Improving SDN Scalability with Protocol-Oblivious Source Routing: A System-Level Study

Article *in* IEEE Transactions on Network and Service Management · October 2017

DOI: 10.1109/TNSM.2017.2766159

CITATION
1

READS
59

8 authors, including:

Shengru Li
University of Science and Technology of China
14 PUBLICATIONS **71** CITATIONS
SEE PROFILE

Nirwan Ansari
New Jersey Institute of Technology
616 PUBLICATIONS **10,934** CITATIONS
SEE PROFILE

Qinkun Bao
University of Science and Technology of China
4 PUBLICATIONS **5** CITATIONS
SEE PROFILE

Zuqing Zhu
University of Science and Technology of China
217 PUBLICATIONS **2,002** CITATIONS
SEE PROFILE

Improving SDN Scalability with Protocol-Oblivious Source Routing: A System-Level Study

Shengru Li, Kai Han, Nirwan Ansari, *Fellow, IEEE*, Qinkun Bao, Daoyun Hu, Junjie Liu, Shui Yu, *Senior Member, IEEE*, and Zuqing Zhu, *Senior Member, IEEE*

***Abstract*—Software-defined networking (SDN) has been considered as a break-through technology for the next-generation Internet. It enables fine-grained flow control that can make networks more flexible and programmable. However, this might lead to scalability issues due to the possible flow state explosion in SDN switches. SDN-based source routing can reduce the volume of flow-tables significantly by encoding the path information into packet headers. In this paper, we leverage the protocol-oblivious forwarding instruction set (POF-FIS) to design protocol-oblivious source routing (POSR), which is a protocol-independent, bandwidth-efficient and flow-table-saving packet forwarding technique. We lay out the packet format for POSR, come up with the packet processing pipelines for realizing unicast, multicast and link failure recovery, and implement POSR in a POF-enabled SDN network system. Experiments are then performed in a network testbed, which consists of 14 stand-alone SDN switches, and to validate the advantages of POSR. Specifically, we compare POSR with several OpenFlow-based benchmarks for unicast, multicast and link failure recovery, and confirm that POSR can reduce flow-table utilization effectively, shorten path setup latency and expedite link failure recovery.**

***Index Terms*—Software-defined networking (SDN), Protocol-oblivious forwarding (POF), Source routing.**

I. Introduction

NOWADAYS, to provision emerging network applications, software-defined networking (SDN) has been proposed to make networks more programmable and application-aware, and has attracted intensive interests from both academia and industry [1, 2]. SDN decouples the control and forwarding planes of a network and leverages the centralized network control and management (NC&M) to make the network more programmable and adaptive. Hence, network innovations for new applications and services can be easily realized through the forwarding plane abstraction provided by the control plane. As one of the most popular implementations of SDN, OpenFlow [3] specifies the protocol for the communication between the control and forwarding planes. It abstracts the behaviors of SDN switches into flow-tables, with which they process packets using the "match-and-action" principle.

However, the centralized NC&M in SDN might lead to scalability issues. For instance, in a relatively large SDN network, the communication between the centralized controller and geographically-distributed switches could bear long latency, which might make the path setup time-consuming since the controller needs to install flow-tables on each switch along the routing path of a flow. Although the path setup latency might not be an issue for long-lasting elephant flows, it can degrade the quality-of-service (QoS) of latency-sensitive flows significantly [4]. In the meantime, the processing capacity of SDN switches to handle OpenFlow messages might also be an issue, since most of the commercially-available OpenFlow switches cannot process more than 500 *FlowMod* messages in one second [4]. Moreover, the "match-and-action" principle can impact the granularity of flow control due to the fact that each SDN switch can only store a limited number of flow-tables. This is because an SDN switch usually stores the flow-tables in the ternary content addressable memory (TCAM), which is expensive and power hungry [5]. In general, a switch's TCAM can only carry hundreds to thousands of flow entries [6]. Therefore, if we want to realize per-flow-based fine-grained traffic control, the controller needs to install at least one flow-entry per flow in each switch along the path used by the flow. This mechanism can use up the TCAM on switches quickly under high traffic load conditions [7].

Recently, source routing has been considered as a promising technique to improve the scalability of SDN, *i.e.*, simplifying the message exchange between the controller and switches and reducing the numbers of flow-tables in switches [8–10]. The basic idea of SDN-based source routing is to encapsulate a flow's path information into the headers of its packets at the source switch. Then, each intermediate switch along the path would only need to pop out the corresponding output port from the packet header and then forward the packet accordingly. Hence, none of the intermediate switches needs to interact with the controller during the path setup, and a small and fixed number of flow-entries can be shared by all the flows to realize per-flow based fine-grained traffic control.

Previously, people have considered to implement SDN-based source routing with OpenFlow [8–10]. Nevertheless, since the forwarding plane of OpenFlow is protocol-dependent, *i.e.*, matching fields and actions are defined based on existing network protocols, the source routing approaches designed in previous studies only have limited system flexibility and packet transmission efficiency. For instance, OpenFlow switches can only leverage the header fields of an existing protocol to encapsulate the path information, which is apparently not flexible and compromises protocol compatibility, *i.e.*, the switches may have difficulties to support the actual forwarding mechanisms

S. Li, K. Han, Q. Bao, D. Hu, J. Liu and Z. Zhu are with the School of Information Science and Technology, University of Science and Technology of China, Hefei 230027, China. E-mail: zqzhu@ieee.org.

N. Ansari is with the Advanced Networking Laboratory, Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ 07102 USA. Email: nirwan.ansari@njit.edu.

S. Yu is with the School of Information Technology, Deakin University, VIC 3125, Australia. Email: syu@deakin.edu.au.

Manuscript received on February 8, 2017.

defined in the existing protocol simultaneously. More importantly, most of these studies only used the SDN-based source routing to realize 1-to-1 unicast [11], while more sophisticated scenarios such as multicast and link failure recovery have not been considered yet. Note that multicast is frequently used in the Internet to realize services such as video delivery [12–14] and data backup [15, 16], and as SDN switches count on the controller to calculate routing paths, how to realize link failure recovery in a fast and resource-efficient way has become a critical problem in SDN. Therefore, SDN-based source routing needs to be further optimized and enhanced, which can be done by leveraging the forwarding plane programmability provided by the protocol-oblivious forwarding (POF) [17–19].

It is known that similar to the programming protocol-independent processors (P4) [20], POF tries to decouple network protocols from the forwarding processing in SDN-based switches and to make the forwarding plane reconfigurable, programmable and future-proof. More specifically, POF provides a protocol-oblivious forwarding instruction set (POF-FIS) [21] that enables system designers to define protocol stack and packet processing procedure in a much more flexible manner than what they can get from OpenFlow. Hence, with POF, we can tailor the packet design for source routing and adapt it to the actual networking scenario, and by leveraging the forwarding plane programmability provided by POF, we can address not only unicast but also multicast and link failure recovery to deliver a more comprehensive solution.

In this work, we propose protocol-oblivious source routing (POSR), which is a protocol-independent, bandwidth-efficient and flow-table-saving packet forwarding technique based on POF-FIS. We design the packet format for POSR, develop the packet processing pipelines for realizing unicast, multicast and link failure recovery, and implement POSR in a POF-enabled SDN network system. Experimental demonstrations are then performed in a network testbed, which consists of 14 stand-alone SDN switches. Our experimental results validate that as compared with OpenFlow-based benchmarks, the proposed POSR can reduce flow-table utilization effectively, shorten path setup latency and expedite link failure recovery. The contributions of this work can be summarized as follows.

- We design the packet format and packet forwarding procedure for POSR to realize flow-table-saving and bandwidth-efficient source routing. To the best of our knowledge, this is the first system work that addresses protocol-independent source routing in SDN networks.
- We extend the use cases of SDN-based source routing to multicast and fast link failure recovery with POSR, and design the corresponding packet processing procedures.
- We implement the proposed POSR schemes in a POF-enabled network system that consists of 14 stand-alone SDN switches, and conduct experimental demonstrations to verify the effectiveness of our proposals.

The rest of the paper is organized as follows. Section II introduces the backgrounds of POF and source routing, and provides a brief survey on related work. Our design of POSR is described in Section III to layout the basic packet processing principle for unicast. Then, we discuss how to realize link failure recovery and multicast with POSR in Sections V and IV, respectively. Section VI analyzes the overhead and scalability of POSR. The experimental evaluations are presented in Section VII. Finally, Section VIII summarizes the paper.

II. Background and Related Work

A. Protocol-Oblivious Forwarding (POF)

We first review the operation principle of POF and its flow instruction set (POF-FIS) briefly to provide a context for the rest of this paper. The architecture of a POF-enabled SDN network is similar to that of an OpenFlow-enabled one, *i.e.*, a centralized controller residing in the control plane manages the behaviors of switches in the forwarding plane with flow-tables. The main innovation of POF lies in its more generic packet field description scheme for flow matching and processing. Fig. 1 shows the packet processing procedure in a POF switch, with which we can explain the basic concepts of POF.

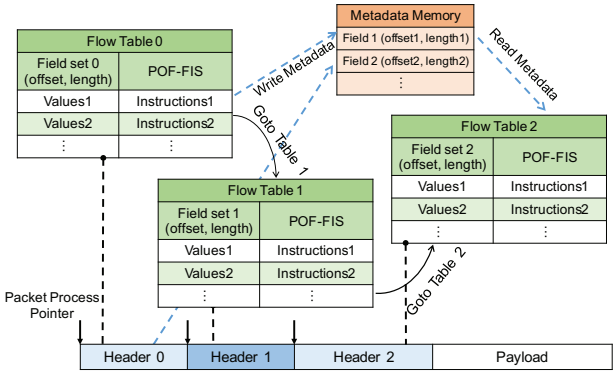


Fig. 1. Packet forwarding procedure in a POF switch.

- **Matching Fields**: As depicted in Fig. 1, POF defines the flow-table search key of a packet field in an protocol-independent way, as a tuple <*offset, length*>. Here, *offset* represents the start location of the field in a packet, while *length* indicates the field's length in bits [17]. A flow-table can contain multiple flow entries, each of which is based on <*offset, length*> tuples and specifies matching fields, its values, and the match action(s) defined with POF-FIS.
- **POF-FIS**: It defines the instructions/actions supported by POF that also utilize the tuple <*offset, length*> to locate data in a packet [22–24]. Hence, with POF-FIS, the controller can manipulate any part of a packet freely without protocol-dependent restrictions. This is much more flexible than the scheme used in OpenFlow. For instance, in OpenFlow v1.5.0 [3], *Push* actions is bound to specific protocol fields (*e.g.*, *PushMPLS*, *PushPBB*, and *PushVLAN*), while all these actions can be merged in POF-FIS as a generic *AddField*, with which we can insert any fields at any locations in a packet.
- **Flow-Tables**: Fig. 1 also shows that each POF flow-table can specify several packet fields by using multiple <*offset, length*> tuples. Hence, with the flow-tables, we can abstract the packet forwarding procedure in a switch as a data-path pipeline to achieve enhanced network programmability and flexibility. The packet processing in

such a pipeline, which consists of multiple flow-tables, is steered according to the *GotoTable* instruction defined in POF-FIS. Specifically, when a flow-table has matched to certain field(s) in a packet and executed the corresponding action(s), *GotoTable* can send the packet to the desired next flow-table as shown in Fig. 1. Here, each flow-table can contain multiple flow-entries, each of which specifies a match field, its value and the corresponding action(s).

- **Metadata Memory**: It is allocated on a POF switch to buffer the packet information that is needed when the switch processes packets. POF-FIS provides the instructions to write/read data in metadata memory based on <*offset, length*> tuples, as illustrated in Fig. 1.

B. Related Work

Technically speaking, source routing itself is not a brand-new idea. It was originally designed for the traditional IP networks to enable a source host to specify how its packets will be routed through the network [25]. This scheme, however, can be easily compromised by malicious users to instigate source address spoofing attacks, thus resulting in security breaches. For these security issues, source routing was not widely deployed in traditional IP networks that use distributed NC&M. On the other hand, since SDN-based source routing counts on the centralized controller to determine the routing paths for packet flows, the aforementioned security breaches can be avoided as long as the controller is safe and uncompromised.

Nevertheless, as OpenFlow [3] uses a protocol-dependent forwarding plane to implement SDN, existing SDN-based source routing schemes can only leverage the legacy protocol fields that are supported by OpenFlow. In [8], the authors proposed an SDN-based source routing system (*i.e.*, SwitchReduce), which uses multiple VLAN tags to encode a path in the packet header. A source routing scheme that leverages MPLS label fields was designed in [9], which encapsulates the output port of each hop in an MPLS label. Guo *et al.* [26] introduced a port-switching based source routing scheme in their datacenter network virtualization system (*i.e.*, SecondNet), which also overwrites the MPLS label fields to encode the output port sequence of a packet. As source routing pushes all the routing-related fields at the source switch and lets each intermediate switch pop the corresponding routing instruction (*i.e.*, the output port of the switch) out, the length of a routing-related field can affect the transmission efficiency of the whole system significantly. Hence, the two aforementioned schemes should be further optimized because the length of a VLAN or MPLS label field (*i.e.*, 32 bits) is too long to encode a switch output port, thus incurring a relatively large transmission overhead, especially in large-scale networks.

To avoid the unnecessary transmission overhead, one can try to squeeze more routing instructions into one legacy protocol field. For instance, Guo *et al.* [27] used the VID field in a VLAN tag to encode the output ports of four consecutive switches. However, their implementation has a restriction on each SDN switch's output port number, *i.e.*, a 12-bit VID field can encode the routing instructions of 4 hops provided that each switch cannot have more than 8 output ports. Moreover,

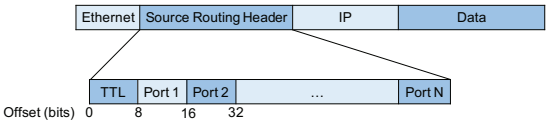


Fig. 2. Header format design of POSR packets.

the other fields (*i.e.*, 20 bits) in the VLAN tag are still not properly utilized. On the other hand, we can also reduce the transmission overhead of source routing by reusing the fields in the Ethernet header. For example, the studies in [10, 28] reused the Ethernet header to encode the path information for source routing. This source routing packet design is still not flexible and might cause compatibility issues.

Existing SDN-based source routing schemes either have relatively large transmission overhead or only provide limited flexibility and backward compatibility. More importantly, a comprehensive experimental investigation covering the use cases of SDN-based source routing for not only unicast but also multicast and link failure recovery is missing in the literature, to the best of our knowledge. These issues can be properly addressed by leveraging the protocol-independent nature and enhanced forwarding plane programmability of POF. In [29], we have presented some preliminary results on protocol-oblivious source routing (POSR), but we only designed the packet processing pipeline for unicast and did not optimize it for efficient data transmission.

III. Protocol-Oblivious Source Routing

In this section, we describe the operation principle of our proposed POSR, including the packet format design and packet processing procedure in POF switches.

A. POSR Packet Format Design

Thanks to the protocol-independent nature of POF, the packet format design of POSR does not need to reuse the packet fields of existing protocols, which were designed for other purposes (*e.g.*, VLAN and MPLS), any more. Specifically, the POSR packet format can be tailored for source routing exactly to adapt to the actual networking scenario. Fig. 2 shows the design of POSR packet header fields. We insert the source routing related header fields in between the Ethernet and IP headers. After inserting the source routing related fields, we modify the *Type* field in the Ethernet header to "0x0908" for indicating that it is a POSR packet. Various source routing related header fields are detailed as follows.

- *Time-to-live (TTL)*: It is an 8-bit field that indicates the remaining hops of the packet. Note that, in source routing, the destination switch needs to know that it is the last hop to pop out a routing instruction from the packet header. Hence, *TTL* is designed for this purpose, and its value is set at the source switch and will be decreased by one in each subsequent hop. Finally, the destination switch will remove this field by applying the *DeleteField* instruction provided by POF-FIS.
- *Port*: It is also an 8-bit field, which stores the packet's designated output port on the switch of a hop (*i.e.*, the

hop's routing instruction). In general, each POSR packet contains multiple *Port* fields to encode its routing path. Each intermediate switch always pops out the outmost *Port* field (*i.e.*, <*offset=120 bits, length=8 bits*>) and parses it to find the designated output port of the packet.

Note that, for SDN networks with various scales, the lengths of *TTL* and *Port* fields can actually be changed to minimize the transmission overhead. Specifically, the length of *TTL* should be determined based on the diameter of the network, *i.e.*, an n-bit field can support the longest path of 2^n hops, while the *Port* field should be designed according to the maximum number of output ports per switch, *i.e.*, an n-bit field can support 2^n output ports per switch. Since POF has the protocol-independent nature and provides enhanced forwarding plane programmability, the lengths of *TTL* and *Port* fields can be adjusted flexibly without any restriction.

B. Packet Processing Procedure for POSR

Fig. 3 shows the operation principle of POSR in a POF network. In this paper, when showing a network topology as in Fig. 3, we mark the ID of a switch's port with a number beside it. When the first packet of a flow arrives at the ingress POF switch, it will trigger a *PacketIn* message from the switch to the controller because no flow entry has been set up for the flow. Upon receiving the *PacketIn*, the controller calculates a path for the flow and then installs a flow entry in the ingress switch for the flow, which instructs the switch to encode the path information in the flow's packets with the POSR format.

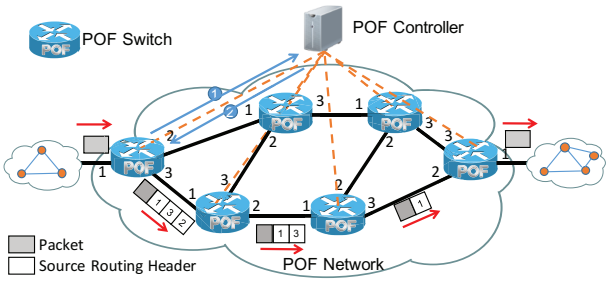


Fig. 3. Operation principle of POSR in a POF network.

Meanwhile, since for all the POSR packets, their packet processing procedure in any intermediate switches (including the destination ones) is the same, we can install the corresponding flow entries in all the POF switches during network initialization and make the POSR packets share them. *Algorithm* 1 provides the detailed processing procedure on each intermediate switch. The intermediate switches on the path pop out the outmost *Port* field, and write the field's value in its metadata memory by applying the *WriteMetadataFromPacket* instruction as shown in *Line* 4. Then, the switch checks the *TTL* field in the POSR header with the *ConditionJump* instruction, which operates as an "if-else" statement in a generic programming language. If the *TTL* field has a value that is greater than 1, the switch knows that it is not the last hop and hence only removes the outmost *Port* field and substrates the value of *TTL* by 1, as indicated in *Lines* 7-8. Otherwise, the switch removes the whole POSR header. Finally, in *Line* 15, the packet is forwarded to its designated output port, *i.e.*, the *Port* field stored in the metadata memory. This packet processing procedure carried out in intermediate switches can be shared by all the POSR flows, and the switches do not need to interact with the controller during the whole process.

Algorithm 1: Procedure for POF Switches to Forward POSR Packets

Input: Packet P arrives at Switch S

```
1  // Realized with flow-table
2  if P.Ether.Type == 0x0908 then
3      // It is a POSR packet
4      P.Metadata.Port_buffer =
       WriteMetadataFromPacket(P.Port) ;
5      if P.TTL > 1 then
6          // Not the last hop
7          DeleteField(P.Port) ;
8          P.TTL = P.TTL - 1;
9      else
10         // The last hop
11         DeleteField(P.POSR_header) ;
12         P.Ether.Type = 0x0800
13     end
14     // Send packet to designated port
15     Output(P) to P.Metadata.Port_buffer ;
16 else
17     // It is not a POSR packet
18     AddField(P.POSR_header) ;
19     Output(P) ;
20 end
```

POF-FIS greatly enhances the forwarding plane programmability of POF switches, and thus with it, we can reduce the burden of the controller and make the data-path more intelligent. We define the following notations to explain the operations with POF-FIS.

- <*offset, length*>: the field starting from the *offset* with *length*.
- {*offset, length*}: the value of field at <*offset, length*> in the packet.
- [*offset, length*]: the value of field at <*offset, length*> in the metadata memory.

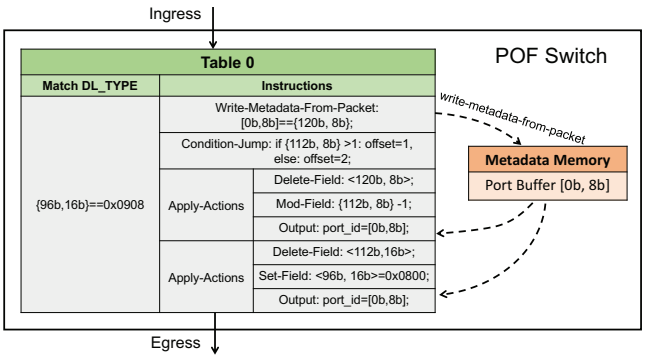


Fig. 4. Flow-table on intermediate POF switches for POSR.

Fig. 4 shows the flow-table that we design with POF-FIS to realize the procedure in *Algorithm* 1, *i.e.*, processing POSR packets in intermediate switches. When a POSR packet arrives at an intermediate switch, it is first processed by Table 0, which includes an entry to check the Ethernet *Type* field (*i.e.*, <96 bits, 16 bits>) of the packet. If the *Type* field has a value of "0x0908", which indicates that the packet is a POSR one, the switch will apply the *WriteMetadataFromPacket* instruction to write the value of the outmost *Port* field in its metadata memory. The switch then uses the *ConditionJump* instruction to check whether the value of the *TTL* field (*i.e.*, {112 bits, 8 bits}) is greater than 1. If yes, the switch executes the subsequent instructions in order, otherwise, it will jump 1 instruction ahead (*i.e.*, *offset* $= 2$) to execute the last instruction in the flow-entry, which removes the whole POSR header and restores the value of Ethernet *Type* field to its original value (*e.g.,* $0x0800$ for IPv4).

Here, the processing in a POF switch behaves like a software program. The procedure defined in the flow-entry can be considered as a function to achieve certain forwarding behavior(s), whose inputs and outputs are the packet fields and the processed packet, respectively. The metadata memory is the storage space for variables used in the function, which buffers the packet information temporarily. Hence, the intermediate POF switches can process the POSR packets according to a predefined generic flow-table, and save the overhead to communicate with the controller during the flow setup. More importantly, Fig. 4 indicates that with POSR, the number of flow-entries in each intermediate switch becomes 1 and would not increase with the number of packet flows or the number of switch output ports. Therefore, in contrast to the OpenFlow-based source routing schemes [8–10], our proposed POSR introduces less communication overhead between the controller and switches and further reduces the number of flow-entries used in the network. As we will show later in the experimental demonstrations, these advantages help to improve the scalability of SDN effectively.

IV. Fast Link Failure Recovery with POSR

Note that, SDN switches rely on the controller to calculate routing paths, and hence without specific design, they cannot automatically recover from link failures as IP routers do. Therefore, how to achieve link failure recovery [30] in a fast and resource-efficient way is a must-solve problem in SDN [31]. Fortunately, with simple extensions, POSR can handle link failures fast and effectively, and all the failure recovery operations are conducted on related POF switches locally without the need to interact with the controller.

We implement the fast failover (FF) group table [3] in POF switches to monitor the status of switch ports. If a port-down event is detected, the related switch (*i.e.*, the upstream switch of the broken link) will conduct POSR-based link failure recovery. Specifically, the switch will use the routing instructions of the backup path segment to replace that of the broken link in the headers of all the affected POSR packets. Fig. 5 provides an intuitive example on POSR-based fast link failure recovery. During network initialization, the controller calculates a backup path for each link[1], encodes the backup path in POSR *Port* fields for an FF group table, and installs the FF group table with correct recovery actions in the source switch of the link. Fig. 5 illustrates the above procedure, in which $S1$ encodes path $S2{\rightarrow}S3{\rightarrow}S4{\rightarrow}S5$ in the headers of the POSR packets from $S1$ to $S5$. When a link failure brings down link $S3{\rightarrow}S4$ (*i.e.*, *Port* 3 of $S3$ encounters a port-down event), its upstream switch $S3$ detects the failure. Then, $S3$ uses the routing instructions of $S3{\rightarrow}S6{\rightarrow}S7{\rightarrow}S4$ to replace that of $S3{\rightarrow}S4$ in the POSR headers to redirect all the POSR packets that originally go to *Port* 3 to *Port* 1.

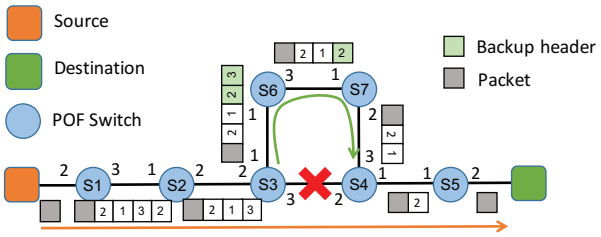


Fig. 5. Example on POSR-based fast link failure recovery.

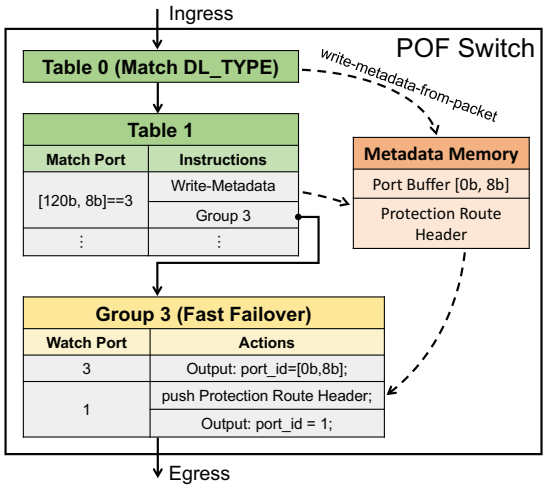


Fig. 6. Flow-tables on switch $S3$ for fast link failure recovery.

To realize the aforementioned POSR-based link failure recovery, we design the flow-tables as shown in Fig. 6 and implement them in the related POF switches. Here, Table 0 is still the same as the one in Fig. 4, which is used for normal POSR packet processing. After being processed by Table 0, a packet goes to Table 1, which determines the backup path segment based on the packet's output port, writes the *Protection Route Header* that represents the backup path segment in metadata memory, and sends the packet to the FF group table that corresponds to its output port. The number of FF group tables is equal to the number of output ports in a switch. Each FF group table includes two entries. The first one is for forwarding packets as normal when the output port is

[1]Considering the facts that link failures might not happen frequently in a well-maintained network and not all the link failures should be addressed with fast recovery, we may only need to calculate the backup paths for a few critical links in a practical situation.

up and running, and the second one takes care of the situation in which the port is down, *i.e.*, the *Protection Route Header* stored in metadata memory should be used.

V. Multicast with POSR

In addition to unicast [32], multicast is also a frequently-used communication scheme in today's Internet, which can realize services such as video conferencing and data backup with high data transmission efficiency [33]. In this section, we discuss how to realize efficient multicast with POSR.

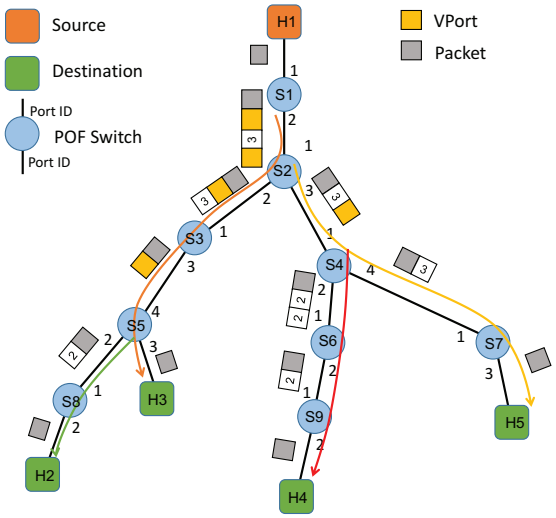


Fig. 7. Example on POSR-based multicast.

The major challenge of realizing multicast with source routing is that it is difficult to encode a whole multicast tree in a packet header. In order to address this issue, we design our POSR-based multicast to take a recursive approach.

***Definition** 1:* The **primary path** of a multicast tree is its shortest branch (in terms of hop-count), which connects the source switch to one of the destination switches.

***Definition** 2:* A **fork node** on a multicast tree is a switch from which multiple branches originates.

After obtaining a multicast tree, we first find its primary path. Here, if the tree contains more than one shortest branches, we randomly select one as the primary path. Then, on the primary path, we treat each fork node as the source switch of a subtree and find the primary path in the subtree accordingly. This procedure is repeated recursively until all the destination switches are connected with primary paths. Therefore, we basically partition the multicast tree into a few non-overlapping branches, which can be leveraged to realize POSR-based multicast. Fig. 7 shows an example on the tree-partition procedure discussed above, which indicates that the multicast tree is transformed into four primary paths, each of which ends at a destination switch.

To realize POSR-based multicast, we design a *VPort* field to replace the *Port* field in the POSR header, as shown in Fig. 8. The length of this field is determined by its two sub-fields, which help to realize the multicast operation on a fork node and are defined as follows.

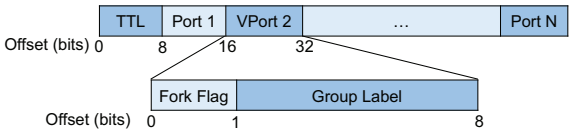


Fig. 8. Header format design of POSR multicast packets.

- *Fork Flag*: It is the first bit in a *VPort* field, which takes 0 if the corresponding switch is not a fork node on the packet's multicast tree, and 1 otherwise.
- *Group Label*: It covers the remaining bits in a *VPort* field. We assign a *Group Label* to each active multicast session, and thus this sub-field can be used to identify the packet's multicast operation at a fork node, *i.e.*, forwarding the packet to multiple output ports and encoding a new POSR header on it if necessary.

Note that the length of the *Group Label* sub-field should be determined based on the maximum number of concurrent multicast sessions in the network. For instance, a 1-Byte *VPort* field contains a 7-bit *Group Label* sub-field, which can support up to 128 concurrent multicast sessions.

Fig. 7 shows an intuitive example on POSR-based multicast. When a packet arrives at the source switch $S1$, $S1$ encodes the primary path of the multicast tree originating from $S1$ onto it, which is $S2{\rightarrow}S3{\rightarrow}S5{\rightarrow}H3$. Meanwhile, the *VPort* fields for $S2$ and $S5$ are encoded with their *Fork Flag* sub-fields turned on, for identifying the switches as fork nodes. Then, when the packet reaches $S2$, it is duplicated to two copies. One of them is forwarded to $S3$ with the packet's original POSR header, while the other one is encapsulated with a new POSR header to indicate the primary path of the subtree originating from $S2$, *i.e.*, $S4{\rightarrow}S7{\rightarrow}H5$. The forwarding procedure performed at $S5$ and $S4$ is similar to that at $S2$.

Algorithm 2 presents the procedure of POSR-based multicast on a POF switch. If the switch is not a fork node for the packet, the processing procedure is similar to that of the POSR-based unicast discussed in Section III. However, if the *Fork Flag* sub-field in the outmost *VPort* field equals 1, the packet is handled as on a fork node. Specifically, it is sent to the *Group Label* matching table, where the designated output ports are first determined and then for each output port, if it is not for the original primary path but starts a new subtree, a new POSR header is assembled in metadata memory for it by leveraging the *WriteMetadata* instruction. Then, the new POSR headers are encapsulated onto the packet's copies according to their output ports, *i.e.*, when applying the *Output* action to them. Fig. 9 shows the flow-tables for realizing the aforementioned procedure in POF switches. We use three stage flow-tables that leverage two types of POF flow-tables (*i.e.,* the masked-match and direct tables) to realize POSR-based multicast.Here, Tables 0 and 2 are mask-match tables, while Table 1 is a direct table that only contains instructions. The processing in Table 0 corresponds to *Lines* 2 to 12 in *Algorithm* 2, which is for forwarding the packet in the POSR manner, similar to the case in the unicast scheme. Moreover, the packet will also be processed by Table 1, which checks whether the current switch is a fork node by examining the *Fork Flag* of

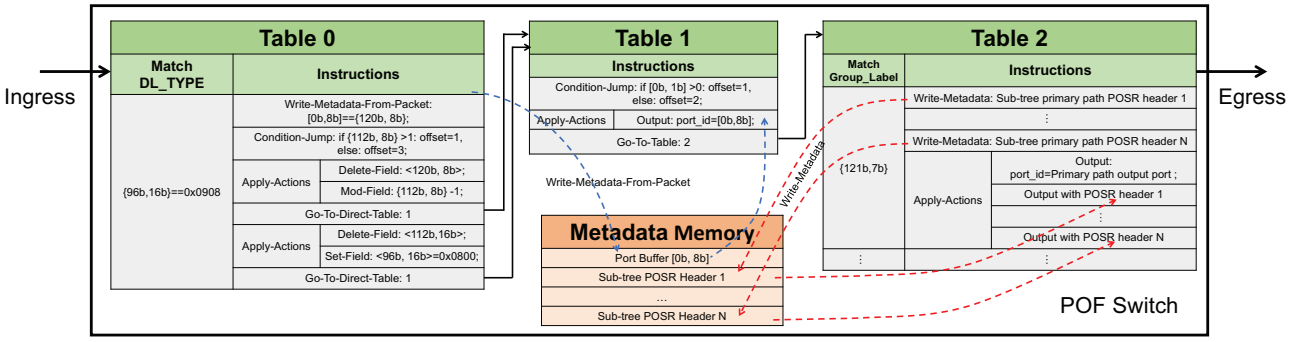


Fig. 9. Flow-tables on intermediate POF switches for POSR-based multicast.

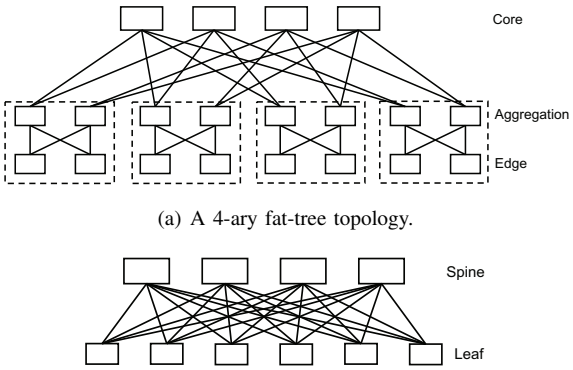
Algorithm 2: Procedure for POF Switches to Realize POSR-based Multicast

Input: Packet P arrives at Switch S

```
// Realized with flow-tables
if P.Ether.Type == 0x0908 then
    // It is a POSR packet
    P.Metadata.Port_buffer = WriteMetadataFromPacket(P.Port) ;
    if P.TTL ≠ 1 then
        // Not the last hop
        DeleteField(P.Port) ;
        P.TTL = P.TTL - 1;
    else
        // The last hop
        DeleteField(P.POSR_header) ;
    end
    if P.Metadata.Port_buffer.Fork_Flag == 1 then
        // Switch is a fork node
        if P matches S.Flow_Table.Group_Label then
            for each subtree rooted from S do
                WriteMetadata(POSR_header of the
                primary path of each subtree) ;
                Output(P) with POSR_header in
                metadata;
            end
        end
    else
        // Switch is not the fork node,
           send packet to designated port
        Output(P) to P.Metadata.Port_buffer ;
    end
end
```

the buffered *Port/VPort* field. If not, the switch will just output the packet. Otherwise, if the switch is a fork node, the packet will be sent to Table 2, which encapsulates the primary path of the current sub-tree in the POSR header according to the *Group Label* and then outputs the packet.

Note that since our POSR scheme encodes the routing path(s) of a unicast or multicast session as a shim header, the sizes of the resulting POSR packets have to be monitored carefully to avoid violating the maximum transmission unit (MTU) of the switches in the network. One way to achieve this is to let the operator set the MTU of each switch in its network to be slightly longer than the size of a standard Ethernet frame (*i.e.*, 1500 Bytes). Also, in this work, we focus on designing the packet forwarding mechanisms to realize POSR-based unicast, fast link failure recovery, and multicast, while the proposed POSR system should be able to work with any algorithms for calculating the unicast paths, backup paths, and multicast trees, as long as they can be implemented in a time efficient manner in the POF controller.



(a) A 4-ary fat-tree topology.

(b) A leaf-spine topology with $s = 4$ and $l = 6$.

Fig. 10. Examples on datecenter network topologies.

VI. Performance Analysis

In this section, we use theoretical analysis to evaluate the proposed POSR in terms of flow-table size and transmission overhead, and compare it with some existing benchmarks.

A. Flow-Table Size

In datacenter networks, topologies such as fat-tree and leaf-spine are widely used [34]. Therefore, we first use them to analyze the performance of POSR in terms of flow-table size. It is known that for a k-ary fat-tree topology, the node degree of each core/aggregation switch is k, the switches are grouped into k pods, and each pod consists of $\frac{k}{2}$ aggregation switches and $\frac{k}{2}$ edge switches. Hence, there are $\frac{k^2}{2}$ edge switches in total. Fig. 10(a) shows a fat-tree topology with $k = 4$. Hence,

TABLE I
MAXIMUM FLOW ENTRIES USED IN DATACENTER NETWORKS

	Fat-Tree Topology (k)	Leaf-Spine Topology (l, s)
Traditional OpenFlow	$\frac{5\cdot k^4}{8}\cdot(\frac{k^2}{2}-1)$	$3\cdot l\cdot(l-1)\cdot s$
Existing SDN-based Source Routing	$\frac{k^4}{8}\cdot(\frac{k^2}{2}-1)+\frac{3\cdot k^3}{4}$	$l^2\cdot s$
POSR	$\frac{k^4}{8}\cdot(\frac{k^2}{2}-1)+\frac{3\cdot k^2}{4}$	$l\cdot(l-1)\cdot s+s$

the number of edge switch pairs is $\frac{k^2}{2}\cdot(\frac{k^2}{2}-1)$. Meanwhile, for each edge switch pair, there are $\frac{k^2}{4}$ alternative paths. Hence, in the worst case, we need to support $\frac{k^4}{8}\cdot(\frac{k^2}{2}-1)$ paths simultaneously in the network. Considering the fact that the length of each path in a fat-tree is 5 hops, the maximum number of flow entries used in the network by using the normal per-hop configuration with OpenFlow would be

$$\frac{5\cdot k^4}{8}\cdot(\frac{k^2}{2}-1). \tag{1}$$

On the other hand, since the existing SDN-based source routing schemes (*e.g.,* SecondNet [26], SlickFlow [28], Path Switching [10], and SwitchReduce [8]) use port matching to reduce flow entries, the number of required flow entries equals the total number of switch ports in the network in the worst case scenario. Hence, each switch uses at most k flow entries to process source routing packets. Meanwhile, a fat-tree topology consists of $\frac{3\cdot k^2}{4}$ non-edge switches [35], and thus, when using these existing SDN-based source routing schemes, the maximum number of flow entries used in the network would be

$$\frac{k^4}{8}\cdot(\frac{k^2}{2}-1)+\frac{3\cdot k^3}{4}. \tag{2}$$

With our POSR, each intermediate switch only needs one flow entry to process POSR packets, and the maximum number of flow entries used in the network would be

$$\frac{k^4}{8}\cdot(\frac{k^2}{2}-1)+\frac{3\cdot k^2}{4}. \tag{3}$$

In a leaf-spine topology, there are s spine switches and l leaf switches. Hence, the number of leaf switch pairs is $l\cdot(l-1)$, and for each pair, there are s alternative paths. Hence, in the worst case, we need to support $l\cdot(l-1)\cdot s$ paths simultaneously in the network. Fig. 10(b) shows a leaf-spine topology with $s=4$ and $l=6$. The length of each path between a leaf switch pair is 3. When using the normal per-hop configuration with OpenFlow, the maximum number of flow entries used in the network would be

$$3\cdot l\cdot(l-1)\cdot s. \tag{4}$$

Similarly, we can obtain the maximum numbers of flow entries used in the network as

$$l^2\cdot s \tag{5}$$

and

$$l\cdot(l-1)\cdot s+s, \tag{6}$$

for the existing SDN-based source routing schemes and our POSR, respectively. Table I summarizes the results. We observe that in terms of the maximum flow entries used in the datacenter networks, POSR only slightly outperforms the existing SDN-based source routing because the most significant entries in Table I are the same. Actually, the most significant advantage of POSR over the existing SDN-based source routing is that POSR makes the source routing packet design very flexible. while keeping the maximum flow entries compatible with other algorithms. Specifically, the existing SDN-based source routing schemes can only leverage the legacy protocol fields supported by OpenFlow in their packet designs, while POSR removes this restriction by using a protocol-independent forwarding plane. The direct benefit brought by this feature is that the transmission overhead incurred by the source routing packet headers can be reduced significantly, which will be analyzed in the next subsection.

In addition to the structural topologies that are normally used in datacenter networks, we also analyze the flow-table size in a non-structural topology that is for a wide-area network (WAN), *i.e.*, the 14-node NSFNET topology in Fig. 11 [36]. Here, if we calculate 5 shortest paths between each node pair, we totally get 910 paths with a mean length of 5 hops. Therefore, our POSR requires 910 flow entries in the network in the worst case because it only installs flow entries in source switches. On the other hand, when using the normal per-hop configuration with OpenFlow, the maximum number of flow entries used in the network would be 12740.

B. Transmission Overhead

As source routing encodes path information into packet headers, additional transmission overhead will be incurred. POSR can set the length of the *Port* field as $\lceil\log_2(d)\rceil$ bits, if the maximum node degree of a network is d. Then, if we assume the diameter of the network is h (*i.e.*, the longest path has h hops), the size of the longest POSR packet header would be $(h\cdot\lceil\log_2(d)\rceil+8)$ bits since there is an 8-bit *TTL* field in each header too. On the other hand, the existing SDN-based source routing schemes usually utilize the MPLS or VLAN header to encode the path information. Specifically, the output port on each hop is encapsulated in either the *Label* field of MPLS header or the *VID* field of VLAN header. Nevertheless, as both MPLS and VLAN headers are 32-bit long, the resulting overhead would be much more significant. For instance, we use a real traffic trace captured on an OC-192 link at Chicago [37] (*i.e.*, with an average data-rate of 2.31 Gbps and a mean throughput of 4.75×10^5 packets/s) to conduct a simple simulation, and find that for an average path length of 5 hops, the existing schemes that use MPLS/VLAN headers can have an average bandwidth overhead of 75 Mbps, while POSR can reduce the overhead down to 23 Mbps.

VII. Experimental Demonstration

In this section, we discuss the experimental setup and results to demonstrate the performance of our proposed POSR.

A. System Implementation

We build a POF network testbed to verify the functionality and performance of our proposed POSR. The testbed consists of 14 software-based POF switches, each of which runs on a stand-alone high-performance Linux server (*i.e.*, Lenovo RD540 server equipped with an Intel Ethernet Gigabit server adapter I340-T4 that uses Intel 82580 Ethernet controller). The software-based POF switch was originally developed in the open-source POFSwitch project initiated by Huawei [38], and we took over the development task to extend its functionality and improve the forwarding performance [39]. Similar to the well-known OpenvSwitch [40] for OpenFlow, our POFSwitch can run on a general-purpose server and realize high-throughput packet forwarding based on POF. Each POF switch is equipped with the network interfaces based on 1GbE, and is locally connected to a host that is realized by a virtual machine to generate traffic. We have verified that when each packet has a size of 64 Bytes (*i.e.*, the smallest packet size), our POFSwitch can still achieve 1 Gbps forwarding rate per port for POSR packets, which corresponds to a packet forwarding throughput of around 1.95 million packets/s. The topology of the testbed is shown in Fig. 11, and Fig. 13 illustrates the actual equipment in the setup. The home-made POF controller [39] also runs on a Linux server, and we implement it by extending the POX platform [41].

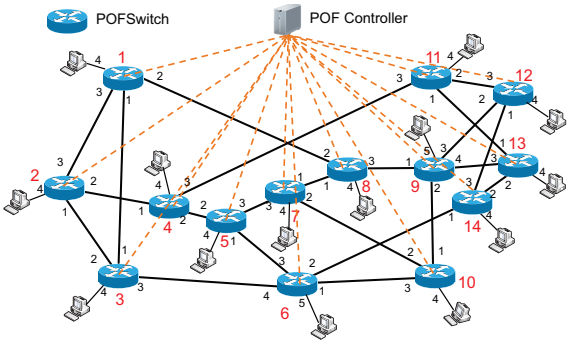


Fig. 11. Experimental topology.

Fig. 12 shows the architecture of our software-based POF switch (*i.e.*, POFSwitch [39]) and also explains its position in the operating system. Specifically, POFSwitch runs in the Linux user space, and the Intel data plane development kit (DPDK)[2] [42] is leveraged to realize fast packet processing in POFSwitch. Note that, DPDK provides a set of data plane libraries for Intel network adapters to accelerate the packet processing in X86 platforms. It uses poll-mode to handle packets and avoids the cost of context switching in the traditional interrupt-mode. More specifically, the packets handled by DPDK are processed in the Linux user space and attached to one CPU core in its whole lifetime in POFSwitch; this helps to avoid the memory copies between the kernel and user spaces as well as frequent switches among CPU cores. With these benefits, we improve the forwarding performance of POFSwitch by leveraging DPDK. Here, with the DPDK driver, packets from the network interface cards (NICs) bypass the Linux kernel space and enter the data-path of the POF switch directly, to accelerate packet processing and forwarding. The switch control module manages the POF switch. Specifically, it installs flow-tables and flow entries according to the parsing results from the POF protocol stack and updates resource usage status in the switch resource database.

[2]Here, we use DPDK version 2.2.0.

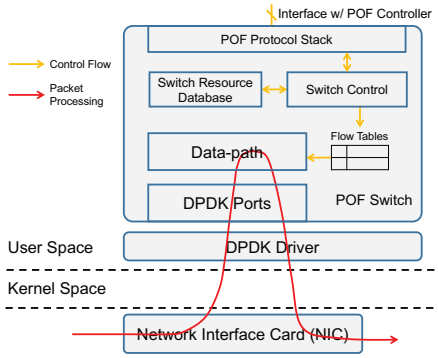


Fig. 12. Architecture of software-based POF switch (*i.e.*, POFSwitch [39]).

Fig. 13 explains the functional designs in POF controller and switches to realize POSR. The controller parses the POF messages from switches with the POF protocol stack, which defines the message formats. Then, the parsing results are sent to the POSR manager for further processing. The POSR manager provides the north-bound APIs to implement POSR-based network applications and it also instructs the POF protocol stack to encode related messages. In this work, we develop the multicast handler and protection scheme installer modules based on the north-bound APIs to realize POSR-based multicast and link failure recovery, respectively, while the unicast handler is leveraged from our previous work [29] and is optimized to enhance efficiency.

During network initialization, the protection scheme installer calculates the backup paths and installs the corresponding FF group tables to related POF switches. When the network is operational, the unicast/multicast handler is responsible for installing the flow-tables and flow entries that we designed in Figs. 4 and 9 for unicast/multicast services. Specifically, the service requests are encoded in *PacketIn* messages by the switches. Each *PacketIn* message encodes the first packet of a flow, and when it has been parsed by the POF protocol stack and a corresponding service request is initialized in the POSR manager. The POSR manager classifies the service as unicast or multicast by checking its destination IP address. Next, if it is a unicast request, the unicast handler calculates the shortest routing path and determines the output port on each switch along the path in sequence. Then, the APIs in the POSR manager are invoked to build flow entries based on the request's original information and the calculated output port sequence. Otherwise, for a multicast request, the multicast handler first finds its multicast group and calculates a multicast

tree (MST) to cover all the group members. It then generates the output port sequence for each primary path rooted from a fork node and invokes the APIs in the POSR manager to install the flow entries to related POF switches. Since the POSR-based multicast does not specify the algorithm to calculate the multicast trees, we leave the choice of the algorithms to the ISPs and let them implement whatever they need.

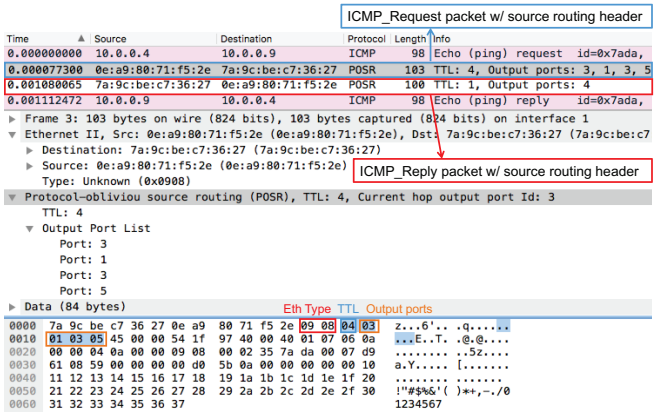
Each POF switch also has the POF protocol stack to communicate with the controller for receiving flow-tables and flow entries. When a packet arrives at the switch, it is first processed by the packet classification table to determine the packet type. If it is a POSR packet, the service classification table will find whether it is for unicast or multicast. Then, the corresponding service pipeline is applied as discussed in Sections III and V. Otherwise, if it is an IP packet, the switch sends it to the table for POSR header encapsulation, where the packet is converted into the POSR format by matching to the flow entries installed for it. After having been processed by these pipelines, the packet is sent to the FF group table for executing the output action. We run a thread in each switch to monitor the status of each output port, and when a link failure is detected, it notifies the FF group tables to update the related group entries for failure recovery immediately.

B. Experiment for Function Verification

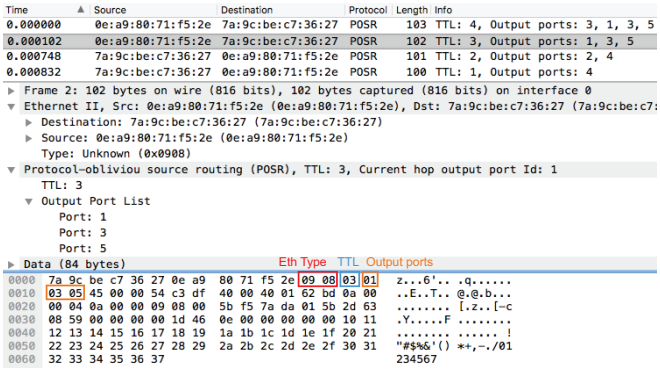
To verify the functionality of POSR, we send *ICMP_Request* packets from the host attached to Node 4 to the one attached to Node 9. When the first *ICMP_Request* packet arrives at Node 4, the switch finds that there is no flow entry to match against. Then, Node 4 encapsulates this packet into a *PacketIn* message and sends it to the controller. The controller calculates the path for the packet as 4→5→7→8→9, and instructs the switch on Node 4 to insert a POSR header into the packet to carry the path information (*i.e.*, the output port on each switch along the path). Next, each intermediate switch only needs to forward the packet according to the corresponding output port stored in its header.

Fig. 14(a) shows the packets captured at Ports 4 and 2 on Node 4. Here, we record four packets among which the first and fourth ones are captured at Port 4 while the remaining two are captured at Port 2. We can see that the first packet is a 98-Byte *ICMP_Request* packet entering from Port 4, and when it leaves the switch at Port 2 (*i.e.*, the second packet), it is converted to a POSR-based one whose *Eth_Type* field is set to "0x0908". Also, the packet length increases from 98 Bytes to 103 Bytes because a POSR header has been encapsulated into the packet. The *TTL* field is set to 4 and the *Port* fields are encoded with the output ports of the intermediate switches along the path. The third and fourth packets in Fig. 14(a) are for the *ICMP_Reply* packet traveling in the opposite direction. We observe that the packet has been restored to the original format at Port 4, by looking at the *Eth_type* of the fourth packet. This verifies that the POSR encapsulated packet can be restored to the original packet at the egress node.

Fig. 14(b) illustrates two packets captured at Ports 2 and 3, respectively, on Node 5, which is an intermediate switch. We find that at the output of Node 5 (*i.e.*, Port 3), the second



(a) ICMP packets captured on Node 4.



(b) ICMP packets captured on Node 5.

Fig. 14. Wireshark captures to verify POSR functionality.

packet has one *Port* field popped out and its *TTL* field's value becomes 3. These results suggest that the packet processing pipeline in Fig. 4 is correctly implemented in intermediate switches to forward POSR packets.

C. Experiments for Performance Evaluation

1) Unicast Scenario: For the unicast scenario, we design experiments to perform stress tests on SDN switches while restricting the maximum number of flow entries on each switch. Specifically, an experiment simultaneously generates 400 UDP flows, each of which has a throughput of 0.25 Mbps and a random destination, on each host with iPerf [43], and compares POSR with a traditional OpenFlow-based scheme that uses shortest path routing (OF-SP)[3]. Fig. 15 shows the experimental results on the percentage of throughput that can be successfully received at destinations, *i.e.*, the receiving throughput, which indicate that the flow entry capacity of each switch does impact the receiving throughput of both schemes.

Fig. 15 indicates that with POSR, we can achieve a receiving throughput of 100% with only less than 1000 flow entries per switch. This is because POSR makes the flows share flow entries on intermediate switches, while only needs to install

[3]Note that, to guarantee a comparable comparison, we configure our POF-Switch as an OpenFlow switch because POFSwitch is backward-compatible and can be configured to organize the flow-tables as defined in the OpenFlow specification. Also, we have verified that by doing so, POFSwitch functions as an OpenFlow switch well and does not bring any unfair drawbacks to the traditional OpenFlow-based scheme.

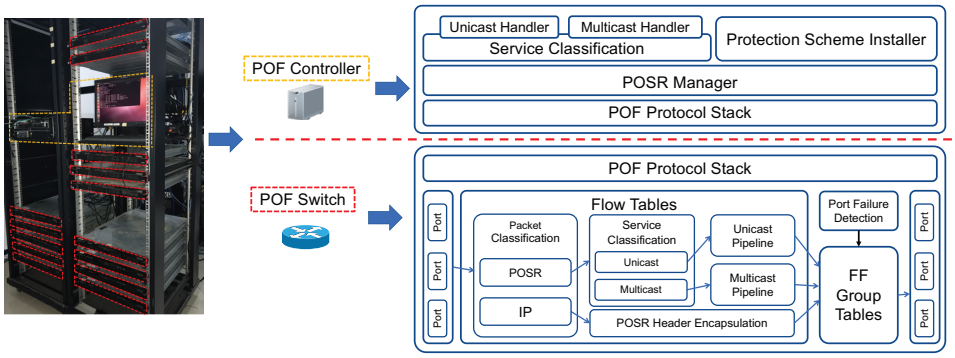


Fig. 13. Architecture of POSR system.

per-flow-based flow entries on ingress switches. However, since OF-SP needs to install per-flow-based flow entries on every hop along the forwarding paths, each switch requires many more flow entries to forward the packets correctly. More specifically, the receiving throughput of OF-SP would not reach 100% until each switch is allocated more than 2200 flow entries. This is because when the capacity of flow entries on each switch is insufficient, the flow entries of existing flows would be overwritten before they actually expire and this would make the switches send *PacketIn* messages to the controller very frequently, thus restricting the receiving throughput. The experimental results verify that for flow-level traffic management, POSR utilizes flow entries much more efficiently than the traditional OpenFlow-based scheme. Since each host generates 400 UDP flows and there are 14 hosts, there are 5600 flows in the network. However, the flow-entry utilization among the switches might not be uniform, and for OF-SP, the switches that are located at the center of the topology would consume more flow-entries than others. This is the reason why the results in Fig. 15 indicate that when 2200 flow-entries are allocated for each switch, OF-SP can reach a receiving throughput of 100%, *i.e.*, the requirement of the most loaded switch can be satisfied in this case.

Note that, the experiments restrict the number of flow entries per switch below 2200, which could be less than the actual T-CAM capacity of a practical hardware SDN switch. Nevertheless, our experiments use software-based POF switches with 1GbE NICs, which make their traffic throughput significantly smaller than that of practical hardware SDN switches, which are usually equipped with 10GbE NICs. Therefore, since the traffic throughput has been scaled down, the setting on the flow entry capacity per switch would be reasonable and fair.

Moreover, since POSR only needs to install flow entry on the first switch along a routing path, it saves a lot of communications between the controller and switches and thus would reduce the path setup latency effectively. To verify this, we measure the path setup latency of $4{\rightarrow}5{\rightarrow}7{\rightarrow}8{\rightarrow}9$ under different traffic loads (*i.e.*, different loads of *PacketIn* messages to the controller), and plot the results in Fig. 16. We observe that POSR does achieve much shorter path setup latency than OF-SP, especially when the traffic load is higher than 5100 flows/sec. The reason behind this appears to be two-fold. Firstly, to setup a path, OF-SP has to install flow entries

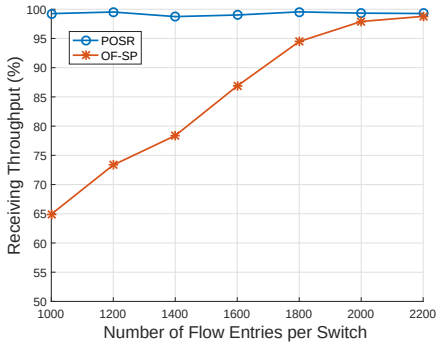


Fig. 15. Experimental results on receiving throughput of unicast.

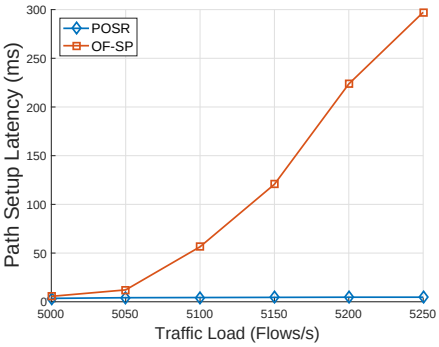


Fig. 16. Experimental results on path setup latency.

on all the switches along it, which results in significantly higher north-/south-bound communication loads than the case in POSR. This can actually congest the controller and result in long message processing time there. Secondly, processing the *FlowMod* messages from the controller and installing the corresponding flow entries on each switch also takes time.

2) Fast Link Failure Recovery: We then evaluate the performance of POSR on fast link failure recovery. We set up the working path as $4{\rightarrow}5{\rightarrow}7{\rightarrow}8{\rightarrow}9$ and transfer traffic with different data-rates over it. The experiments compare the link failure recovery with POSR and OpenFlow. Specifically, POSR allocates a backup path segment for each link along the working path, *e.g.*, using segment $5{\rightarrow}6{\rightarrow}10{\rightarrow}7$ to protect link $5{\rightarrow}7$, while the OpenFlow-based scheme assigns a link-

disjoint backup path (*e.g.*, using backup path $4\rightarrow11\rightarrow13\rightarrow9$ to protect working path $4\rightarrow5\rightarrow7\rightarrow8\rightarrow9$) for link failure recovery. Note that, to ensure fairness, the experiments compare the OpenFlow and POSR based schemes under the assumption that the numbers of backup flow-tables used by them are the same. Hence, the OpenFlow scheme has to use path protection because link protection would make it consume many more backup flow-tables. Here, as the backup path consists of four switches, the OpenFlow scheme pre-installs four backup flow-tables in them. Meanwhile, since POSR needs to pre-install an additional FF group table for each protected link and the working path consists of four links, four backup flow-tables are pre-installed too. Therefore, we can realize an apple-to-apple comparison in this way. Note that since an FF group table does not contain match fields, it actually occupies less switch memory than an OpenFlow-based backup flow-table. Then, in the experiments, we interrupt link $5\rightarrow7$ randomly and measure the packet loss due to the link failure. The traffic over the working path is generated by iPerf and the packet length is fixed as 1250 Bytes.

Fig. 17 shows the experimental results on the number of packet losses. We can see that the OpenFlow-based scheme not only induces many more packet losses than the POSR-based one but also increases the number of packet losses faster with the transfer data-rate. This is because the OpenFlow-based scheme makes the switches interact with the controller during the link failure recovery and hence prolongs the recovery latency. While with our POSR-based scheme, the upstream switch (*i.e.*, the one in Node 4) can directly switch the traffic flow to its backup path after detecting the link failure, and there is no need to interact with the controller in the process. The above analysis can be verified by the results in Table II, which indicate that the average failure recovery time of POSR is shorter. Note that, in the network system, the recovery procedure would only be invoked when the link failure has been detected by the operating system (OS) of an affected switch. Hence, the failure recovery time includes the time used for the failure detection in the OS, which is also shown in Table II. We observe that the failure detection actually takes most of the failure recovery time. Therefore, if we exclude it, the advantage of POSR would become much more significant. We also hope to point out that the advantage of POSR on fast link failure recovery would become even more significant in a wide-area network, since the messaging delay between the controller and switches will be much longer than that in the experimental testbed.

TABLE II
EXPERIMENTAL RESULTS ON AVERAGE FAILURE RECOVERY TIME

Average Failure Recovery Time of POSR (msec)	70.38
Average Failure Recovery Time of OpenFlow (msec)	85.36
Failure Detection in OS (msec)	54.13

3) Multicast Scenario: Finally, we evaluate the performance of POSR-based multicast. The experiments simultaneously generate 100 multicast sessions on each host, and each multicast session tries to deliver 0.25 Mbps UDP traffic

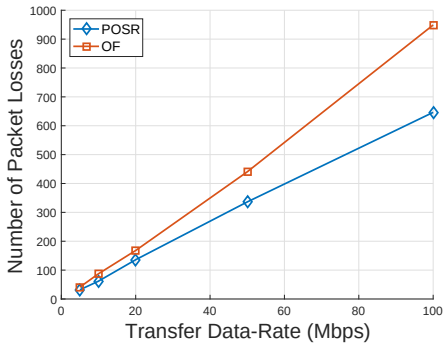


Fig. 17. Experimental results of packet loss ratio for single link failure.

to 2 to 3 destinations randomly. Here, we still compare the results on receiving capacity while restricting the maximum number of flow entries on each switch. Note that, since POSR-based multicast scheme needs to install different types of flow-tables on the source and fork nodes on a multicast tree, *i.e.*, a flow match table on the source node and a *Group_Label* match table on each fork node, we assume that on each POF switch, the capacities of flow match and *Group_Label* match tables are equal. The benchmark scheme is the OpenFlow-based multicast (OF-MST). In the experiments, both schemes use the minimum Steiner tree (MST) algorithm in [44] to calculate the multicast trees. Fig. 18 shows the experimental results on receiving throughput. It can be seen that similar to the unicast scenario, our POSR-based scheme can achieve a receiving throughput of 100% with much less flow entry utilization. Hence, the experimental results confirm that our POSR-based scheme uses flow entries much more efficiently not only for unicast but also for multicast.

VIII. CONCLUSION

In this paper, we leveraged POF-FIS to design protocol-oblivious source routing (POSR), which can realize SDN-based packet forwarding in a protocol-independent, bandwidth-efficient and flow-table-saving packet forwarding manner. We designed the packet format for POSR, formulated the packet processing pipelines for realizing unicast, multicast and link failure recovery, and implemented POSR in a POF-enabled SDN network system. Then, we built a network testbed that included 14 SDN switches and demonstrated the advantages of POSR experimentally. Specifically, we compared POSR with several OpenFlow-based benchmarks for unicast, multicast and link failure recovery, and verified that POSR can reduce flow-table utilization effectively, shorten path setup latency and expedite link failure recovery.

We will consider further research from two perspectives. First, we would like to study how to provide differentiated services with POSR because its current design would simply process all the traffic flows in the same way in intermediate switches, no matter what kinds of network services are provisioned by them. Second, we would like to improve the performance of POFSwitch such that POFSwitch could process packets more efficiently and work smoothly for switches equipped with 10GbE NICs.

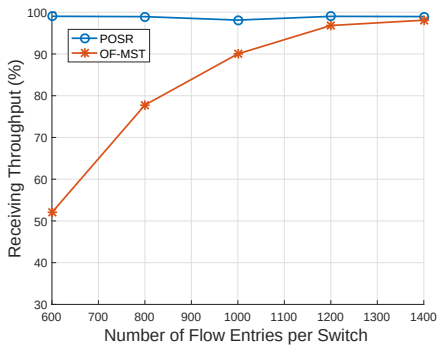


Fig. 18. Experimental results of throughput for multicast.

ACKNOWLEDGMENTS

This work was supported in part by the NSFC Project 61371117, the Key Project of the CAS (QYZDY-SSW-JSC003), the NGBWMCN Key Project under Grant No. 2017ZX03001019-004, and the Strategic Priority Research Program of the CAS (XDA06011202).

REFERENCES

[1] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, pp. 14–76, Jan. 2015.
[2] P. Lu *et al.*, "Highly-efficient data migration and backup for big data applications in elastic optical inter-datacenter networks," *IEEE Netw.*, vol. 29, pp. 36–42, Sept./Oct. 2015.
[3] OpenFlow Switch Specifications. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf
[4] S. Shirali-Shahreza and Y. Ganjali, "ReWiFlow: Restricted wildcard OpenFlow rules," *Comput. Commun. Rev.*, no. 45, pp. 29–35, Sept. 2015.
[5] H. Huang *et al.*, "Cost minimization for rule caching in software defined networking," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 4, pp. 1007–1016, Apr. 2016.
[6] S. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On scalability of software-defined networking," *IEEE Commun. Mag.*, vol. 51, pp. 136–141, Feb. 2013.
[7] M. Rifai, D. Lopez-Pacheco, and G. Urvoy-Keller, "Coarse-grained scheduling with software-defined networking switches," in *Proc. of SIGCOMM 2015*, pp. 95–96, Aug. 2015.
[8] A. Iyer, V. Mann, and N. Samineni, "SwitchReduce: Reducing switch state and controller involvement in OpenFlow networks," in *Proc. of NETWORKING 2013*, May. 2013.
[9] S. Jyothi, M. Dong, and P. Godfrey, "Towards a flexible data center fabric with source routing," in *Proc. of ACM SOSR 2015*, pp. 10:1–10:8, Jun. 2015.
[10] A. Hari, T. Lakshman, and G. Wilfong, "Path Switching: Reduced-state flow handling in SDN using path information," in *Proc. of CoNEXT 2015*, Dec. 2015.
[11] Z. Zhu *et al.*, "Demonstration of cooperative resource allocation in an OpenFlow-controlled multidomain and multinational SD-EON testbed," *J. Lightw. Technol.*, vol. 33, pp. 1508–1514, Apr. 2015.
[12] N. Xue *et al.*, "Demonstration of OpenFlow-controlled network orchestration for adaptive SVC video manycast," *IEEE Trans. Multimedia*, vol. 17, pp. 1617–1629, Sept. 2015.
[13] Z. Zhu, S. Li, and X. Chen, "Design QoS-aware multi-path provisioning strategies for efficient cloud-assisted SVC video streaming to heterogeneous clients," *IEEE Trans. Multimedia*, vol. 15, pp. 758–768, Jun. 2013.
[14] K. Wu, P. Lu, and Z. Zhu, "Distributed online scheduling and routing of multicast-oriented tasks for profit-driven cloud computing," *IEEE Commun. Lett.*, vol. 20, pp. 684–687, Apr. 2016.
[15] J. Yao, P. Lu, L. Gong, and Z. Zhu, "On fast and coordinated data backup in geo-distributed optical inter-datacenter networks," *J. Lightw. Technol.*, vol. 33, pp. 3005–3015, Jul. 2015.
[16] Z. Zhu *et al.*, "Impairment- and splitting-aware cloud-ready multicast provisioning in elastic optical networks," *IEEE/ACM Trans. Netw.*, vol. 25, pp. 1220–1234, Apr. 2017.
[17] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. of ACM HotSDN 2013*, pp. 127–132, Aug. 2013.
[18] D. Hu *et al.*, "Flexible flow converging: A systematic case study on forwarding plane programmability of protocol-oblivious forwarding (POF)," *IEEE Access*, vol. 4, pp. 4707–4719, 2016.
[19] S. Li *et al.*, "SR-PVX: A source routing based network virtualization hypervisor to enable POF-FIS programmability in vSDNs," *IEEE Access*, vol. 5, pp. 7659–7666, 2017.
[20] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *Comput. Commun. Rev.*, vol. 44, pp. 87–95, Jul. 2014.
[21] S. Li *et al.*, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, pp. 12–20, Mar./Apr. 2017.
[22] D. Hu *et al.*, "Design and demonstration of SDN-based flexible flow converging with protocol-oblivious forwarding (POF)," in *Proc. of GLOBECOM 2015*, pp. 1–6, Dec. 2015.
[23] K. Han *et al.*, "Leveraging protocol-oblivious forwarding (POF) to realize NFV-assisted mobility management," in *Proc. of GLOBECOM 2017*, pp. 1–6, Dec. 2017.
[24] S. Li, K. Han, H. Huang, and Z. Zhu, "PVFlow: flow-table virtualization in POF-based vSDN hypervisor (PVX)," in *Proc. of ICNC 2018*, pp. 1–5, Mar. 2018.
[25] RFC 791: Internet Protocol. [Online]. Available: https://tools.ietf.org/html/rfc791
[26] C. Guo *et al.*, "SecondNet: A data center network virtualization architecture with bandwidth guarantees," in *Proc. of CoNEXT 2010*, pp. 15:1–15:12, 2010.
[27] Z. Guo *et al.*, "JumpFlow: Reducing flow table usage in software-defined networks," *Comput. Netw.*, vol. 92, Part 2, pp. 300 – 315, Dec. 2015.
[28] R. Ramos, M. Martinello, and C. Rothenberg, "SlickFlow: Resilient source routing in data center networks unlocked by OpenFlow," in *Proc. of LCN 2013*, pp. 606–613, Oct. 2013.
[29] S. Li, D. Hu, W. Fang, and Z. Zhu, "Source routing with protocol-oblivious forwarding (POF) to enable efficient e-health data transfers," in *Proc. of ICC 2016*, pp. 1–6, May. 2016.
[30] F. Ji *et al.*, "Dynamic p-cycle protection in spectrum-sliced elastic optical networks," *J. Lightw. Technol.*, vol. 32, pp. 1190–1199, Mar. 2014.
[31] X. Chen *et al.*, "Flexible availability-aware differentiated protection in software-defined elastic optical networks," *J. Lightw. Technol.*, vol. 33, pp. 3872–3882, Sept. 2015.
[32] S. Li *et al.*, "Flexible traffic engineering (F-TE): When OpenFlow meets multi-protocol IP-forwarding," *IEEE Commun. Lett.*, vol. 18, pp. 1699–1702, Oct. 2014.
[33] L. Gong *et al.*, "Efficient resource allocation for all-optical multicasting over spectrum-sliced elastic optical networks," *J. Opt. Commun. Netw.*, vol. 5, pp. 836–847, Aug. 2013.
[34] Y. Zhang and N. Ansari, "On architecture design, congestion notification, tcp incast and power consumption in data centers," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 1, pp. 39–64, First Quarter 2013.
[35] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. of SIGCOMM 2008*, pp. 63–74, 2008.
[36] W. Lu and Z. Zhu, "Dynamic service provisioning of advance reservation requests in elastic optical networks," *J. Lightw. Technol.*, vol. 31, pp. 1621–1627, May 2013.
[37] The CAIDA chicago statistical information for the CAIDA anonymized internet traces. [Online]. Available: http://www.caida.org/data/realtime/passive/?monitor=equinix-chicago-dirB
[38] POFSwitch. [Online]. Available: http://www.poforwarding.org
[39] POFSwitch and POX-based POF controller developed by the team in the University of Science and Technology of China. [Online]. Available: https://github.com/USTC-INFINITELAB
[40] OpenvSwitch. [Online]. Available: http://openvswitch.org/
[41] POX. [Online]. Available: https://openflow.stanford.edu/display/ONL/POX+Wiki#POXWiki-InstallingPOX
[42] DPDK: Data Plane Development Kit. [Online]. Available: http://dpdk.org/
[43] iPerf. [Online]. Available: https://iperf.fr
[44] L. Kou, G. Markowsky, and L. Berman, "A fast algorithm for steiner trees," *Acta Informatica*, vol. 15, no. 2, pp. 141–145, 1981.