Efficient Topology Discovery in Software Defined Networks

Farzaneh Pakzad*, Marius Portmann†, Wee Lum Tan‡ and Jadwiga Indulska§
School of ITEE, The University of Queensland
Brisbane, Australia
Email: *farzaneh.pakzad@uq.net.au, †marius@ieee.org, ‡w.tan4@uq.edu.au, §jaga@itee.uq.edu.au

***Abstract*—Software Defined Networking (SDN) is a new networking paradigm, with a great potential to increase network efficiency, ease the complexity of network control and management, and accelerate the rate of technology innovation. One of the core concepts of SDN is the separation of the network's control and data plane. The intelligence and the control of the network operation and management, such as routing, is removed from the forwarding elements (switches) and is concentrated in a logically centralised component, i.e. the *SDN controller*. In order for the controller to configure and manage the network, it needs to have up-to-date information about the state of the network, in particular its topology. Consequently, topology discovery is a critical component of any Software Defined Network architecture. In this paper, we evaluate the cost and overhead of the de facto standard approach to topology discovery currently implemented by the major SDN controller frameworks, and propose simple and practical modifications which achieve a significantly improved efficiency and reduced control overhead. We have implemented our new topology discovery approach on the widely used POX controller platform, and have evaluated it for a range of network topologies via experiments using the Mininet network emulator. Our results show that our proposed modifications achieve an up to 45% reduction in controller load compared to the current state-of-the-art approach, while delivering identical discovery functionality.**

I. Introduction

Software Defined Networking (SDN) is a new networking paradigm which has recently gained tremendous momentum, both in terms of commercial activity as well as academic research [1], [2], [3].

One of the key concepts of SDN is the separation of the data plane from the control plane. This is in contrast to traditional IP networks, where routers perform both packet forwarding (data plane) and run routing protocols which discover network paths and make routing decisions (control plane). In SDN, control 'intelligence' is removed from the forwarding elements (routers, switches), and concentrated in a logically centralised entity called the *SDN controller*, implemented in software [1].

Logical centralisation of control does not necessarily imply physical centralisation, which can result in scalability and reliability problems, since the SDN controller would represent a single point of failure. To address this issue, researchers have proposed physically distributed SDN controllers, such as the Onix system for example [4].

[1]If some control intelligence, and how much of it, should remain in SDN switches is an issue of ongoing debate between the various SDN proponents.

Via the centralisation of network control, SDN essentially does away with traditional distributed network protocols. As a result, SDN forwarding elements, in the following simply referred to as switches, can become simpler and cheaper than traditional routers, and network configuration and management is significantly simplified.

With SDN, the network becomes much more programmable and enables a higher rate of innovation. New network services, applications and policies can simply be implemented via an application running on the controller, which controls the forwarding elements (data plane) via appropriate abstractions and a well defined API, such as OpenFlow [5]. By installing the appropriate rules, a controller application can program SDN switches to perform a wide range of functionality, such as routing, switching, firewalling, network address translation, load balancing, etc. This can be done at different layers of the protocol stack.

Another critical benefit of SDN is its ability to facilitate network virtualisation, e.g. via tools such as FlowVisor [6] or OpenVirteX [7], which is essential in many deployment scenarios, in particular in data centre applications. These and other benefits of SDN have resulted in a great amount of recent industry traction, with many established and new vendors offering an increasing number of SDN enabled switches and other devices, as well as a range of SDN controller platforms.

Figure 1 shows a logical view of the basic SDN architecture. At the bottom is the infrastructure layer, consisting of a set of interconnected forwarding elements, i.e. SDN switches, which provide the data plane functionality. Switches can be both hardware devices, or software based, virtual switches, such as Open vSwitch [8].

The next layer up is the control layer, consisting of a logically centralised, software based SDN controller. One of the key roles of the controller is to provide and maintain a global view of the network. The SDN controller provides an abstraction to the application layer, hiding a lot of the complexity of maintaining and configuring a distributed network of individual network devices. Topology discovery, the focus of this paper, is an essential service provided by the SDN control layer.

In order to implement this abstraction, the SDN controller is responsible for managing and configuring the individual SDN switches, by installing the appropriate forwarding rules. The interface between the control layer and the infrastructure layer used for this is often referred to as the *southbound* interface.

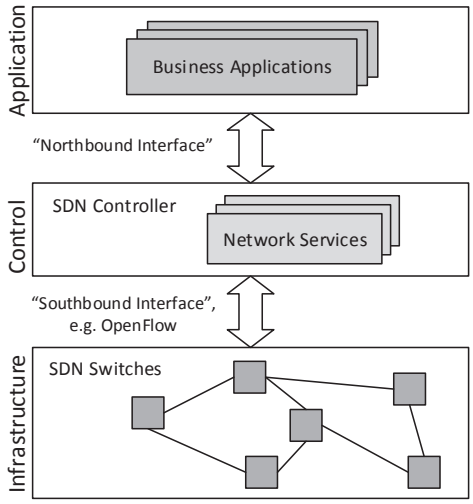

Fig. 1. SDN Architecture

The most prominent southbound interface standard is OpenFlow [5], [9], which we will assume and use in this paper. Section II will provide some more details on OpenFlow that are relevant for our discussions later in the paper.

The top layer in the SDN architecture is the application layer, where high level network policy decisions are defined and implemented. The interface between the application layer and the control layer is called the *northbound* interface. However, in contrast to the southbound interface, the definition of a standard for the northbound interface is still very much a work in progress.

In this paper we focus on topology discovery, which is a critical service provided at the control layer of the SDN architecture, and which underpins the centralised configuration and management in SDN. The contribution of the paper includes an analysis of the overhead of the current de facto standard for SDN topology discovery. We further propose an improved version, which manages to achieve the same functionality with a significantly reduced load on the controller. We present experimental evaluation results which demonstrate this.

As a basis for our following discussions, Section II provides the relevant background on OpenFlow. Section III discusses the current state-of-the-art approach to topology discovery in SDN, and Section IV presents our proposed new approach. Sections V and VI present evaluation results and conclusions respectively.

II. Background - OpenFlow

OpenFlow [9] is a standard protocol for SDN southbound interface, i.e. the interface between control layer and the infrastructure layer in Figure 1. In practical terms, OpenFlow provides a communications interface between the SDN controller and SDN switches, which allows the controller to configure and manage the switches.

While there are other protocols proposed and employed as the southbound interface, such as SNMP, BGP, PCEP etc. [10], OpenFlow, promoted by Open Network Foundation (ONF) [3], is the dominant standard.

As of writing this paper, the latest edition of the OpenFlow standard is version 1.4 [11]. OpenFlow has undergone a few changes from its initial version (1.0) to the current version. However, the differences between the various protocol versions are not significant in the context of this paper. Our discussion of the topology discovery mechanism and our proposed improvement are protocol version agnostic, and work for all versions of the protocol.

An OpenFlow enabled SDN switch is assumed to be configured with the IP address and TCP port number of its controller. On startup, a switch will contact its controller on the corresponding IP address and TCP port, and establish a Transport Layer Security (TLS) f session to secure the connection.

As part of the initial protocol handshake, using an OpenFlow *OFPT_FEATURES_REQUEST* message, the controller requests configuration information from the switch, including its active switch ports (network interfaces) and corresponding MAC addresses. We will make use of this in our proposed topology discovery method in Section IV. The initial switch-controller handshake informs the controller about the existence of the nodes (switches) in the network, but what is missing for a complete topology view is the information about the available links between switches in the network. This is addressed by the topology discovery mechanism discussed later in this paper.

OpenFlow allows a controller to access and manipulate the forwarding rules of SDN switches, and thereby control how traffic should flow through the network. An OpenFlow compliant switch needs to support the basic *match-action* paradigm, which means that each incoming packet is matched against a set of rules, and the action or action list associated with the matching rule is executed. The matching is flow-based, and relatively fine grained. The match fields supported in OpenFlow include the switch ingress port, various packet header fields such as MAC source and destination address, IP source and destination address, UDP/TCP source and destination port numbers etc. One of the key actions supported by an OpenFlow switch is forwarding a packet to a particular switch port, or dropping the packet.

Switch ports defined in the forwarding rules comprise physical ports, but also include the following virtual ports: *ALL* (sends packet out on all physical ports), *CONTROLLER* (sends to SDN controller via an OpenFlow *Packet-In* message), *FLOOD* (same as *ALL*, but excluding the ingress port).

In addition, OpenFlow also supports a number of actions which allow the rewriting of packet headers by the switch, including the updating of the TTL fields, adding or removing VLAN and MPLS tags, and the rewriting of MAC source and destination addresses, etc. We will make use of this feature in our proposed topology discovery mechanism discussed in Section IV. In this context, it is important to note that OpenFlow does not support access to and rewriting of any packet payload, due to performance reasons.

A switch can also forward data packets that it has received via any of its ports to the controller. This is done via an OpenFlow *Packet-In* message (*OFPT_PACKET_IN*). For example,

this is used whenever a switch receives a packet which does not match any forwarding rule. In this case, the default behaviour is that the switch forwards the packet to the controller encapsulated in an OpenFlow *Packet-In* message. The controller can then decide to install the appropriate forwarding rules for the corresponding new flow.

The forwarding of packets to the controller can also be achieved via dedicated rules installed on the switch. For example, in the current de facto standard SDN topology discovery mechanism, discussed in more detail below, each switch has a pre-installed rule which says that all topology discovery packets (of *EtherType* 0x88cc) are to be forwarded to the CONTROLLER port.

Complementary to the *Packet-In* primitive, OpenFlow also supports a *Packet-Out* message (*OFPT_PACKET_OUT*). Via this message, the controller can send a data packet to a switch, with instructions of what to do with it, in the form of an *action list*. For example, the controller can send a packet to a switch and instruct to send it out on a particular port. Alternatively, the controller can instruct the switch to send the packet via the *OFPP_TABLE* virtual output port, which means the packet is treated as if it was received via any of the switch's 'normal' ports, and is handled according to the normal forwarding rules (or *flow tables*) installed on the switch.

Both OpenFlow *Packet-In* and *Packet-Out* messages are essential for the topology discovery mechanisms discussed in the following sections.

III. SDN Topology Discovery - Current Approach

In order for an SDN controller to be able to manage the network and to provide services such as routing, it needs to have up-to-date information about the network state, in particular the network topology. Therefore, a reliable and efficient topology discovery mechanism is critical for any SDN system.

To be precise, when we refer to topology discovery in the following, we are really concerned with connectivity discovery or link discovery. An SDN controller does not need to discover the network nodes (switches), since it is assumed that they will initiate a connection to the controller, and thereby announce their existence.

OpenFlow switches do not support any dedicated functionality for topology discovery, and it is the sole responsibility of the controller to implement this service. Furthermore, there is no official standard that defines a topology discovery method in OpenFlow based SDNs. However, most current controller platforms implement topology discovery in the same fashion, derived from an implementation in NOX [12], the original SDN controller. This makes that mechanism the de facto SDN topology discovery standard. The mechanism is referred to as *OFDP* (OpenFlow Discovery Protocol) in [13], and for lack of an official term, we will use that name in this paper.

OFDP leverages the Link Layer Discovery Protocol (LLDP) [14]. LLDP allows nodes in a IEEE 802 Local Area Network to advertise to other nodes their capabilities and neighbours.

LLDP is typically implemented by Ethernet switches, and they actively send out and receive LLDP packets. LLDP

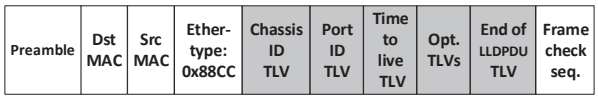

Fig. 2. LLDP Frame Structure

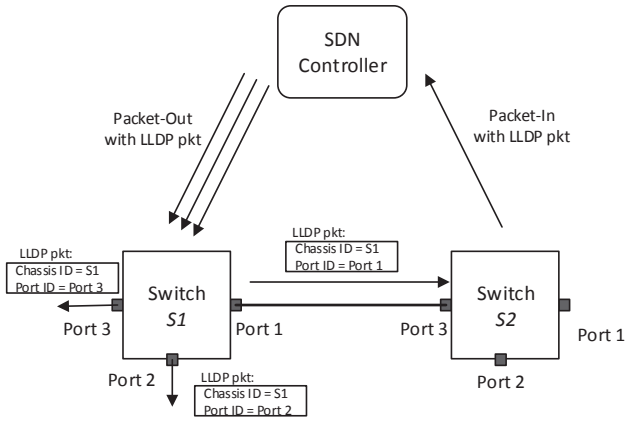

Fig. 3. Basic OFDP Example Scenario

packets are sent regularly via each port of a switch and are addressed to a bridge-filtered multicast address, and are therefore not forwarded by switches, but only sent across a single hop.

The information learned from received LLDP packets is stored by all the switches in a local Management Information Base (MIB). By crawling all the nodes in the network and retrieving the corresponding information in the MIB, e.g. via SNMP, a network management system can discover the network topology.

An LLDP payload is encapsulated in an Ethernet frame with an *EtherType* field set to 0x88cc. The frame contains an LLDP Data Unit (LLDPDU), which consists of a number of type-length-value (TLV) structures. The mandatory TLVs are *Chassis ID*, which is a unique switch identifier, *Port ID* and *Time to live*, which are self explanatory, followed by a number of optional TLVs and an *End of LLDPDU TLV*. Figure 2 shows the frame structure, with the LLDP payload highlighted.

OFDP leverages the packet format of LLDP, but otherwise operates quite differently. Given its quite narrow API and limited match-action functionality, an OpenFlow switch cannot by itself send, receive and process LLDP messages. This needs to be initiated and executed entirely by the controller. The process is illustrated in a very simple scenario shown in Figure 3.

First in this scenario, the SDN controller creates an individual LLDP packet for each port on each switch, in this case, a packet for *Port 1*, one for *Port 2* and one for *Port 3* on switch *S1*. Each of these three LLDP packets has the *Chassis ID* and

Port ID TLVs initialised accordingly. [2]

The controller then sends each of these three LLDP packets to switch *S1* via a separate OpenFlow *Packet-Out* message, with the included instruction to send the packet out on the corresponding port. For example, the LLDP packet with *Port ID = Port 1* is to be sent out on *Port 1*, the packet with *Port ID = Port 2* on *Port 2* and so forth.

All switches have a pre-installed rule in their flow table which says that any LLDP packet received from any port except the CONTROLLER port, is to be forwarded to the controller, which is done via an OpenFlow *Packet-In* message.

In our example in Figure 3, we consider the LLDP packet which is sent out on *Port 1* of switch *S1* and is received by switch *S2* via *Port 3*, via to the corresponding link.

According to the pre-installed forwarding rule, switch *S2* forwards the received LLDP packet to the controller via a *Packet-In* message. This *Packet-In* message also contains meta data, such as the ID of the switch and the ingress port via which the packet was received. From this information, and from information contained in the payload of the LLDP packet, i.e. the *Chassis ID* and *Port ID* TLVs, the controller can now infer that there exists a link between *(S1, Port 1)* and *(S2, Port 3)*, and will add the information to its topology database.

The process is repeated for every switch in the network, i.e. the controller sends a separate *Packet-Out* message with a dedicated LLDP packet for each port of each switch, allowing it to discover all available links in the network. The entire discovery process is performed periodically, with a new discovery *round* or *cycle* initiated in fixed intervals, with a typical default interval size of 5 seconds.

Most current SDN controller platform implement this topology discovery mechanism (OFDP), such as NOX [12], POX [15], Floodlight [16], Ryu [17], Beacon [18], etc. Looking at the source code of the topology discovery implementations reveals some very minor variations, mostly in regards to the timing of message sending. POX (and NOX) will spread the sending of all the LLDP *Packet-Out* messages equally over the discovery interval. In contrast, Floodlight sends all these messages back-to-back at the beginning of the discovery interval, while Ryu adds a small constant gap between messages.

A. Controller Overhead of OFDP

Controller load and performance is critical for the scalability of a Software Defined Network [19]. Since topology discovery is a service that typically runs continuously in the background on all SDN controllers, it is important to know the load it imposes on the controller. The controller load due to OFDP is determined by the number of LLDP *Packet-Out* messages the controller needs to send and the number of LLDP *Packet-In* messages it receives and needs to process.

The number of LLDP *Packet-In* messages (P_{IN_OFDP}) received by the controller in a single discovery round depends on the network topology, and is simply twice the number of active inter-switch links in the network, one packet for each link direction.

The total number of LLDP *Packet-Out* messages (P_{OUT_OFDP}) a controller needs to send per OFDP discovery round is the total number of ports in the network. As discussed earlier in this section, the controller needs to send a dedicated LLDP packet with corresponding *Port ID* and *Chassis ID* to each individual port.

With L being the number of links between switches in the network, N the number of switches, and p_i the number of ports of switch i, we can express this as follows:

$$P_{IN\_OFDP} = 2L \tag{1}$$

$$P_{OUT\_OFDP} = \sum_{i=1}^{N} p_i \tag{2}$$

Sending an LLDP *Packet-Out* message for each port on each switch seems inefficient. A better alternative would be to only send a single *Packet-Out* message to each switch, and ask it to send the corresponding LLDP packet out on all its ports. This functionality is supported in OpenFlow.

The problem is that each LLDP packet needs to have the *Port ID* TLV initialised to the corresponding switch egress port. This is required so that the controller, upon receiving the LLDP packet via a *Packet-In* message from the receiving switch, can determine the source port of the discovered link. (As is illustrated in Figure 3.)

The current way to achieve this in OFDP, is for the controller to prepare and send a dedicated LLDP packet via a separate *Packet-Out* message for each port of every switch.

A solution for the problem would be if we were able to instruct the switch to rewrite the LLDP *Port ID* TLV on-the-fly, according to the port the packet is being sent out on. However, this is not possible, since OpenFlow switches do not support access to and rewriting of packet payload data. In the following section, we present a solution to this problem.

IV. Proposed Improvement - OFDPv2

In order to achieve the goal of reducing the number of LLDP *Packet-Out* messages to one per switch, we use the fact that upon connection establishment between an OpenFlow switch and controller, the switch informs the controller about its ports, their Port IDs and the associated MAC addresses, in response to an OpenFlow *OFPT_FEATURES_REQUEST* message. The controller therefore has a one-to-one mapping of MAC addresses and *Port ID*s for each switch.

The second feature that we use is the ability of OpenFlow switches to rewrite packet headers. This is typically used to update TTL fields, implement Network Address Translation, etc. In particular, we will use the fact that switches can rewrite source MAC addresses of outgoing Ethernet frames.

Our proposed new SDN topology discovery method, which we call *OFDPv2*, involves the following changes to the current version (OFDP):

[2] Any other unique switch identifier could be used instead of the *Chassis ID*, and could for example be included as an optional TLV.

1) We modify the controller behaviour to limit the number of LLDP *Packet-Out* messages sent to each switch to one. The *Port ID* TLV field in the LLDP payload is set to 0, and will be ignored.

2) We install a new rule on each switch, which specifies that each LLDP packet received from the controller is to be forwarded on all available ports, and that the source MAC address of the corresponding Ethernet frame is to be set to the address of the port via which it is sent out (Algorithm 1).

3) Finally, we modify the *Packet-In* event handler on the controller, which processes incoming LLDP packets. Instead of parsing the *Port ID* TLV of the LLDP payload, we now look at the source MAC address of the Ethernet header and lookup the corresponding *Port ID* in the controller's database, which has a one-to-one mapping of MAC addresses and switch *Port ID*s.

As mentioned, Algorithm 1 shows the modified LLDP packet processing on each switch, i.e. it shows the installed match-action rule.

If an incoming packet is an LLDP packet (*EtherType*=0x88cc), and is received from the controller via a *Packet-Out* message (inPort = CONTROLLER), then a copy of the packet is sent out on each switch port (line 5), after the source MAC address has been set to the MAC address of the port on which it is to be sent out on (line 4).

OpenFlow does not support a loop construct such as used in Algorithm 1. We therefore need to 'unroll' the loop, and install a specific action list for each switch port.

Algorithm 1 OFDPv2 LLDP Packet Processing at Switch

```
1: for all received packets pkt do
2:   if pkt.etherType=LLDP and pkt.inPort=CONTROLLER
     then
3:     for all switch ports P do
4:       pkt.srcMAC ← P.MACaddr
5:       send copy of pkt out on port P
6:     end for
7:   end if
8: end for
```

V. Evaluation

We have implemented our proposed topology discovery mechanism (OFDPv2) in Python on the POX SDN controller platform [15]. Our implementation is based on *discovery.py*, POX's implementation of OFDP, which we also used as a benchmark for comparison.

We performed extensive tests for a wide range of network topologies, to establish the functional equivalence of OFDP and OFDPv2. As expected, both versions were identical in regards to their ability to discover active links in the network. (Details about our experiments are provided below in Section V-A.)

The main purpose of our evaluation was to establish by how much OFDPv2 can reduce the controller overhead compared to OFDP, the current de facto standard. A key measure of the overhead imposed on the controller is the number of control message it needs to handle.

There is no difference between OFDP and OFDPv2 in regards to the number of LLDP packets that are received by the controller via OpenFlow *Packet-In* messages. Irrespective of the version of OFDP, this number is simply twice the number of active inter-switch links L in the network. Therefore, based on Equation 1, we have:

$$P_{IN\_OFDPv2} = P_{IN\_OFDP} = 2L \tag{3}$$

We consequently focus on the number of LLDP packets that are sent out by the controller via OpenFlow *Packet-Out* messages in each round of the topology discovery process. As discussed in Section III and shown in Equation 2, in OFDP the controller sends out a separate LLDP packet for each port on each switch in the network.

The key advantage of our modifications in OFDPv2 is that the number of LLDP *Packet-Out* messages is reduced to only one per switch, or N in total, with N being the number of switches in the network, i.e. we have:

$$P_{OUT\_OFDPv2} = N \tag{4}$$

We define the *efficiency gain* G in terms of the relative number of *Packet-Out* control message reduction of OFDPv2 versus OFDP. For a network with N switches, and p_i ports for a switch i, this can be expressed as follows:

$$\begin{aligned} G &= \frac{P_{OUT\_OFDP} - P_{OUT\_OFDPv2}}{P_{OUT\_OFDP}} \\ &= \frac{\sum_{i=1}^{N} p_i - N}{\sum_{i=1}^{N} p_i} = 1 - \frac{N}{\sum_{i=1}^{N} p_i} \end{aligned} \tag{5}$$

We see that the gain is greater for networks with a higher number of total ports, i.e. it is higher for a network with a higher average switch port density. We will verify this via experiments using a number of example topologies.

A. Experimental Setup

For our experimental evaluation, we used the Linux based Mininet network emulator [20], which allows the creation of a network of virtual SDN switches and hosts, connected via virtual links. We further used Open vSwitch [8], a popular virtual (software based) SDN switch with support for OpenFlow. We ran Mininet in the VirtualBox virtualisation software under Linux as the host operating system.

As previously mentioned, we used POX as our SDN controller platform, and therefore implemented our proposed changes to the SDN topology discovery mechanism in Python. Table I summarises the software that we have used for our prototype implementation and in all our experiments.

All experiments were run on a Dell PC (OptiPlex 9020) with an Intel i7-4770 CPU, running at 3.40GHz, with 16GB of RAM.

TABLE I. SOFTWARE USED IN IMPLEMENTATION AND EXPERIMENTS

Software	Function	Version
Mininet [20]	Network Emulator	2.0.0
Open vSwitch [8]	Virtual SDN Switch	1.4.3
POX [15]	SDN Controller Platform	*betta* branch
Oracle VM VirtualBox	x86 Virtualization Software	4.3.10
Linux (Ubuntu)	Host Operating System	12.10
Python	Programming Language	2.7

TABLE II. EXAMPLE NETWORK TOPOLOGIES AND KEY PARAMETERS

	Topology	# Switches	# Ports
Topology 1	Tree, $d = 4$, $f = 4$	85	424
Topology 2	Tree, $d = 7$, $f = 2$	127	380
Topology 3	Linear, $N = 100$	100	298
Topology 4	Fat Tree	20	80

We considered four network topologies of switches and hosts in our experiments, two basic tree topologies, a simple linear topology and a fat tree topology. Key parameters of these four topologies, in particular the number of switches and ports, are shown in Table II.

Topology 1 is a tree topology with fanout $f = 4$ and depth $d = 4$. Topology 2 is also a tree topology, but with $f = 2$ and $d = 7$.

In these two tree topologies, hosts form the bottom layer of the tree. The basic idea is illustrated in Figure 4, showing a smaller and easier to visualise example of such a tree topology with $f = 3$ and $d = 3$. Switches are shown as (rounded) squares, and hosts are ovals.

Switches at the second layer from the bottom will still send out LLDP packets on all their ports, including the ones connected to hosts. However, hosts do not understand LLDP or OpenFlow, and will simply ignore any LLDP packet they receive as part of the topology discovery process. [3]

Topology 3 is a simple linear topology of N=100 switches, with a host attached to each switch.

Finally, Topology 4 is a small fat tree [21], as often used in data centre networks. The topology has 20 switches and a total of 80 switch ports, and is shown in Figure 5. As in the other tree topologies, hosts are attached to the switches at the bottom layer of the topology.

As can be seen from Equations 1 to 5, the specific type of the topology is not really relevant for our discussion, the key parameters are the number of switches and ports in the network.

B. *Number of* Packet-Out *Control Messages*

In a first experiment, we instrumented the POX controller to collect statistics about the number of *Packet-Out* messages sent by the topology discovery component in each discovery cycle, for each of our four example topologies. We ran each experiment 10 times, with identical results, as expected.

[3]The discovery of hosts in a SDN network is a separate issue, not addressed by the topology discovery mechanism and therefore beyond the scope of this paper.

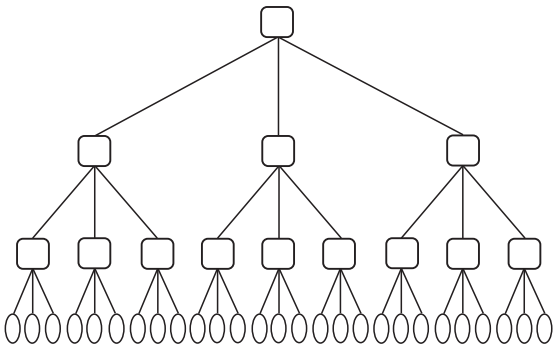

Fig. 4. Tree Topology with depth 3 and fanout 3

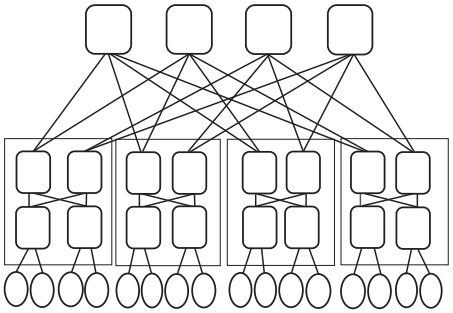

Fig. 5. Fat Tree Topology

TABLE III. NUMBER OF LLDP *Packet-Out* CONTROL MESSAGES

	OFDP	OFDPv2	Efficiency Gain G
Topology 1	424	85	80%
Topology 2	380	127	67%
Topology 3	298	100	67%
Topology 4	80	20	75%

Table III shows the measured results, as well as the relative reduction in the number of control messages, i.e. the efficiency gain G of OFDPv2 over OFDP, as defined in Equation 5.

We see that the experimental results correspond to Equations 2 and 4 and the relevant parameters for the various topologies. For example, since Topology 1 has 85 switches and 424 ports, OFDP requires 424 LLDP *Packet-Out* messages compared to the 85 of OFDPv2, as expected.

Figure 6 shows a graphical representation of these experiment results. It is evident that OFDPv2 achieves a great reduction in required LLDP *Packet-Out* control messages, with up to 80% fewer messages for Topology 1, and a minimum reduction of 67% for Topologies 2 and 3.

As per Equation 5, the degree of efficiency gain of OFDPv2 over OFDP simply depends on the total number of ports and switches in the network, and no other topology characteristics.

In the next experiment, we will explore how this reduction in control messages impacts the CPU load of the controller.

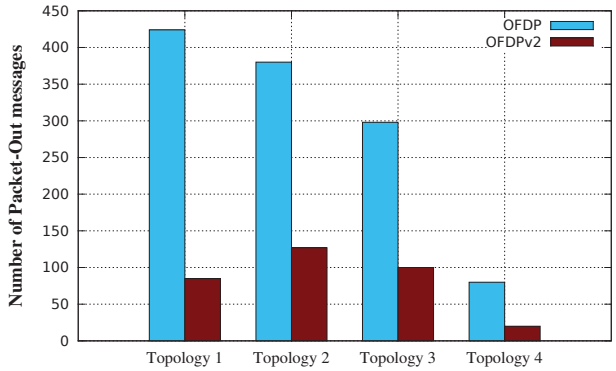

Fig. 6. Number of Packet-Out Messages

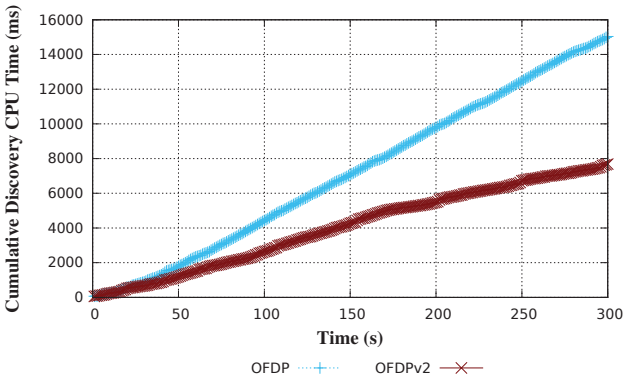

Fig. 7. Cumulative CPU Time of Topology Discovery

C. Impact on Controller CPU Load

Controller load is critical for any SDN application and is a key factor for network scalability and performance [19].

In this experiment, we were interested in how the reduction in LLDP *Packet-Out* control messages achieved in OFDPv2 reduces the CPU load imposed by the SDN controller's topology discovery service.

In our experiment, we continuously ran the topology discovery service, initiating a new discovery round every 5 seconds, which is the default interval in POX. No other service or application was running at the controller, which meant that the CPU load caused by the POX process is a good indication of the load of the topology discovery service.

We start our measurements after the initial network initialisation, e.g. the establishment of all switch-controller connections and handshakes, have been completed. The duration of each experiment was 300 seconds.

We used the *get_cpu_times()* method from *psutil*, a cross-platform process and system utilities module for Python [22].

Figure 7 shows the cumulative CPU time consumed by the POX controller running only the topology discovery module for both OFDP and OFDPv2. The figure shows the results of a single run of the experiment for Topology 1. The CPU time is plotted in 1 second intervals.

We see that in this scenario, OFDPv2 achieves an almost 50% reduction in CPU load compared to OFDP. This indicates that the processing and sending of LLDP *Packet-Out* messages is a significant component of controller CPU load of the topology discovery service, and a reduction of these messages directly results in a lower load for the controller.

We further notice that for both versions of OFDP, the cumulative CPU time relatively gradually and smoothly increases, indicating that there are no major bursts of CPU activity. This is partly due to the fact that in POX, all LLDP *Packet-Out* messages are evenly spread out over the discovery interval.

We repeated the same experiment 20 times for each of our four example topologies. Figure 8 shows the total CPU time used by the topology discovery process over the entire duration of the experiment (300 seconds). The figure shows the average over the 20 experiment runs and the 95% confidence interval.

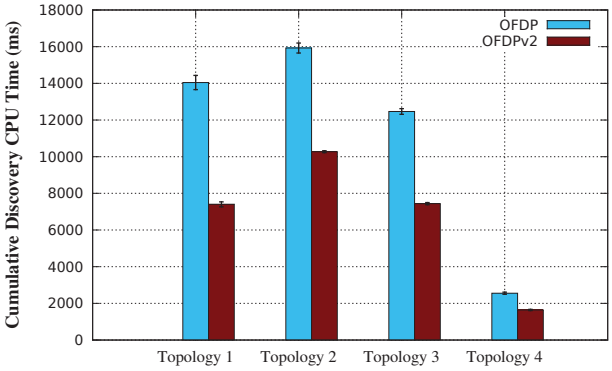

Fig. 8. Cumulative CPU Time of Topology Discovery

In summary, we observe a reduction in CPU time and load, ranging from a minimum of 35% for Topology 2 up to 45% for Topology 1, and potentially greater for networks with a higher port density.

While this is less than the 67% to 80% reduction of the number of LLDP *Packet-Out* messages achieved by OFDPv2, as shown in Table III, this is as expected. OFDPv2 only achieves a reduction of the number of LLDP *Packet-Out* messages, whereas the number LLDP *Packet-In* messages, the other type of topology discovery control messages, is unchanged.

However, a CPU load reduction of up to 45% for a central component of any SDN architecture is a significant improvement over the state-of-the-art.

There are further benefits of OFDPv2 over the current method, which we have not quantified in this paper. For example, by reducing the number of *Packet-Out* messages sent by the SDN controller, the traffic on the controller-switch channel is significantly reduced. This is particularly relevant for SDN scenarios with an in-band controller. Furthermore, the load on the SDN switches is also lowered, since instead of handling multiple OpenFlow *Packet-Out* messages, each switch only needs to handle a single message per discovery cycle in OFDPv2. The flooding of the corresponding LLDP packets via all the switch ports is implemented via standard OpenFlow datapath pipeline processing, which is typically much more efficient.

VI. Conclusions

In this paper, we have addressed the issue of topology discovery in OpenFlow based Software Defined Networks. Topology discovery is a key component underpinning the logically centralised network management and control paradigm of SDN, and is a service provided by all SDN controller platforms.

We have discussed OFDP, the current de facto standard for SDN topology discovery. OFDP has been implemented by NOX, the original SDN controller, and has since been adapted by most if not all key SDN controller platforms.

We have analysed the overhead of OFDP in terms of controller load, and have proposed and implemented an improved version, which we informally call OFDPv2. Our modified version is identical in terms of discovery functionality, but achieves this with a significantly reduced number of control messages that need to be handled by the controller and SDN switches. Via experiments, we have demonstrated that our changes can reduce the topology discovery induced CPU load on the SDN controller by up to 45% in our considered example topologies, and potentially more for other topologies with a higher node degree or port density.

Given that the controller is often the performance bottleneck of a Software Defined Network, making a core service such as topology discovery more efficient can have a significant impact on the overall network performance and scalability.

Our proposed changes are compliant with the OpenFlow standard. They are also simple and very practical, and can be implemented with relatively minimal effort, as outlined in this paper.

Finally, we are not aware of any related works that have analysed the overhead of topology discovery in SDN, or have proposed any improvements to the current state-of-the-art.

References

[1] N. McKeown, "Software-defined networking," *INFOCOM keynote talk*, 2009.

[2] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *Queue*, vol. 11, no. 12, p. 20, 2013.

[3] *Open Networking Foundation*. [Online]. Available: https://www.opennetworking.org

[4] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *OSDI*, vol. 10, 2010, pp. 1–6.

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

[6] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[7] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, W. Snow, and G. Parulkar, "Openvirtex: A network hypervisor," *Open Networking Summit*, 2014.

[8] *Open vSwitch*. [Online]. Available: http://openvswitch.org

[9] *Open Flow Standard*. [Online]. Available: https://www.opennetworking.org/sdn-resources/onf-specifications/openflow

[10] *OpenDaylight*. [Online]. Available: http://www.opendaylight.org/project/technical-overview

[11] *Open Flow Standard version 1.4*. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf

[12] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1384609.1384625

[13] *GENI Wiki*. [Online]. Available: http://groups.geni.net/geni/wiki/OpenFlowDiscoveryProtocol

[14] *IEEE Standard for Local and Metropolitan Area Networks - Station and Media Access Control Connectivity Discovery*, IEEE Std 802.1AB, 2009.

[15] *POX SDN Controller*. [Online]. Available: http://www.noxrepo.org/pox/about-pox/

[16] *Floodlight SDN Controller*. [Online]. Available: http://www.projectfloodlight.org/floodlight/

[17] *Ryu SDN Controller*. [Online]. Available: http://osrg.github.io/ryu/

[18] D. Erickson, "The beacon openflow controller," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 13–18.

[19] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, vol. 54, 2012.

[20] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[21] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *Computers, IEEE Transactions on*, vol. 100, no. 10, pp. 892–901, 1985.

[22] *psutil*. [Online]. Available: https://code.google.com/p/psutil/