# Improving SDN with InSPired Switches

Roberto Bifulco[†], Julien Boite[‡], Mathieu Bouet[‡], Fabian Schneider[†]
[†] NEC Laboratories Europe [‡] Thales Communications & Security

## ABSTRACT

In SDN, complex protocol interactions that require forging network packets are handled on the controller side. While this ensures flexibility, both performance and scalability are impacted, introducing serious concerns about the applicability of SDN at scale. To improve on these issues, without infringing the SDN principles of control and data planes separation, we propose an API for programming the generation of packets in SDN switches. Our InSP API allows a programmer to define in-switch packet generation operations, which include the specification of triggering conditions, packet's content and forwarding actions. To validate our design, we implemented the InSP API in an OpenFlow software switch and in a controller, requiring only minor modifications. Finally, we demonstrate that the application of the InSP API, for the implementation of a typical ARP-handling use case, is beneficial for the scalability of both switches and controller.

## CCS Concepts

•**Networks** → **Programming interfaces; Bridges and switches; Programmable networks; Packet-switching networks;** *Network performance evaluation; Network manageability;*

## Keywords

Software-defined Networking; Programming abstractions; OpenFlow

## 1. INTRODUCTION

The last few years have seen the establishment of SDN as a concrete approach to build better networks and to introduce innovation in an ossified field [24], with a growing number of deployments certifying this success [15]. Nonetheless, despite the good behind the intuitions that led to the design of the SDN principles [9], the SDN architecture and technologies are iteratively being updated to address the issues that are highlighted by the production deployments [28]. On the one hand, the current generation of forwarding devices, i.e., switches, is not ready to support the flexible switch's programming model introduced with SDN. Limited forwarding table space [16], slow forwarding policy updates [14], limited throughput in control messages handling [25], and slow synchronization between data and control planes [21] are just some of the issues that are being addressed on the switch side. Likewise, a number of problems are being addressed on the controller side, i.e., where the network's control plane is implemented. Controller scalability [8], reliability [3], as well as fundamental questions about controller placement [12, 13], network policy consistency [34] and network view consistency [20] can be mentioned as relevant examples of work dealing with the SDN's control plane implementation.

### *Delegation of control.*

A way to address some of the mentioned issues is to evolve the SDN design, redrawing the line that separates controller's functions from switch's functions [4, 25]. When looking at OpenFlow, one of the most deployed SDN technologies, we can spot the evidence of this design adaptation activity, observing the changes introduced in the different versions of the OpenFlow specification. In OpenFlow 1.0 [30], the switch was completely described by a single flow table that contains Flow Table Entries (FTEs), which in turn were composed of a *match* part and an *action* part. In this match/action paradigm, the match clause defines the traffic to which the specified action is applied, and any change for a flow's action requires an interaction with the controller. Thus, supporting, e.g., fast rerouting of the flows when a link fails requires a round trip with the controller, which usually corresponds to an unacceptable increase in the reaction delay. Already in OpenFlow 1.1 [31], the specification was enriched with the definition of *group tables*, i.e., an abstraction to program a flexible selection of the switch's out port when performing a forwarding action, without requiring interactions with the controller. That is, the functions separation line was redrawn to let the switch react autonomously (e.g., port selection) within the boundaries put by the controller (e.g., in response to a given port down event). In general, this kind of design decisions are discussed in the context of *delegation of control*, and it should be clear that they do not violate the SDN principle of data plane and control plane separation. According to the SDN architecture [36] published by the ONF, the SDN controller will take all the decisions required to operate a network, and instruct the network elements, which in turn execute the decisions of the SDN controller. However, the SDN architecture also explicitly allows to delegate certain functionality to the network element. Such a delegation of control functionality, from the SDN controller to the SDN switch, is bound to the constraints that the SDN controller can alter and/or revoke the delegated control functionality at any time. Moreover, it is expected that enough information is fed back from the SDN switch to the controller.

As in the case of the mentioned OpenFlow example, delegating

*SOSR '16, March 14-15, 2016, Santa Clara, CA, USA*

DOI: http://dx.doi.org/10.1145/2890955.2890962

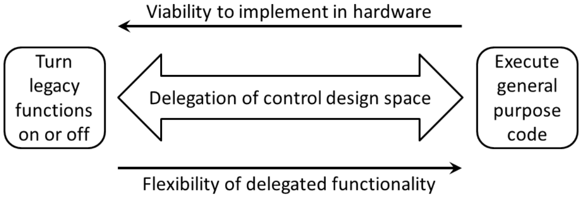

Viability to implement in hardware
Turn legacy functions on or off
Delegation of control design space
Execute general purpose code
Flexibility of delegated functionality

**Figure 1: Design space for delegation of control to SDN switches**

functions to the SDN switch may have several benefits, such as *(i)* reducing the processing load at the SDN controller, *(ii)* reducing the control loop delay (SDN switch ↔ SDN controller), *(iii)* reducing the load on the control channel/network. However, there is a fairly broad design space for the delegation of functionality to the SDN switch. We illustrated this in Figure 1. One extreme is simply to turn on or off certain well-known functions such as MAC learning, ICMP handling or path protection switching (cf. left-hand side of Fig. 1). This is for example the current approach used in the recent versions of the OpenFlow specification [33]. The main drawback is that this approach does not really create an abstraction of the capabilities in the SDN switch. As such, it does not allow to re-purpose the capabilities and create new functionality from it. Yet, this option is easy to support in hardware, as it only requires to expose the existing functions to the controller. Another extreme option is to let the SDN controller push arbitrary code fragments to the SDN switch, which can execute general purpose functions (cf. right-hand side of Fig. 1). While from a flexibility and re-purpose point of view this is certainly desirable, it brings along several issues, as the past research on active networks taught us [9]. Examples of such issues are the need to support a common code execution platform on all network elements, the need to guarantee certain throughput with arbitrary processing, a new world of potential security issues, etc. Therefore, we believe that the ideal solution lies somewhere in the middle, following the spirit of the original OpenFlow design.

*Contribution.*

In this paper we address a specific issue in delegation of control: the programming of packet generation operations in SDN switches, and in particular in OpenFlow switches[1]. Thus, even if we believe our findings have broader application, this paper will consider only OpenFlow networks.

Our main contribution is the presentation of an In-Switch Packet generation (InSP) API, which allows the controller to program the autonomous generation of packets in the switch. In our API, the packet generation operation can be specified by providing three pieces of information: the *trigger*, the *content* and the *actions*. The trigger tells the switch *when* a packet should be generated; the content specifies *what* are the packet's header and payload; the actions specify *how* the switch should use the packet. We leverage the OpenFlow abstractions such as flow tables and FTEs, and define new ones for supporting the specification of the trigger, content and actions information. First, we define a Packet Template Table to store the content of the packets that will be generated by the switch. Each Packet Template Table Entry (PTE) specifies the *content* of one packet and has a unique identifier that is used as a reference in other parts of the API. Second, we add a new OpenFlow instruction, the InSP instruction, that specifies the *actions* using standard OpenFlow actions. Finally, the *trigger* is provided by defining a FTE that contains the InSP instruction. In fact, the InSP instruction contains also a PTE's identifier that points to the corresponding PTE. Whenever a packet is matched by the FTE, the InSP instruction is triggered and the pointed PTE is used to generate the packet to which the instruction's actions are applied.

We implemented our InSP API in a software prototype, which we used to evaluate both the API design and its implementation's performance. To validate the API's design, we present the implementation of a typical service offered in OpenFlow networks, i.e., an ARP responder, and discuss the implementation of other common services, such as ICMP handling. The prototype's performance evaluation shows that implementing InSP is feasible and beneficial to the scalability of both switch and controller. In fact, handling the InSP's packet generation requires less switch's resources than handling an interaction with the controller using the OpenFlow protocol. Likewise, the control plane scalability is increased, since the controller is offloaded of the packet generation operations. Furthermore, handling packet generation in the switch guarantees a tenfold reduction for the packet generation time, when compared to the standard OpenFlow case, in our prototype. We conclude our evaluation presenting an extended analytical study of the implementation of an ARP responder in a datacenter, comparing typical OpenFlow approaches against an InSP-based approach. Our evaluation shows that the InSP-based ARP responder can save up to 65%-91% dataplane messages and 30%-96% control plane messages, depending on the considered case.

*Organization.*

The paper is organized as follows. In Section 2 we give an overview of OpenFlow and of ARP services implementations in OpenFlow networks, since it is relevant information for the InSP API design and its evaluation. Section 3 presents the API, while its application to relevant use cases is discussed in Section 4. In Section 5 we present our prototype and its evaluation using several benchmarks, furthermore, we present an analytical study to evaluate the benefits of using InSP to implement an ARP reponder in a datacenter. In Section 6 we discuss the rationale behind our design decisions and point out possible issues. Finally, Section 7 presents related work and Section 8 concludes the paper.

## 2. BACKGROUND

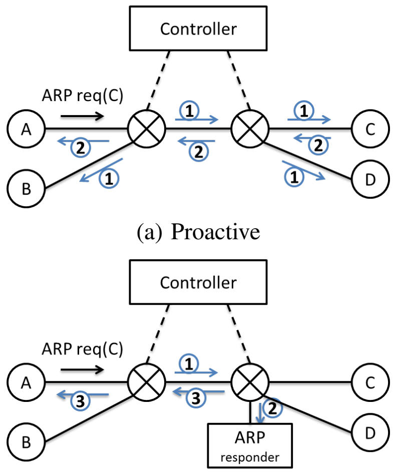
This section presents a brief overview over OpenFlow, introducing concepts and data structures that are required for the understanding of the InSP design presented in Section 3. Furthermore, we provide examples of ARP services in OpenFlow networks, since the implementation of an ARP responder service is the main use case that we study to validate the InSP API in Section 4 and Section 5.

### 2.1 OpenFlow

The OpenFlow specification defines a switch programming model and a network protocol to program/configure the switch. In this paper we will always refer to the version 1.3 of the OpenFlow specification [32], since it is widely implemented and deployed.

A switch is described by a pipeline of flow tables. Each flow table can contain one or more Flow Table Entries (FTEs), which are constituted by a match part and an instruction part. The match part is composed by a set of values for packet header's fields, whose combination is used to identify a network flow. The instruction part contains one or more OpenFlow instructions. An instruction is executed only when the containing FTE matches a packet, and it is always executed as soon as the packet is matched by the entry. Only one instruction per type can be attached to a FTE and the instruction execution order is pre-specified by the OpenFlow specification.

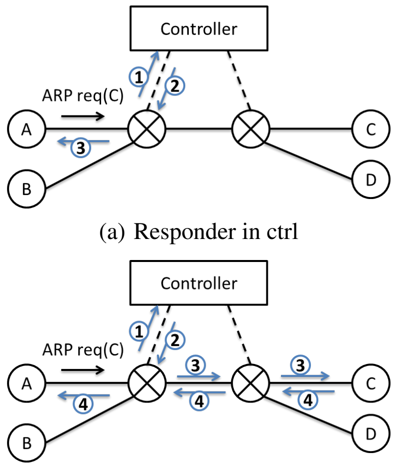
[1] This work is part of a broader research project, the EU BeBa project [1], which also studies the options to enable the programmability of stateful forwarding actions and protocols in SDN switches.



(a) Proactive

(b) Responder

**Figure 2: Proactive approaches for ARP handling.**



(a) Responder in ctrl

(b) Unicast

**Figure 3: Reactive approaches for ARP handling.**

Examples of instruction types are the GoTo instruction, which tells the switch what should be a packet's next flow table in the pipeline, and the WriteActions instruction, which adds OpenFlow actions to a packet's action set.

The packet's action set is a data structure attached to any packet received by the switch, as soon as the packet is injected in the pipeline. The action set is initially empty, but it is modified during the pipeline traversing. For example, a packet could be matched by a FTE that contains a WriteActions instruction. When the packet exits the pipeline, all the actions contained in its action set are executed. Typical actions are SetField, to change the value of a packet header's field, and Output. The latter is particularly interesting, because it is specified together with a port variable, which can be either a physical switch's port or a logical one. If a physical port is specified, the packet is forwarded to such a switch's port. If a logical port is used, different operations may be performed depending on the port type. For instance, a TABLE port tells the switch to re-inject the packet at the beginning of the switch's pipeline. Another example is the CONTROLLER port, which instructs the switch to send the packet to the controller, using the OpenFlow PACKET_IN message.

We conclude this overview mentioning two OpenFlow protocol messages: the PACKET_OUT message, which is used by the controller to send a packet to the switch in order to forward it to a switch's port(s); and the FLOW_MOD message used by the controller to install and delete FTEs.

## 2.2 ARP in OpenFlow networks

ARP handling is a basic network function that enables end-hosts at learning neighbors' MAC addresses. As such, even being a simple service, its implementation in SDN is always required and, thus, critical when end-hosts run unmodified network stacks. In Ethernet networks, an ARP request is used to learn the MAC address of a host for which only the IP address is known. A host makes a query for unknown MAC addresses by sending ARP requests. ARP requests are flooded to all the other hosts on the same LAN, since they are encapsulated in Ethernet broadcast frames. Eventually, a request arrives at the destination host, which sends an unicast ARP reply that traverses the network back to the sender, providing the MAC/IP address mapping information. The support for this basic function in OpenFlow networks may be introduced either proactively (cf. Fig. 2), without involving the controller in the ARP handshake, or it can be introduced using a reactive approach (cf. Fig. 3), which requires the controller to deal with ARP requests.

In proactive approaches, the controller emulates the behavior of legacy Ethernet networks. In the most simple case, the controller proactively installs FTEs to flood the ARP requests (cf. Fig. 2(a)). This solution keeps the controller unaware of the network location of the hosts and, in particular, has the drawback of introducing broadcast traffic in the network. Broadcast may be an issue for some networks that implement advanced services. For instance, it may force the implementation of spanning tree protocols to avoid forwarding loops and, in general, it may require the introduction of additional complexity in the controller to deal with the broadcast traffic. An alternative proactive solution, which avoids broadcasts, involves the deployment of a function that receives all the broadcast packets [19]. In the case of ARP handling, such a function is an ARP responder (cf. Fig. 2(b)). The switches at the network's edge transform any broadcast ARP message in an unicast ARP message with the responder as destination. In turn, the ARP responder learns from the received ARP messages the MAC/IP address mappings, so that it can later create ARP replies to answer the incoming ARP requests.

In reactive approaches, shown in Figure 3, the ARP responder function can be implemented in the controller itself (cf. Fig. 3(a)). This is the approach used, for instance, by the OpenDaylight [29] and ONOS [3] controllers. The edge switch generates a PACKET_IN message for the controller, which contains the received ARP request. Then, the controller responds generating an ARP reply and sending it to the switch using a PACKET_OUT message. Finally, the switch forwards the ARP reply. Notice that, in this case, the controller has to take care of keeping an updated ARP/IP address mapping information. For example, it may need to refresh its mapping information by creating ARP requests and injecting those in the network.

To simplify the controller implementation, the approach of Figure 3(b) can be used instead. In this case, the controller lets the actual destination host generate the ARP reply. An edge switch forwards a received ARP request (using a PACKET_IN) to the controller. Assuming that the mapping information was learned already, the controller changes the ARP request message's broadcast destination MAC address with the known unicast destination MAC address of the destination host. Then, the modified ARP request message is sent back to the switch (using a PACKET_OUT), which, in turn, sends it to the actual destination host that will respond with the ARP reply. This solution is implemented by the Beacon [8] and Floodlight [10] controllers.

# 3. IN-SWITCH PACKET GENERATION API

In this section, we describe the abstractions and design of our In-Switch Packet Generation (InSP) API, including the corresponding structures and function calls. Our work is inspired by, and builds on top of, the abstractions defined by the OpenFlow specification.

A programmable in-switch packet generation operation is described by the following three pieces of information:

- *trigger*: the event in response of which a packet is generated. For instance, a triggering event could be the reception of a given packet, or the expiration of a timer.
- *content*: specifies the packet's header and payload.
- *actions*: defines what the switch should do with the packet. For instance, which switch's port should be used to send out the generated packet on the network.

We believe that considering these three components individually gives greater flexibility and extensibility to the API, as it simplifies the independent definition and extension of each of those. For instance, a packet generation may be triggered both by a packet reception event or by a switch's port down event. Likewise, the same generated packet may be associated with different forwarding actions, depending on the event(s) that triggered the generation.

To provide these components, the InSP API leverages the OpenFlow's abstractions, i.e., the flow table and instruction data structures, and adds two more data structures: the Packet Template Table and the In-Switch Packet Generation Instruction. The rest of this section describes these two data structures and their use to provide in-switch packet generations.

## 3.1 Packet Template Table

The Packet Template Table is the data structure used to store the content of the packets the switch will generate. As the name suggests, a Packet Template Table Entry (PTE) specifies a template that is used for the generation of a packet's content, with each PTE specifying the content for exactly one type of packet. A PTE is composed of three pieces: (i) a packet template id (*pkttmp_id*); (ii) the packet content template; (iii) an array of *copy operations*. The *pkttmp_id* is used to identify a PTE and to reference it from other data structures defined in the API. The packet content template is specified as a byte array, which should have the same size of the packet that is going to be generated. Finally, the copy operations are applied whenever a new packet has to be generated. Each copy operation changes a subset of the PTE's packet content template's bytes. Once all the copy operations are applied, the resulting byte array is used as the generated packet's content.

A copy operation looks like a regular copy between byte arrays, being completely specified by data source, offset in the source, offset in the destination, data length. The destination of the copy operation is always the generated packet's content (which is originally a plain copy of the PTE's packet content template). The source may instead have different values and it is one of the parts of the API that is subject for future extensions, as we discuss later in this section. We currently allow only one type of source, which is the content of a packet that triggers the generation of a new packet.

To modify the Packet Template Table's entries, we defined a Packet Template Modification (PKTTMP_MOD) message type. With a semantic similar to the one of OpenFlow's FLOW_MOD messages, a PKTTMP_MOD is used to add or delete PTEs. If the PKTTMP_MOD message contains the "add" command, then it specifies all the information required by the PTE, i.e., *pkttmp_id*, packet content template and copy operations, if any. Instead, if the PKTTMP_MOD contains a "delete" command, only the *pkttmp_id* is specifed.

## 3.2 In-Switch Packet Generation Instruction

We leverage the OpenFlow's instruction data structure to create a new instruction type, the In-Switch Packet Generation instruction, to trigger the generation of a new packet. The instruction contains a *pkttmp_id* and a set of OpenFlow actions. The *pkttmp_id* is used to identify the PTE that should be used to create the generated packet's content, while the set of actions defines what should happen with the newly generated packet. The main difference with OpenFlow's standard instructions is that the InSP instruction creates a new packet that the switch has to handle, in addition to the packet that matched the FTE. Thus, while the standard OpenFlow instructions are applied to the same packet that was matched by the FTE that triggered the instruction execution, the InSP instruction is instead just triggered by such packet and its execution has effects only on the newly generated packet. As final effect, the original packet received by the switch, i.e., the triggering packet, will continue its processing on the switch's pipeline, while the processing of the newly generated packet will depend by the actions defined in the InSP instruction.

The support for standard OpenFlow actions in the InSP instruction opens a number of possibilities for defining the behavior of the generated packet. For example, a programmer may define an explicit forwarding action like the OpenFlow's OUTPUT action, selecting the switch's output port to use for the forwarding of the packet. In another case, the programmer may instead inject the generated packet in the beginning of the switch's pipeline. For instance, this may be helpful when the desired output port is unknown at the moment in which the InSP instruction is defined, or when the actual output port should be decided by the current state of the switch.

## 3.3 InSP walkthrough

As we said in the beginning of this section, an in-switch packet generation operation is completely specified by the definition of the trigger, content and actions. The API presented so far allows a programmer at defining these three components. First, the programmer specifies the content by creating a PTE. Second, she specifies the actions by defining an InSP instruction. Finally, the packet generation trigger is specified by defining a FTE which includes the InSP instruction in its instructions list.

Assuming that a programmer has performed these three steps, the in-switch packet generation process unfolds as follows. The switch receives a packet at one of its ports and injects it in the flow tables pipeline. The action set gets filled as the packet flows through the pipeline, since matching FTEs' instructions may write actions to it. If the packet is matched by a FTE that contains an InSP instruction, then the packet generation process is triggered. A new packet is created by copying the InSP instruction's packet content template. Then, copy operations are applied to the newly generated packet. For example, a copy operation may copy the received packet's source Ethernet address and write it to the new packet's destination Ethernet address. Finally, the InSP instruction's actions are applied to the packet. Notice that the InSP instruction only contains actions that are immediately applied to a packet. That is, the newly generated packet is not associated with an action set. The triggering packet, which is still being processed by the pipeline, continues its processing after the InSP instruction has been executed. That is, the triggering packet eventually exits the pipeline and its action set is executed.

## 3.4 Issues and extensions

It is worth to highlight a few important points that stem out from the description we presented so far.

**Copy operation**. We defined just one possible source for a copy operation, which is the content (header and payload) of the triggering packet. We believe several different sources may be specified, for instance we foresee as possible sources the values of an entry in a flow table, a counter's value, a timeout value, etc. We believe that the definition of new copy operation's sources will come as soon as use cases will bring new requirements. A second observation is that a copy operation is completely agnostic to protocol definitions, i.e., a predefined number of bytes is copied starting from a given offset, without any knowledge about, e.g., header fields locations. This may rise concerns regarding copying the wrong data if the packet includes, e.g., unexpected header fields. However, we believe this way we can provide much more flexible and efficient packet generation procedures, while still giving programmers a mean to guarantee the correct execution of the packet generation. For instance, a programmer can define the FTEs in order to make sure that only packets with the expected packet header fields are actually handled by a given PTE. In effect, this way we leverage the packet parsing done by the switch to perform FTEs matching, without wasting additional resources to parse again the packet during copy operations execution.

**Checksum**. So far, for clarity we have omitted in our presentation the need to deal with protocols' checksum. While for the Ethernet's header the checksum may be easily added automatically by the outgoing network interfaces, this may not be the case for other protocols' checksum, such as IP, ICMP, TCP, etc. In InSP we support the definition of a checksum in a generated packet using *checksum operations*. One could think of a checksum operation as a special type of copy operation, which contains the following information: type, source offset, length, destination offset. The source field of the copy operation is substituted by a *type* field in this case, since the source for a checksum operation is always the content of the generated packet. The field is instead used to specify the type of checksum, which specifies also its length in bytes. A checksum operation is used in the same way of copy operations, i.e., it is contained in a PTE.

**InSP instruction processing**. When a InSP instruction is executed, the corresponding processing of a generated packet is decoupled from the processing of the triggering packet, i.e., the packets are handled independently in the switch. It should be clear that while we mention an immediate execution for the InSP instruction, whenever it is triggered, we actually do not mandate any ordering between the processing of the generated and of the triggering packets. In other words, one can think of the process of generating a packet as a parallel execution thread to that of processing the triggering packet. The rationale behind this decision is that we believe mandating an ordering may help only in the implementation of a small subset of use cases, while we recognize it is not always achievable (or desirable) to have it. For example, in a hardware switch the packet generation may be performed on the slow path of the device, while packet forwarding happens in the fast path. Mandating, e.g., that packet generation should happen before the triggering packet handling may delay the packet forwarding process, which may be undesirable. Moreover, it may introduce complications to the switch implementation, as the triggering packet may require staging in a buffer while waiting for the packet generation. Please notice that the lack of a mandated execution ordering does not break the OpenFlow specification with regard to the relative instructions execution ordering. In fact, instruction ordering according to OpenFlow relates only to the handling of a received packet, while the unspecified ordering in our API is related to the relationship between the processing events of two different packets (the received and the generated ones).

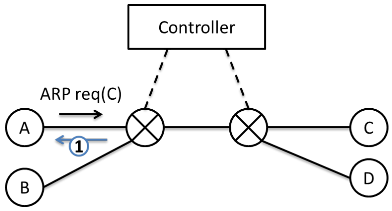


**Figure 4: InSP implementation of an ARP responder.**

**Trigger**. Since the OpenFlow's instructions can be attached only to a FTE, our API can trigger the generation of a packet only in reaction to the reception of a packet at the switch. This admittedly limits the applicability of our packet generation API as we cannot handle, for instance, timed events. We believe it is possible to provide extensions to include additional triggering logic, such as programmable timers in the switch, that could complement our InSP API. However, we believe that providing such extensions is out of the scope of our InSP proposal. In fact, we provide use cases that can be already fully implemented with the defined InSP API. As such, we believe the InSP API has already a value as is, and will consider any extension as part of our future work.

# 4. EXAMPLES

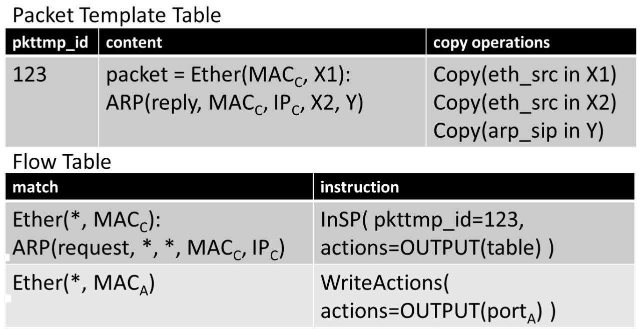
In this section we provide application examples of the InSP API for the implementation of ARP and ICMP handling. While we could report on the implementation of more complex and innovative use cases, we believe these two examples immediately highlight the advantages of the proposed API in relation to the current OpenFlow-based approach.

## 4.1 ARP

When the network supports the InSP API, ARP handling can be implemented leveraging the packet generation capabilities of the switches. In this case, a switch generates an ARP reply when it receives an ARP request for a known host (Fig. 4). The procedure to program a switch unfolds as follows. First, whenever the controller learns a MAC/IP address mapping, a corresponding PTE is installed at the edge switches. The PTE specifies an ARP reply with the MAC/IP address mapping information. It also includes a set of copy operations to copy the source MAC and IP addresses of a triggering packet to the PTE content's bytes corresponding to *(i)* the header's destination MAC and IP addresses, and *(ii)* the ARP reply's target addresses. Together with the PTE, also a FTE is installed. Such FTE matches the ARP requests that will trigger the generation of a packet using the aforementioned PTE. Fig. 5 shows an example of configurations for the Packet Template Table and Flow Table for the case of Fig. 4. The InSP instruction's action is OUTPUT(table), thus, it injects a generated packet in the switch's pipeline, where it will be matched by the second FTE. Such entry is configured by the controller to implement L2 forwarding, since the generated packet has the Ethernet's destination MAC address set to the ARP request's source MAC address value, it will be forwarded to the correct switch's port.

## 4.2 ICMP

The InSP API enables an easy implementation of many ICMP reply messages. For instance, Figure 6 shows the tables configuration used to generate ICMP TimeExceeded messages when an IP



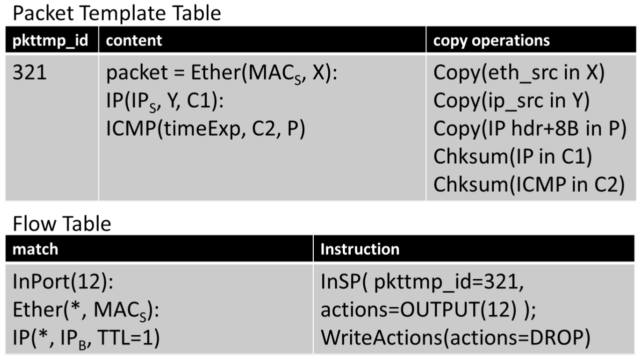
Packet Template Table

| pkttmp_id | content | copy operations |
|---|---|---|
| 123 | packet = Ether(MAC$_C$, X1):<br>ARP(reply, MAC$_C$, IP$_C$, X2, Y) | Copy(eth_src in X1)<br>Copy(eth_src in X2)<br>Copy(arp_sip in Y) |

Flow Table

| match | instruction |
|---|---|
| Ether(*, MAC$_C$):<br>ARP(request, *, *, MAC$_C$, IP$_C$) | InSP( pkttmp_id=123,<br>actions=OUTPUT(table) ) |
| Ether(*, MAC$_A$) | WriteActions(<br>actions=OUTPUT(port$_A$) ) |

**Figure 5: Example of Packet Template Table and Flow Table configurations for handling ARP requests.**



Packet Template Table

| pkttmp_id | content | copy operations |
|---|---|---|
| 321 | packet = Ether(MAC$_S$, X):<br>IP(IP$_S$, Y, C1):<br>ICMP(timeExp, C2, P) | Copy(eth_src in X)<br>Copy(ip_src in Y)<br>Copy(IP hdr+8B in P)<br>Chksum(IP in C1)<br>Chksum(ICMP in C2) |

Flow Table

| match | Instruction |
|---|---|
| InPort(12):<br>Ether(*, MAC$_S$):<br>IP(*, IP$_B$, TTL=1) | InSP( pkttmp_id=321,<br>actions=OUTPUT(12) );<br>WriteActions(actions=DROP) |

**Figure 6: Example of Packet Template Table and Flow Table configurations for the generation of ICMP Time Exceeded messages.**

packet with TTL=1 is received. In particular, notice the use of the checksum operations for setting both the IP and ICMP checksums.

## 5. EVALUATION

This section presents a prototype implementation of the InSP API and a number of benchmarks that evaluate the prototype's performance. Then, we study the case of ARP handling in a datacenter, to evaluate the impact on the number of control and data messages generated when the network supports InSP, comparing it to two standard OpenFlow cases.

### 5.1 Benchmarks

To test the InSP API, we implemented a software prototype for both the switch and the controller sides. The switch implementation is based on OfSoftSwitch13 [27], and requires the addition of less than 700 lines of C code to the original switch's code base. For the controller side, we modified the RYU SDN framework [35] to support the generations of the PKTTMP_MOD messages and the specification of InSP instructions: a modification that required less than 300 lines of python code. In both cases, the InSP API was implemented as an OpenFlow's experimenter extension, which allows one at introducing new features in an OpenFlow switch, while being compatible with any other OpenFlow switch and agnostic to the OpenFlow protocol version. The rest of this subsection describes an evaluation of our implementation in terms of in-switch packet generation reaction time, processing costs for an InSP instruction, performance impact on the controller and memory requirements for the Packet Template Table.

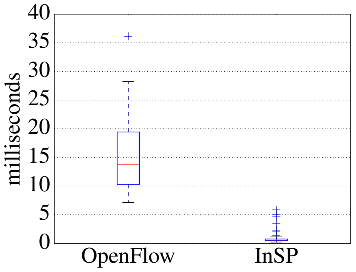


**Figure 7: Time (in milliseconds) to generate an ARP reply when using traditional OpenFlow (i.e., involving the controller) and when using InSP.**

All the tests are executed on a computer equipped with an Intel(R) Core(TM) i5-2540M CPU @ 2.60GHz (2 cores, 4 threads). During the tests hyper-threading has been disabled. The operating system and the load generator run on the first CPU's core. We use Nping [26] as load generator, sending ARP packets at different rates depending on the test. The controller and the switch share the second CPU's core. Any communication between the controller and the switch happens over a local TCP channel. We run a single switch instance and an ARP responder application at the controller during our tests. The controller's application installs a PTE at the switch, which contains an ARP reply template. Then, it installs a FTE that contains an InSP instruction, in order to trigger the packet generation upon reception of an ARP request. In all the cases we compare the InSP implementation of the ARP responder with an analogous application implemented using standard OpenFlow. The OpenFlow ARP responder installs a FTE at the switch to generate a PACKET_IN upon reception of an ARP request. The controller answers to the PACKET_IN with a PACKET_OUT that contains a statically defined ARP response, in order to minimize the processing time at the controller.

**Reaction time**. In our first test, we measure the time it takes to send an ARP request and receive the corresponding ARP reply. That is, we define the reaction time as the time difference between the time of reception of an ARP reply and the time at which the corresponding ARP request was sent. We generate ARP requests instrumenting Nping to generate a total of 100 requests at a rate of 5 requests per second.

As expected, the response time is much lower for the InSP case (cf. Fig. 7), with an average reaction time of less than 1ms, since the ARP reply is generated as soon as the ARP request is received at the switch. For the OpenFlow case, instead, the generation of a response requires a round trip with the controller (PACKET_IN + PACKET_OUT), which is furthermore running on top of the python interpreter. Thus, the reaction time grows to 10-20ms, in most of the cases. Please notice that even if we recognize that most of the delay in this case is introduced by the python implementation, for which the high variation in the measured reaction time is an evidence, we also point out that most of the controllers are implemented in high level languages, such as Java or python. Furthermore, the InSP ARP responder application is also implemented in python, but, by pushing the generation of packets down to the switch, the controller implementation technology is decoupled from the actual packets generation performance. Finally, the InSP case speeds up the response generation by avoiding the communication with the controller, which may introduce bigger delays in geographically distributed networks.

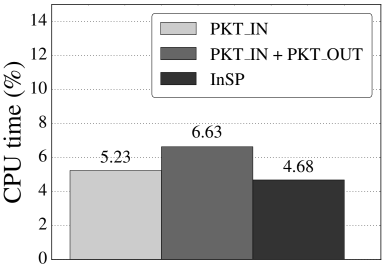


**Figure 8: Average CPU time (in %) used by the software switch to process 100 ARP requests per second. Three cases are shown: the switch generates only PACKET_IN messages, the switch generates PACKET_IN and process the corresponding PACKET_OUT, the switch implements InSP and performs an in-switch packet generation.**

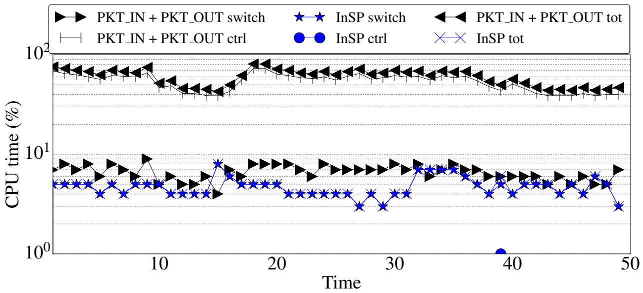


**Figure 9: CPU time (in %) used by the software switch to process 100 ARP requests per second, over a period of 50 seconds. The current OpenFlow approach, which includes the processing of PACKET_IN and PACKET_OUT both at the switch and the controller, is compared to the InSP approach.**

**Switch CPU**. While the reaction time may be improved with switches that implement the InSP API, one may be concerned about the increased processing load on the switch. In fact, the switch's CPU is one of the performance bottlenecks for current SDN and OpenFlow switches [18]. To understand the cost of implementing in-switch packet generation, we run a new test using the ARP responder application, monitoring the CPU time used by the switch in the meanwhile. Again, we compare the results with a switch configured to run the standard OpenFlow version of the ARP responder, in which the switch does not perform packet generation, but has to handle PACKET_IN and PACKET_OUT messages. In this test, we instrument Nping to generate ARP requests at a rate of 100 packets per second, over a time window of 50s. In Fig. 8, the central bar and the right-hand bar show the results of our test for the OpenFlow and InSP case, respectively. From the results it is clear that the in-switch packet generation is actually cheaper in terms of CPU time than the PACKET_IN/PACKET_OUT handling. Actually, in a third test, in which we let the switch generate only PACKET_INs without processing PACKET_OUTs in response, we verified that the PACKET_INs handling alone is more expensive than the in-switch packet generation (cf. left-hand bar of Fig. 8). The explanation for this somewhat counterintuitive result is that the generation of a PACKET_IN is a more complex operation than the in-switch packet generation. In fact, the PACKET_IN handling includes copying the received packets, encapsulating it in an OpenFlow message and sending it to the controller using TCP. Also, notice that a similar processing is required for the PACKET_OUT handling as well. The in-switch packet generation, instead, requires only a lookup in a hash table (to find the relevant PTE) and a few copy operations.

**Controller CPU**. In this test we consider also the impact of using InSP on the system constituted by the combination of controller and switch. In fact, while the switch CPU load lowers, with InSP the controller's CPU is completely offloaded, making the combined system more scalable than its standard OpenFlow alternative. Fig. 9 shows the CPU loads (in percentage over the overall CPU time) during the test. The figure plots the CPU load contributed by the controller, the switch and the total resulting from the sum of these two contributions, both for the case of standard OpenFlow and InSP implementations. As in the previous test, we run the OpenFlow and InSP versions of the ARP responder application, generating 100 ARP requests per second for 50s. The results show that, overall, the InSP implementation requires much less system-wide resources than the OpenFlow implementation, in terms of processing time. That is, while the former uses about 5% of the CPU time (the controller does not contribute to the load), the latter requires about the 60% of it, mostly because of the controller processing time.

**Memory**. For the implementation of the InSP API on a switch, after the processing time (i.e., CPU) another important resource to take into account is memory. In fact, a switch has usually limited memory that can be used to implement additional data structures. While the memory used for installing FTEs with InSP instructions should not have any significant impact when compared to the OpenFlow case, e.g., because in the OpenFlow case there would be anyway FTEs for generating PACKET_INs, the InSP's Packet Template Table is a completely new data structure that OpenFlow does not implement. In our implementation, each PTE is relatively small, with just 4B required for the *pkttmp_id* and 32B (source, source offset, destination offset and length are variables of 4B each) for each copy operation. However, the packet content template size is at least 60B (minimum Ethernet frame size, excluding the 4B CRC code) and may grow to several hundreds of bytes depending on the packets that the programmer wants to generate. Therefore, since memory is limited, a switch cannot support the definition of millions of PTEs with packet content templates of several hundreds of bytes. Nonetheless, we do not believe memory will be an issue for most of the use cases, as many packets that may require in-switch generation are small ones (e.g., ARP replies). In this case, even with one million PTEs, the Packet Template Table would be in the size of 100s of MBs, which easily fits the DRAM of modern switches [6].

## 5.2 ARP handling in datacenter

So far we have shown the benefits of implementing the InSP API in a switch, still we did not perform any evaluation of the impact of using the API in a network. To this end, we provide an analytical study of an InSP-based implementation of the ARP handling use case in a datacenter. Our analytical evaluation compares the number of control and data massages generated by the InSP-based solution against those generated by the OpenFlow reactive approaches presented in Section 2.

### *5.2.1 Topology, parameters and assumptions*

For the purpose of this analysis, we assume a typical datacenter hierarchical network [2] like the one shown in Figure 10. In such highly redundant topologies, protocols like ARP can produce a large amount of broadcast traffic [11].

In our model, the network is managed by an OpenFlow controller and it is composed of: *(i)* a single core switch ($C$); *(ii)* $M$ aggregation switches, each of which is connected to the core switch and to each of its neighboring aggregation switch(es); *(iii)* as many edge

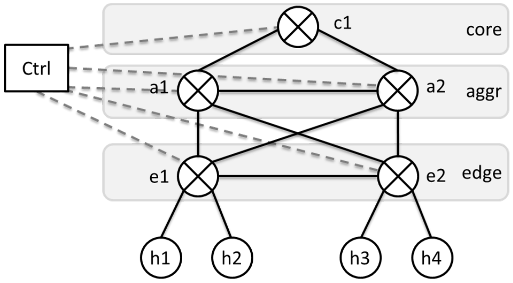


**Figure 10: Datacenter topology model**

switches as aggregation ones, i.e., *M*, each of which is connected to all the aggregation switches and to each of its neighboring edge switch(es). Finally, we assume that each of the edge switches is connected to $\alpha$ hosts. The total number of switches (*S*), hosts (*N*) and links (*E*) in the considered network is shown in Table 1, together with a summary of the aforementioned parameters. To perform our analytical evaluation, we introduce also the parameter *d* (for "distance"), which represents the *maximum number of switches* in the shortest path between two hosts[2].

Throughout our evaluation, we assign to each host a state value that can be either *learnt* (L) or *not learnt* (NL). For a learnt host, the controller knows the host's MAC/IP address mapping, while in case of a not learnt host the mapping is unknown. This classification is helpful to define the behavior of the controller in the 4 possible communication scenarios for an ARP interaction between hosts. That is, an ARP request can be sent by a host which is either learnt or not, to a host which is either learnt or not. Notice in our model a host never changes its state from learnt to not learnt. While the state change from not learnt to learnt can happen in different ways, depending on the considered ARP handling implementation. In all the cases, whenever a host state changes from not learnt to learnt, the controller installs a FTE in all switches to enable direct forwarding of unicast L2 frames to the learnt host. Also, we assume the controller will enforce the forwarding of the flows using the shortest path between the hosts.

### 5.2.2 Evaluation scenarios and comparative results

**Base scenarios and closed-form formulas**. We consider three ARP handling implementation approaches: OpenFlow unicast (*OF-unicast*, cf. Fig. 3(b)), OpenFlow responder (*OF-responder*, cf. Fig. 3(a)) and InSP (cf. Fig. 4).

In OF-unicast, when a host (L or NL) sends an ARP request, the first switch on the path generates a PACKET_IN message from which, if the host was not learnt, the controller learns the host's MAC/IP address mapping. Then, if the destination host is learnt, the controller sends a PACKET_OUT back to the switch transforming the ARP request from broadcast to unicast (cf. Section 2). Otherwise, the ARP request is flooded. In the former case, the ARP request travels throughout the network to the destination host, which generates the ARP reply and sends it back to the requesting host. In the latter case, the ARP request is flooded and at each next switch a new interaction with the controller (PACKET_IN/PACKET_OUT) happens, followed by a new flooding. Notice that, in this case, we assume the controller enforces a spanning tree for the broadcast packets, i.e., broadcast messages are counted once per link.

**Table 1: Description of the evaluation parameters**

| Par. | Description | Value(s) |
|---|---|---|
| C | # of core switches | 1 |
| M | # of aggr/edge switches | $2 \leq \text{M} \leq 50$ |
| $\alpha$ | # of hosts per edge switch | $5 \leq \alpha \leq 50$ |
| S | Tot # of switches | 1 + 2M |
| N | Tot # of hosts | M$\alpha$ |
| E | Tot # of links | $\text{M}^2 + \text{M}(\alpha+3) - 2$ |
| d | Max # of switches on the shortest path between 2 hosts | d=M for $2 \leq \text{M} \leq 3$<br>d=3 for M>3 |

In OF-responder, when a host (L or NL) sends an ARP request, the first switch on the path generates a PACKET_IN message from which, if the host was not learnt, the controller learns the host's MAC/IP address mapping. If the destination host is learnt, the controller generates an ARP reply and sends it back to the switch using a PACKET_OUT message. In turn, the switch forwards the ARP reply to the requesting host. When the destination host is not learnt, the controller sends a PACKET_OUT for each port not connected to a switch of each edge switch. That is, in our model the controller sends as many PACKET_OUTs as hosts[3]. The switches in turn forward the ARP request contained in the received PACKET_OUT. When the destination host responds with an ARP reply, the receiving switch sends it to the controller using a PACKET_IN message. At this point, the state of the destination host becomes learnt, and the controller sends a PACKET_OUT to the requesting host's switch, in order to deliver the ARP reply.

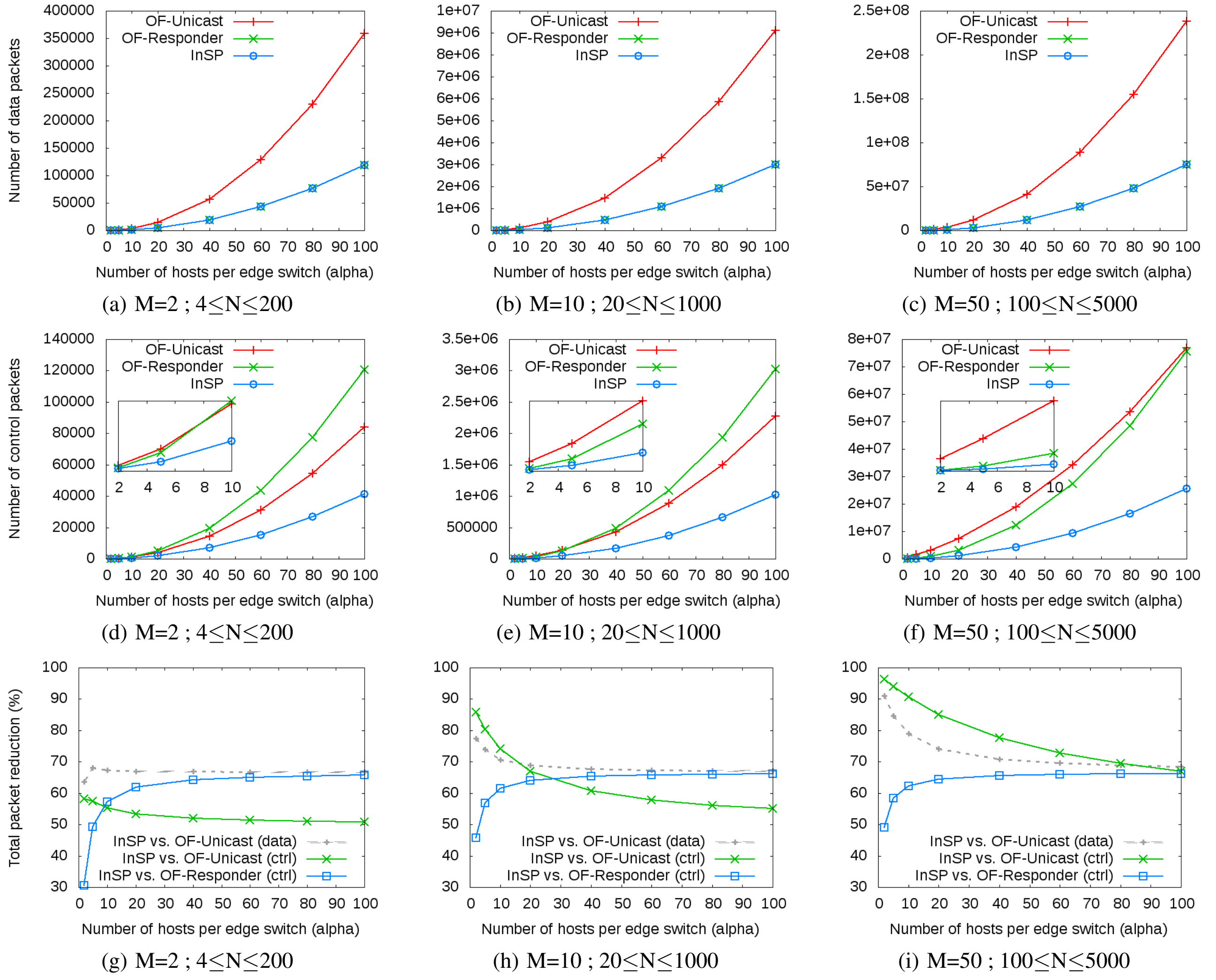
In the InSP case, all the interactions involving not learnt hosts work as in the OF-responder case. However, whenever a host is learnt, together with the installation of the related FTEs, the controller installs PTEs for generating ARP replies for such host, in all the edge switches. Thus, after a host state becomes learnt, any ARP reply, for an incoming ARP request for such host, is generated directly at the edge switch where the request is first received.

In Table 2 we provide the formulas for calculating the number of messages, associated to each scenario for each category of traffic. From the formulas, we can observe that *OF-unicast* uses more dataplane messages, while *OF-responder* and *InSP* generate the same number of ARP messages. Furthermore, *OF-responder* and *InSP* generate the same number of control plane messages for the 3 first scenarios. However, once mappings are learnt ("L-L" scenario), *InSP* does not need generating any control plane messages any more, since ARP replies are generated by the switch. Indeed, this last case where MAC/IP address mappings are known is supposed to happen more often than the cases that involve interactions with not learnt hosts. That is, we assume that the learning usually happens only when the host first connects to the network and that the learnt information does not change for a reasonable period of time.

**General evaluation**. To quantify the benefits that *InSP* may offer, we perform an analysis that is representative of an operational network. We analyze the number of messages generated by the ARP protocol during normal operation, i.e., by considering together the individual scenarios identified above. For this evaluation, we consider that each host in the studied topology wants to ping all the other hosts, assuming that no host is *learnt* at the beginning, and that pings are generated sequentially (no simultaneous pings). Thus, when a first host pings the *N-1* other hosts, the ARP request generated for the very first ping falls into the *NL* to *NL* case, and the first host becomes *learnt*. Then, the ARP requests generated for the *N-2* following pings fall into the *L* to *NL* scenario. After the *N-1* first pings, all hosts are *Learnt*, and all subsequent ARP requests

[2] Notice that the number of switches between 2 hosts actually depends on the relative location of the hosts. We believe that always using the maximum value for this parameter, for all the evaluated scenarios, is a reasonable approximation to simplify the model.

[3] actually, *N*-1, since the requesting host port is excluded



(a) M=2 ; 4≤N≤200 (b) M=10 ; 20≤N≤1000 (c) M=50 ; 100≤N≤5000

(d) M=2 ; 4≤N≤200 (e) M=10 ; 20≤N≤1000 (f) M=50 ; 100≤N≤5000

(g) M=2 ; 4≤N≤200 (h) M=10 ; 20≤N≤1000 (i) M=50 ; 100≤N≤5000

**Figure 11: Comparison of *InSP* versus *OpenFlow-unicast* and *OpenFlow-responder* for ARP handling.**

**Table 2: Closed-form formulas for the different scenarios**

| Scenario | # ARP mess. | # CTRL mess. |
|---|---|---|
| NL-NL OF-unicast | E+d+1 | 2.[E-(N-1)+S]+d+3 |
| NL-NL OF-resp | N+2 | N+2S+3 |
| NL-NL InSP | N+2 | N+2S+3 |
| NL-L OF-unicast | 2.(d+1) | S+2 |
| NL-L OF-resp | 2 | S+2 |
| NL-L InSP | 2 | S+2 |
| L-NL OF-unicast | E+d+1 | 2.[E-(N-1)]+S+2 |
| L-NL OF-resp | N+2 | N+S+3 |
| L-NL InSP | N+2 | N+S+3 |
| L-L OF-unicast | 2.(d+1) | 2 |
| L-L OF-resp | 2 | 2 |
| L-L InSP | 2 | 0 |

fall into the *L* to *L* scenario.

The sum of the formulas identified above for the individual scenarios, weighted according to this sequence, gives closed-form formulas to compute the number of messages generated for this global operation for the 3 approaches. The generic formula is given by Eq. 1.

$$\begin{aligned} \#total &= \mathbf{1} \times (\#ARP_{NL-NL} + \#CTRL_{NL-NL}) \\ &+ \mathbf{(N\text{-}2)} \times (\#ARP_{L-NL} + \#CTRL_{L-NL}) \\ &+ \mathbf{(N\text{-}1)} \times \mathbf{(N\text{-}1)} \times (\#ARP_{L-L} + \#CTRL_{L-L}) \end{aligned} \quad (1)$$

We computed the formula for each ARP handling approach and for different topology sizes, when the number of hosts in the network increases. The results are shown in Fig. 11, where we plot the number of ARP requests and replies (Fig. 11(a) to 11(c)), the number of control plane messages (Fig. 11(d) to 11(f)) and the resulting messages reduction in the data and control planes offered by InSP versus *OF-unicast* and *OF-responder*.

On the data plane side, we observe (Fig. 11(a) to 11(c)) that *OF-unicast* generates more messages than any of the 2 other approaches, no matter the size of the topology and the number of hosts. This is due to not only broadcasting ARP requests when hosts are not learnt, but also forwarding ARP requests up to the target host even when it is learnt. With *OF-responder* however, the controller forwards the request to only a subset of switches when not learnt hosts must be discovered, and handles itself the resolution for learnt hosts. This way, only 2 ARP messages are generated for the resolution of known mappings. In this, *InSP* performs the same as *OF-responder* in terms of data messages. However, in this case the ARP replies are generated by the switch that received the request, without involving the controller (and thus improving also reaction times, cf. Fig. 7).

The offloading of the control plane is visible in Fig. 11(d) to 11(f), where the number of packets generated with *InSP* is clearly lower than for the 2 other approaches, whatever the size of the topology and the number of hosts. It is interesting to observe that *OF-responder* generates more control plane messages than *OF-unicast* when the number of nodes is high in small to medium topologies. This happens because in *OF-responder* the number of *Packet-Out* messages is proportional to the number of hosts and switches, while it is rather related to the size of the topology (number of links and switches) in the *OF-unicast* case. We also note that, while *InSP* behaves like *OF-responder* during the learning phase, it produces much less messages because no controller interactions are required once mappings are learnt.

A summary of the reduction in the number of messages obtained using InSP is shown in Fig. 11(g) to 11(i). In these figures, since *InSP* does not offer any gain in the data plane compared to *OF-responder* (cf. Table 2), we did not plot the corresponding line. When compared to *OF-unicast*, however, *InSP* saves from 63% to 91% of ARP messages for a small number of nodes in small to large topologies, respectively, and this gain converges around 67% in any case[4].

In the control plane, compared to *OF-unicast*, *InSP* saves from 58% to 96% of the messages for a small number of nodes in small to large topologies, respectively. These savings decrease to about 50%, 55% and 66% when the number of nodes grows in small, medium and large topologies, respectively. In any case, the control message savings are always above 50%. When compared to *OF-responder*, *InSP* saves 30%, 45% and 49% of the messages for a small number of nodes in small, medium and large topologies, respectively. Each of these already significant gains increases with the number of nodes up to a convergence point of 66%, whatever the size of the topology.

## 6. DISCUSSION

In this section, we discuss how we envision the design of networks that support the InSP API, consistency issues between the state at the controller and the configured PTEs and implementation options for the API in hardware switches.

**Network design**. The InSP API introduces a new function that is executed by the switch autonomously, according to some local information provided by the controller. The controller is in charge of deciding which one of the switches should generate a given set of packets. For instance, in our ARP in datacenter evaluation of Section 5, we evaluated a strawman solution for the InSP case, distributing all the PTEs for the ARP replies generation to all the edge switches. An alternative solution may decide for the installation of a PTE only in those switches that are supposed to see very frequently corresponding ARP requests. Also, one could design an algorithm in which only a subset of switches is in charge of handling packet generations. We believe that the implementation of more complex use cases will require the exploration of smart distribution strategies for the PTEs, in similar manner to what today's OpenFlow networks do with FTEs [16].

**Consistency**. The distribution of state to switches may create inconsistencies between the actual state of the network and the information stored in a switch. This is a general problem of any distributed systems, and as such it affects also switches that implement InSP. Considering again as example the ARP case, a given PTE in a switch may provide an ARP reply for a host which is not connected to the network anymore. The reason may be that the switch's PTEs where not updated yet after the host disconnected. While this may be a big issue in some cases, we believe that in no way this is different from any other distributed system. That is, any network design should take into account the possibility of incurring in stale information. In fact, for ARP this is already the case even in legacy systems (ARP caches on end hosts are not updated for tens of seconds). The InSP API does not introduce a timeout concept to help with the implementation of strategies that guarantee consistency. However, to the same purpose the FTE's timeouts can be used instead, since without a trigger (i.e., the FTE) a packet is never generated anyway.

**Implementation options**. While our software implementation shows that the InSP API is a very simple addition to both controller and software switches, we did not try to implement it in a hardware switch. However, considering that legacy switches and routers already handle messages generation, usually in the device's slow path, we believe that supporting InSP in a hardware switch should be as simple as supporting it in software switches. In fact, the implementation would actually be done in software, as part of the switch firmware that already implements the OpenFlow agent.

## 7. RELATED WORK

In this Section, we present an overview of the related work, which we organize in three categories. First we present work that deals with the problem of delegation of control, i.e., approaches that address the same issues or are complementary to those addressed by us. Then, we present work that deals with the SDN scalability, including work that handles packet generation scalability in SDN. Finally, we present an overview of switch architectures that may be of interest for the implementation of InSP in different switch technologies.

**Delegation of control**. How to distribute functions between the controller and the switch is a typical dilemma in Software-defined architectures. OpenFlow [24] introduced switches that can only deal with forwarding actions execution, leaving to the controller any decision logic. Quite early in the days of OpenFlow, the scalability concerns with such an architecture triggered several works that revisited the functions distribution. DevoFlow [25] proposes to devolve back to the switch some functions for taking fast rerouting decisions and to increase the efficiency of traffic statistics gathering. Difane [39] computes the forwarding rules distribution strategy at the controller, but delegates the action of actually distributing the rules to a subset of switches, called authority switches. In OpenState [4], the switch is enhanced with the ability to perform stateful forwarding actions. In addition to the flow programming model introduced with OpenFlow, OpenState introduces a finite state machine programming model. The model allows the controller to define flow states and state transition logic that is executed autonomously by the switch. Other approaches addressed the delegation issue exclusively on the controller side, introducing a hierarchical controller architecture [5]. The delegation decision, in this case, is about the distribution of functions between different controllers. For instance in Kandoo [12] a two layers structure is defined, with local and "remote" controllers. A local controller performs a shorter control loop with the switch but has no visibility about network-wide state, while the remote controller can take decisions knowing the network-wide state but being offloaded of the handling of local events.

**SDN scalability**. The studies of delegation of control and SDN scalability issues are intertwined. The previously mentioned hierarchical controller architectures deal with controller scalability,

[4] *OF-responder*, compared to *OF-unicast*, would offer the same gains in the data plane.

however there are other important scalability challenges in SDN switches. For instance, several production deployments of SDN pointed out that switches cannot interact with the controller at high rates. For example, OpenFlow switches can support a limited number of FTEs installations and only generate a limited number of PACKET_INs per second [14, 18]. To deal with these issues, in Tango [22] the controller is enhanced with a system that measures and takes into account several performance properties of the switches, in order to optimize the interactions of the switches with the controller. In Scotch [38], the limited switch performance in handling PACKET_INs is addressed by building and orchestrating an overlay network, which is used to move the PACKET_IN generation to auxiliary switches.

**Switch architectures**. In an effort to increase scalability and flexibility of the SDN switches, several proposals have been dealing with new switch architectures. While we already mentioned DevoFlow and OpenState, which add features to a switch, in this paragraph we present work that revisits the switch design to increase the performance of currently available switch's functions. For instance, ShadowSwitch [6] combines a software switch with a hardware switch to improve on the FTEs installation time. In general, hybrid hardware/software architectures have been proposed also to enlarge switch's buffers when required [23] and to increase flow tables' size [17]. Recently, a lot of attention has been raised by the implementation of re-configurable hardware switches [7] and by the definition of configuration languages to deal with them [37].

While all the cited works are somewhat related to the InSP API, our differs from the previous work in the field, since it is the first one that proposes and evaluates a general API to program packet generation in the switches. Furthermore, using the InSP API, we demonstrated that we are able to improve on both controller and switch scalability.

# 8. CONCLUSION

This paper presented the In-Switch Packet generation API for OpenFlow switches. The programmable in-switch generation of packets slightly redraws the separation between controller's and switch's functions, enabling the controller at offloading some of its tasks while still maintaining full control over the network according to the SDN principles. We demonstrated that the InSP API is helpful for the implementation of very common use cases, such as ARP and ICMP handling, while being beneficial to both the switch and controller scalability. In particular, the in-switch packet generation operation, implemented in our prototype, requires less resources than the handling of an interaction with the controller. Furthermore, the controller is completely offloaded of any packet generation operation. Our analytical study about the application of the InSP API, for the handling of ARP in a datacenter, also shown that the total number of control messages can be reduced from 30% to 96%, depending on the network topology and on the used OpenFlow implementation.

The InSP API has been already presented to the ONF, where we are committed to continue with the standardization of the interface. In view of that, we plan to implement a number of new use cases in the near future. A task for which we ask the help of the research community by making available our prototype implementation as open source code [1].

# Acknowledgment

This work has been partly funded by the EU in the context of the "BEBA" project (Grant Agreement: 644122).

# 9. REFERENCES

[1] Beba—behavioural based forwarding, 2015. http://www.beba-project.eu/.

[2] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[3] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[4] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, Apr. 2014.

[5] R. Bifulco, R. Canonico, M. Brunner, P. Hasselmeyer, and F. Mir. A practical experience in designing an openflow controller. In *Software Defined Networking (EWSDN), 2012 European Workshop on*, pages 61–66, Oct 2012.

[6] R. Bifulco and A. Matsiuk. Towards scalable sdn switches: Enabling faster flow table entries installation. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 343–344, New York, NY, USA, 2015. ACM.

[7] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[8] D. Erickson. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[9] N. Feamster, J. Rexford, and E. Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Comput. Commun. Rev.*, 44(2):87–98, Apr. 2014.

[10] Floodlight SDN Controller. http://www.projectfloodlight.org/floodlight/.

[11] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. *Commun. ACM*, 54(3):95–104, Mar. 2011.

[12] S. Hassas Yeganeh and Y. Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[13] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 7–12, New York, NY, USA, 2012. ACM.

[14] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 43–48, New York, NY, USA, 2013. ACM.

[15] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a

globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, Aug. 2013.

[16] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM.

[17] N. Katta, J. Rexford, and D. Walker. Infinite CacheFlow in software-defined networks. In *Proceedings of the 3rd ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN)*, 2014.

[18] M. Kobayashi, S. Seetharaman, G. M. Parulkar, G. Appenzeller, J. Little, J. van Reijendam, P. Weissmann, and N. McKeown. Maturing of openflow and software-defined networking through deployments. *Computer Networks*, 61, 2014.

[19] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 203–216, Berkeley, CA, USA, 2014. USENIX Association.

[20] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[21] M. Kuzniar, P. Peresini, and D. Kostić. Providing reliable FIB update acknowledgments in SDN. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 415–422, New York, NY, USA, 2014. ACM.

[22] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. Tango: Simplifying sdn control with automatic switch property inference, abstraction, and optimization. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 199–212, New York, NY, USA, 2014. ACM.

[23] G. Lu, R. Miao, Y. Xiong, and C. Guo. Using cpu as a traffic co-processing unit in commodity switches. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 31–36, New York, NY, USA, 2012. ACM.

[24] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[25] J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, A. R. Curtis, and S. Banerjee. DevoFlow: Cost-effective flow management for high performance enterprise networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[26] Nping. https://nmap.org/nping.

[27] OfSoftSwitch13. https://github.com/CPqD/ofsoftswitch13.

[28] ONF. Migration use cases and methods. https://www.opennetworking.org/images/stories/downloads/sdn-resources/use-cases/Migration-WG-Use-Cases.pdf.

[29] OpenDaylight Platform. https://www.opendaylight.org/.

[30] OpenFlow switch specification—version 1.0.0. Open Networking Foundation. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf.

[31] OpenFlow switch specification—version 1.1.0. Open Networking Foundation. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.1.0.pdf.

[32] OpenFlow switch specification—version 1.3.0 (wire protocol 0x04). Open Networking Foundation, 2012. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf.

[33] OpenFlow switch specification—version 1.5.0. Open Networking Foundation. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

[34] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of ACM HotNets '11*, 2011.

[35] RYU SDN framework. http://osrg.github.io/ryu/.

[36] SDN architecture, issue 1, tr-502. Open Networking Foundation, June 2014. https://www.opennetworking.org/images/stories/downloads/sdn-resources/technical-reports/TR_SDN_ARCH_1.0_06062014.pdf.

[37] A. Sivaraman, C. Kim, R. Krishnamoorthy, A. Dixit, and M. Budiu. Dc.p4: Programming the forwarding plane of a data-center switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 2:1–2:8, New York, NY, USA, 2015. ACM.

[38] A. Wang, Y. Guo, F. Hao, T. Lakshman, and S. Chen. Scotch: Elastically scaling up sdn control-plane using vswitch based overlay. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 403–414, New York, NY, USA, 2014. ACM.

[39] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable flow-based networking with difane. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 351–362, New York, NY, USA, 2010. ACM.