN; this conforms to the previous analysis in Section IV-A.

V. Conclusion

We have studied bandwidth-efficient flow setup against SDN control channel bottleneck and proposed FastLane design. FastLane completes flow setup by requiring communication between the controller and only a flow's ingress switch. FastLane thus saves other en-route switches's associated control channel bandwidth, which otherwise is required by traditional SDN. The saved bandwidth can benefit faster flow setup and finer-grained statistics collection. Furthermore, FastLane gains bandwidth efficiency without sacrificing flow visibility. Future work lies in implementing and evaluating FastLane with NOX controller [20] and click-based OpenFlow Switch [21] on MiniNet testbed [22].

Acknowledgment

This work is supported in part by the Fundamental Research Funds for the Central Universities under Grant No. 2014QNA5012 and the National Science Foundation of China under Grant No. 61402404. The author would also like to sincerely thank IEEE SDDCS 2015 chairs and reviewers for their helpful feedback.

References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with difane," in *ACM SIGCOMM*, 2010, pp. 351–362.

[3] HP ProCurve 5400 zl switch series. [Online]. Available: http://h17007.www1.hp.com/us/en/products/switches/HP_E5400_zl_Switch_Series/index.aspx

[4] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *ACM SIGCOMM*, 2011, pp. 254–265.

[5] A. S. Iyer, V. Mann, and N. R. Samineni, "Switchreduce: Reducing switch state and controller involvement in openflow networks," in *IFIP Networking*, 2013, pp. 1–9.

[6] D. Kreutz, F. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *arXiv preprint arXiv:1406.0440*, 2014.

[7] M. Moshref, M. Yu, R. Govindan, and A. Vahdat, "Dream: dynamic resource allocation for software-defined measurement," in *ACM SIGCOMM*, 2014, pp. 419–430.

[8] R. Wang, D. Butnariu, J. Rexford, *et al.*, "Openflow-based server load balancing gone wild," in *USENIX Hot-ICE*, 2011.

[9] X. Wen, C. Diao, X. Zhao, Y. Chen, L. E. Li, B. Yang, and K. Bu, "Compiling minimum incremental update for modular sdn languages," in *ACM HotSDN*, 2014, pp. 193–198.

[10] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey, "Veriflow: Verifying network-wide invariants in real time," in *USENIX NSDI*, 2013, pp. 15–27.

[11] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "Avant-guard: scalable and vigilant switch flow management in software-defined networks," in *ACM CCS*, 2013, pp. 413–424.

[12] M. Dhawan, R. Poddar, K. Mahajan, and V. Mann, "Sphinx: Detecting security attacks in software-defined networks," in *NDSS*, 2015.

[13] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *USENIX INM/WREN*, 2010.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, *et al.*, "Onix: A distributed control platform for large-scale production networks." in *USENIX OSDI*, 2010, pp. 351–364.

[15] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Elasticon: an elastic distributed sdn controller," in *ACM ANCS*, 2014, pp. 17–28.

[16] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *USENIX NSDI*, 2010.

[17] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite cacheflow in software-defined networks," in *ACM HotSDN*, 2014, pp. 175–180.

[18] 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 *USENIX NSDI*, 2014.

[19] FlowMod. [Online]. Available: http://flowgrammable.org/sdn/openflow/message-layer/flowmod/#tab_ofp_1_3

[20] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[21] Y. Mundada, R. Sherwood, and N. Feamster, "An openflow switch element for click," in *Symposium on Click Modular Router*, 2009.

[22] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *ACM Hotnets*, 2010.