IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received October 17, 2021, accepted November 1, 2021, date of publication November 8, 2021,
date of current version November 22, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3125716

One-Pass In-Band Automatic Bootstrapping for OpenFlow Switches

CHIEN-YUNG LI[1], LI-HSING YEN[1], (Member, IEEE), KUANG-HUI CHI[2], (Senior Member, IEEE), AND CHIEN-CHAO TSENG[1], (Member, IEEE)

[1]Department of Computer Science, College of Computer Science, National Yang Ming Chiao Tung University, Hsinchu 30010, Taiwan
[2]Department of Electrical Engineering, National Yunlin University of Science and Technology, Douliu 64002, Taiwan

Corresponding author: Chien-Chao Tseng (cctseng@cs.nctu.edu.tw)

This work was supported in part by Center for Open Intelligent Connectivity from the Featured Areas Research Center Program within the Framework of the Higher Education Sprout Project by the Ministry of Education, Taiwan; in part by the Ministry of Science and Technology, Taiwan, under Grant 109-2221-E-009-077, Grant 110-2221-E-A49 -044 -MY3, Grant 110-2221-E-A49 -064 -MY3, and Grant 110-2224-E-011-002; and in part by the Ministry of Economic Affairs, Taiwan, under Grant 107-EC-17-A-02-S5-007.

ABSTRACT An SDN switch newly added to a network needs to establish a control channel with an SDN controller to manage control-plane traffic. The setup of such a channel is termed bootstrapping. In the case of in-band control, bootstrapping involves setting up a control path that may traverse one or more switches between the new switch and the controller, which is achieved through the configuration of layer-two to layer-four parameters on relevant switches. Previous approaches either result in lengthy bootstrapping time due to their level- by-level mechanisms, demand complicated modules, or lack some essential features. This paper proposes an approach to fast automatic bootstrapping that overlaps the bootstrapping processes among a set of switches. Our emulations using virtual switches completed the process of bootstrapping 50 switches arranged in a chain in 2.0 seconds, which is a 98% time reduction compared with a prior study.

INDEX TERMS SDN, OpenFlow, in-band control, bootstrapping.

I. INTRODUCTION

Software-defined networking (SDN) is a trend in network management that decouples the control plane from the data plane. In an SDN network, a server called the *controller* manages SDN switches through control messages. OpenFlow (OF) is a de facto standard which defines the communication between the controller and switches under its management. When a new OF switch is put into an SDN network, it needs to establish a control channel connected to the controller. The control channel is for control-plane traffic only and has two classifications: out-of-band and in-band. An out-of-band channel uses a dedicated physical link between a switch and a controller; it is not always feasible (e.g., in a widely distributed network) or cost-effective. On the other hand, an in-band control channel demands no additional physical links, using existing ones because the control-plane traffic is simply intermixed with data-plane traffic in the SDN network. If the new switch has a direct link to the controller, the link will serve as the in-band control channel. Otherwise, the control channel between the switch and the controller need to be realized as a *control path* traversing one or multiple other switches.

Establishing a control path requires configurations on not only the OF switches and the controller but also the intermediate switches, and possibly other assisting servers. A manual configuration on all relevant switches and the controller is usually time consuming and error-prone. Therefore, the process of automatic configuration called *bootstrapping* [1] is more desirable. Bootstrapping consists of the setup of a bidirectional layer-two (L2) connection between the switch and the controller, the acquisition of a layer-three (L3) identifier for the switch, and the creation of layer-four (L4) connectivity for an OF session between the switch and the controller.

There have been some studies on automatic bootstrapping mechanisms. Many approaches perform bootstrapping in a *level-by-level* manner [1]–[6]. In these approaches, a switch can only begin its configuration process once a neighboring switch is configured. Consequently, the time it takes to bootstrap a batch of switches is contingent on the maximal length of the control path. Refer to Fig. 1a and Fig. 1b for an illustration. An experimental study in [2] reported that it took two seconds to bootstrap one switch,

The associate editor coordinating the review of this manuscript and approving it for publication was Haipeng Yao.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

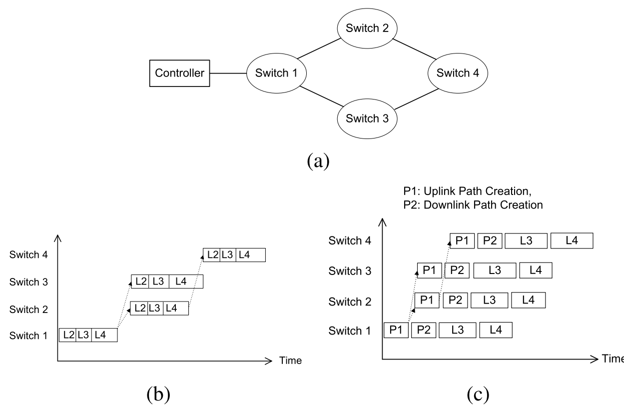


FIGURE 1. (a) A four-switch topology (b) Running of level-by-level bootstrapping (c) Running of the proposed approach.

so a level-by-level approach spent 100 seconds configuring 50 switches arranged in a chain. There are other approaches that do not operate level-by-level, but these approaches also have weaknesses. Some designs demand the installation of complicated modules [7], some likely cause performance degradation due to their L2 designs [8], and some do not consider the binding between L2 and L3 identifiers (i.e., MAC and IP addresses) of the switch [9].

In this paper, we propose a fast and automatic bootstrapping mechanism for a set of OF switches, named One-Pass In-Band Automatic Bootstrapping (One-Pass IBAB). This approach allows for the overlap of bootstrapping processes among switches so as to reduce overall configuration time. More explicitly, One-Pass IBAB allows a switch to commence its configuration process as soon as one of its neighboring switches has created a uplink control path to the controller (refer to Fig. 1c). It can thus shorten significantly time to bootstrap a batch of switches. We used virtual switches to emulate the execution of One-Pass IBAB on various network topologies. It took merely 2.0 seconds to bootstrap a 50-switch chain, which is a 98% time reduction compared with [2]. Moreover, One-Pass IBAB requires even less time in other topologies than in chain topologies. We also report on the topologies of all the final control paths in two grid-based physical topologies.

The rest of this paper is organized as follows. Sec. II briefs on background and related work. Sec. III details the proposed bootstrapping approach. Numerical results are presented in Sec. IV, and the last section concludes this paper.

II. BACKGROUND AND RELATED WORK

A. BACKGROUND

An OpenFlow switch processes an incoming packet based on the flow rules specified in its flow table — for example, dropping it, forwarding it to some port, or to the controller. A new switch with an empty flow table will drop all incoming packets by default. Therefore, when setting up an in-band control channel, all existing designs assume that default/hidden flows will be automatically added to an OF switch when it boots up. These default/hidden flows are used to forward in-band control messages to set up control paths. OpenFlow also defines a virtual port in every OF switch (the *local port* [1]) for control-plane communication between the switch and a remote entity (i.e., the controller). Any packet forwarded to the local port will be processed by a local network stack in the switch.

Typical local network stacks include OF agent, Dynamic Host Configuration Protocol (DHCP) client, and the agent for bootstrapping. The goal of bootstrapping is to establish a control path from the local port to the controller, which involves the setup of an L2 connection, the acquisition of an L3 identifier, and the creation of L4 connectivity for the OF session between the switch and the controller. OF switches use IP and Transmission Control Protocol (TCP) as L3 and L4 protocols, respectively. The controller and a neighboring OF switch can help a new OF switch set up its L2 connection with the controller. If the new switch is a hybrid switch that supports both OF and traditional Ethernet switching, the switch can also use the *normal mechanism* [1] (i.e., Ethernet switching) for L2 connection. For Ethernet switching, switches use *MAC learning* to learn of the bindings between their ports and the MAC addresses of the controller and other switches in the network. These bindings are kept in switching tables to facilitate frame forwarding for the controller and other switches. Note that the physical topology connecting these switches may contain cycles. The existence of a cycle will induce a broadcast storm, which occurs when a switch disseminates a broadcast frame or forwards a unicast frame with a previously-unseen destination MAC address. Therefore, whenever Ethernet switching is used, it should be coupled with the Spanning Tree Protocol (STP) [10] to maintain a loop-free logical tree on top of the physical topology; a broadcast storm can be avoided by confining frame forwarding to the tree. The downside is that STP usually takes considerable time to converge.

The controller may itself manage its L2 connection with a new switch (switch A for the purpose of this discussion) if the switch has a physical link to either the controller or a neighboring OF switch (switch B) that has already been configured. Specifically, switch A may broadcast an L2 frame firstly to the controller or switch B for two reasons:

- A has yet to configure an IP address so A requests this by broadcasting a DHCP Discover message;
- Or, A somehow acquires the IP addresses for both itself and the controller but does not know the controller's MAC address, so A broadcasts an ARP (Address Resolution Protocol) request message.

If this broadcast frame is received by the controller, it will thereby detect the presence of this new switch. If the broadcast frame is received by switch B, the message will not match any rule in B's flow table since A is a new addition. Therefore, B will forward the message to the controller using a packet-in message. Either way, the controller will find a route from A to the controller. It will also instruct switches along the route to add flow rules for the delivery

of A's message, if need be. If the controller manages its L2 connection with a new switch this way (e.g., [6]), the control path cannot be set up unless and until one of the switch's neighboring switch gets configured — an action also managed by the controller. Consequently, the controller will configure switches in a level-by-level manner.

Besides an L2 connection to the controller, a switch should also configure its L3 and L4 parameters for the OF session with the controller. We do not need an L3 routing protocol because the switch and the controller are in the same network segment. However, for identification purposes, the switch still needs to configure an IP address for itself and acquire the controller's IP address. It may also need to configure other L3 parameters (the IP address of the gateway, the subnet mask, etc.) for compatibility or other concerns.

It is common that a DHCP server is running at the controller to assign IP addresses to hosts. In that case, a new switch may also use DHCP to configure its IP setting and, with an extension to DHCP, obtain the controller's information, including its IP address and transport-layer parameters such as protocol type and port number. If this is the case, the switch should also establish an L2 connectivity with a DHCP server. Because the switch may not have a direct link to a DHCP server, other switches (both configured and not, possibly with the intervention of the controller) should cooperatively forward or relay DHCP-related messages between the switch and the DHCP server. If the new switch's L2 connection with the controller has been set up, a simple approach is to run a DHCP server in the controller as a controller application. This approach is feasible even if there is no DHCP server in the switch's broadcast domain, because the controller could act as a DHCP relay for the switch's requests.

After acquiring all the necessary controller information, the switch then proceeds to create a TCP connection with the controller, upon which an OF session between the switch and the controller is established. The controller may then perform authentication and send out probe message to explore the network topology.

An OF switch becomes *fully-configured* after the establishment of an OF session between itself and the controller. A fully-configured OF switch could then assist the controller in bootstrapping other switches. Before a switch becomes fully-configured, any data packet received by the switch will be dropped by default.

B. RELATED WORK

OpenVSwitch (OVS) is a virtual switch that supports OpenFlow protocol. Though an OVS is usually used to simulate an OF switch in an SDN network, it can be embedded to a real switch as an OF agent that communicates with the controller. To perform bootstrapping, the OVS installs hidden flows on the switch to run normal mechanisms for L2 to L4 connectivities with the controller. It then creates an OF session with the controller. The OVS does not specify how it obtains the controller's IP address, so the network administrator may need to manually configure this in every OVS.

Sharma *et al.* [2] and [1] were the first to propose an in-band automatic bootstrapping mechanism for OF networks. They used DHCP to assign IP addresses to switches and extended it to include controller-related information, such as the controller's IP address, transport-layer protocol, and port number. When new switches broadcast DHCP Discover messages, only those directly connected to a DHCP server can get configured. Afterwards, other switches directly connected to these switches can be configured.

Schiff *et al.* [3] and [4] assumed that any new switch is pre-configured with its own IP address and is aware of the controller's anycast addresses. When a new switch broadcasts an ARP request (with the controller's anycast address), a neighboring switch that is already connected to the controller will forward the request to the controller. The controller can then set up flow rules in the intermediate switches for the new switch. This approach were also adopted in [5], [6].

All the aforementioned procedures work in a level-by-level manner. The key to overlapped bootstrapping is to initiate the establishment of a control path between a new switch and the controller or DHCP server without fully-configured intermediate switches.

Katiyar *et al.* [7] proposed using pre-configured VLAN to lay a path (consisting of possibly non-SDN switches) from a new SDN switch to a DHCP-based configuration server. When the server receives DHCP Discover messages from a switch, other modules will locate the switch's location and configure all intermediate switches between the controller and the new switch. This scheme requires some prior configuration in intermediate switches to ensure the connectivity and allow the controller to handle MAC address resolution. Another bootstrapping approach in [8] uses normal mechanisms to deliver DHCP/ARP/TCP-SYN messages. Once a switch has been configured, the controller installs flow rules onto the switch to force it to forward subsequent ARP requests to the controller. Though this approach provides the necessary L2 connectivity, it will either cause a broadcast storm or suffer from long convergence time caused by STP.

Lopez-Pajares *et al.* [9] proposed an automatic bootstrapping scheme that assigns hierarchical labels (i.e., MAC addresses) to switches to facilitate frame forwarding on the control plane. The scheme assumes software-configurable MAC addresses and assigns one or more MAC addresses to each switch. Each MAC address consists of a fixed number of fields that collectively encode a possible control path from the controller to the switch. MAC address assignment is done by flooding a special message. This approach provides resilience against link failure by maintaining multiple control paths from a switch to the controller. However, this design induces switch overhead, and the length of the control path is limited by the number of fields in the MAC address. The dynamic binding of IP and MAC addresses may also create issues due

to inconsistency when a switch changes its MAC address as a response to a link failure. In fact, Lopez-Pajares *et al.* did not discuss how to perform configuration of (possibly dynamic) IP addresses.

A fundamental part of bootstrapping is to create a control path between every switch and the controller. The control paths may collectively form a ring structure [11] or, more commonly, a spanning tree rooted at the controller. Such a spanning tree has been exploited for other purposes such as topology discovery [12], [13] and fast flow rule installation [14]. Goltsman *et al.* [15] studied how to pre-calculate recovery paths for a spanning tree to handle possible link and switch failures. Their approach could be complementary to One-Pass IBAB as it also creates a spanning tree rooted at the controller for control paths.

III. PROPOSED APPROACH: ONE-PASS IBAB

This section elaborates on One-Pass IBAB, the proposed bootstrapping design for SDN switches. Each switch will undergo four phases in the proposed approach. The first is to identify its root port and create an uplink control path to the controller. The second is to create a downlink control path from the controller to it. The third phase is executing DHCP, and the last is creating an OF session with the controller.

A. ASSUMPTIONS AND REQUIREMENTS

We assume an OF controller and a set of switches to bootstrap which are in the same network domain as the OF controller. The proposed approach does not demand any modification on the OF controller but the ability to run a third-party controller application or service. Each switch as well as the controller has a unique MAC address used in the control plane. All switches support normal mechanisms, which means each switch has switching tables and ARP tables in addition to flow tables. Switches can turn on or off MAC learning during bootstrapping but do not need STP. Every switch uses DHCP to configure its own IP address and other IP-layer parameters (e.g., the IP address of the gateway and the subnet mask). We also assume a DHCP server running on the controller. Each switch by default runs a DHCP client to communicate with the DHCP server.

We designate a default flow rule to be pre-configured in new OF switches to override normal pipeline processing. This default rule instructs the switch to forward all incoming IBAB messages to its local port and drop all other messages.

There is an IBAB server running on the controller to disseminate controller information and discover switches. For each new OF switch, we require an agent that serves as an IBAB client to exchange bootstrapping-related control messages with the IBAB server. The IBAB client is a part of the local network stack in addition to the DHCP client and OF stack.

B. PHASE 1A: IDENTIFYING ROOT PORT

Every switch in IBAB needs to first identify its *root port*. The root port is the switch's portal to the controller. According to the default rule, prior to setting up its root port, a new switch will drop all incoming messages except IBAB messages.

To help switches identify their root ports, the IBAB server proactively disseminates IBAB Adv messages using con strained flooding. IBAB Adv is a User Datagram Protocol (UDP) broadcast message that contains the controller's MAC and IP addresses along with other transport-layer parameters, such as the TCP port number of the OF session to be created. It not only allows new switches to acquire essential information to reach the controller, but also helps switches identify their root ports.

When an IBAB client receives an IBAB Adv message for the first time, it takes the incoming port (i.e., *in_port*) as the root port of the switch and forwards the message to every other port. It also drops any future IBAB Adv messages coming from ports other than this root port. This rule cuts off redundant flooding of IBAB Adv messages, thus alleviating the potential for a broadcast storm problem. To periodically disseminate IBAB Adv, however, the switch still forwards subsequent IBAB Adv messages coming from the root port to all other ports. Tables 1 and 2 exhibit the rules that a new switch uses to handle incoming frames/messages before and after setting up its root port, respectively.

When a switch detects a link-down event in the root port, the switch should reset all settings and restart the bootstrapping process.

C. PHASE 1B: CREATING UPLINK PATH

The *uplink path* of a switch s_i, denoted by $up(s_i)$, is the control path from s_i to the controller. In IBAB, $up(s_i)$ becomes available when s_i first receives IBAB Adv message from a upstream switch s_j. The only thing s_i needs to do is to take its root port as the portal to the controller. This is done by:

- Adding an entry to the switching table that binds the controller's MAC address with the root port. This is for the delivery of subsequent control messages during bootstrapping.
- Adding a new rule in the flow table that directs all frames with a destination MAC address matching the controller's to the root port. This will be used after the switch completes its bootstrapping process (while some other switches may not).

Afterwards, when an agent (e.g., IBAB client or DHCP client) running on switch s_i wants to send a control message to the controller, it does so through s_i's root port to s_j. Upon receiving the message, s_j then forwards it to s_j's root port. In this way, the message is delivered all the way to the controller.

Accordingly, a switch s_i can proceed to subsequent phases as soon as it has set up its root port. This is true even if some or all switches along $up(s_i)$ have not yet configured their L3/L4 parameters or established OF sessions with the controller. Refer to Fig. 2 for an illustration. This is the main difference between the proposed approach and other level-by-level methods.

TABLE 1. Before setting up the Root Port (RP).

(a) Behavior of OF Switches

Frame (Message) Type	Processing	Corresponding Flow Rule
Unicast	Dropped	None
Broadcast (IBAB Adv)	Forwarded to all other ports	No.1
Broadcast (others)	Dropped	None

(b) Corresponding Flow Rule

No.	Condition	Direction		Flow Rule		MAC Learning
		From	To	Match Fields	Action	
1	Receiving IBAB Adv	C	S	*priority*=19000, *dl_dst*=ff:ff:ff:ff:ff:ff, *ip*, *udp*, *tp_dst*=12345, *nw_dst*=255.255.255.255	flood	Yes (added to No.8)

TABLE 2. After setting up the Root Port (RP).

(a) Behavior of OF Switches

Frame (Message) Type	Processing	Corresponding Flow Rules
Unicast (with a known dest. MAC addr.)	Forwarded to the dest. port	No.1, No.3, No.8, and No.9
Unicast (with an unseen dest. MAC addr.)	Dropped	Default drop
Controller's broadcast (e.g., IBAB Adv.) from the root port	Forwarded to all other ports	No.2, No.4, and No.5
Controller's broadcast (e.g., IBAB Adv.) from any other port	Dropped	Default drop
Broadcast by any other switch	Forwarded to the root port	No.6 and No.7

(b) Corresponding Flow Rules

No.	Condition	Direction		Flow Rule		MAC Learning
		From	To	Match Fields	Action	
1	Receiving frames to the switch	C	S	*priority*=15000, *vid*="*in_band_vid*", *dl_src*="*Controller Mac*", *dl_dst*="*Switch Mac*"	action=local	Yes (update No.9)
2	Receiving IBAB Adv	C	S	*priority*=20000, *in_port*="*Root Port*", *vid*="*in_band_vid*", *dl_src*="*Controller MAC*", *dl_dst*=ff:ff:ff:ff:ff:ff, *ip*, *nw_src*="*Controller IP*", *nw_dst*=255.255.255.255, *udp*, *udp_dst*=12345	flood	No
3	Sending IBAB Reply	S	C	*priority*=20000, *vid*="*in_band_vid*", *dl_dst*="*Controller MAC*", *ip*, *udp*, *tp_src*=12345, *nw_dst*="*Controller IP*"	output="*RP*"	Yes (added to No.9)
4	ARP requests from the local port's MAC address.	S	C	*priority*=20000, *in_port*= *local*, *vid*="*in_band_vid*", *dl_dst*=ff:ff:ff:ff:ff:ff, *arp*	flood	No
5	ARP requests (broadcast)	C	S	*priority*=20000, *in_port*="*Root Port*", *vid*="*in_band_vid*", *dl_dst*=ff:ff:ff:ff:ff:ff, *arp*, *arp_spa*="*Controller IP*", *arp_op*=1	flood	No
6	ARP requests (broadcast)	S	C	*priority*=20000, *vid*="*in_band_vid*", *arp*, *arp_tpa*="*Controller IP*", *arp_op*=1	output="*RP*"	No
7	DHCP Discovery/Request (broadcast)	S	C	*priority*=20000, *vid*="*in_band_vid*", *ip*, *udp*, *dl_dst*=ff:ff:ff:ff:ff:ff, *udp_dst*=67, *nw_dst*=255.255.255.255,	output="*RP*"	No
8	Other unicast packets (ARP Reply and TCP Traffic)	S	C	*priority*=20000, *vid*="*in_band_vid*", *dl_dst*="*Controller MAC*"	output="*RP*"	Yes (update No.9)
9	Other unicast packets (ARP Reply, DHCP Offer/Ack and TCP Traffic)	C	S1	*priority*=18000, *vid*="*in_band_vid*", *dl_dst*="*Swtich S1 MAC*"	output="Port to S1"	Yes (updated)
	Other unicast packets (ARP Reply, DHCP Offer/Ack and TCP Traffic)	C	S2	*priority*=18000, *vid*="*in_band_vid*", *dl_dst*="*Swtich S2 MAC*"	output="Port to S2"	Yes (updated)

D. PHASE 2: CREATING DOWNLINK PATH

The *downlink path* of a switch s_i, denoted by $down(s_i)$, is the control path from the controller to s_i. In IBAB, $down(s_i)$ is the reverse of $up(s_i)$ because the setup of $down(s_i)$ is based on $up(s_i)$. More explicitly, the setup of $down(s_i)$ demands the delivery of an IBAB Reply message from s_i back to the controller.

After setting up the root port, the IBAB client in s_i sends back an IBAB Reply message to the IBAB server. IBAB Reply is a unicast UDP message that includes s_i's MAC address. Since the switch is not yet fully-configured, it needs to configure a temporary IP address — which can be a fake or default IP address — for itself as the source IP address. For the rest of the reply, the IBAB client has the following two implementation options. Their performances will be empirically analyzed in the next section.

- Since the client already has all the information needed to encapsulate the message in a unicast frame, it can use a raw socket (i.e., bypassing the protocol stack in the kernel) to send the frame.

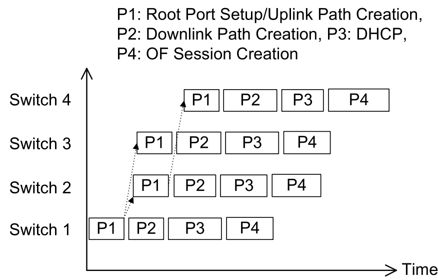


FIGURE 2. **Overlapping bootstrapping in One-Pass IBAB for the four-switch topology shown in Figure 1.**

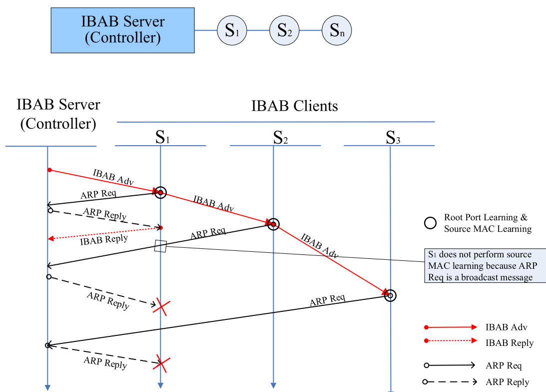


FIGURE 3. **A scenario where dynamic ARP does not work in Phase 2.**

- Alternatively, the IBAB client can make a system call to send out an IBAB Reply as a UDP datagram, i.e., by requesting a UDP socket from the kernel. The IBAB client should specify the IP address and transport-layer parameters (e.g., port number) of the controller. To prevent the kernel from issuing an ARP request, the IBAB client also needs to add a static entry to the switch's ARP table beforehand that maps the controller's IP address to the controller's MAC address. This is possible since the IBAB client has already acquired these two parameters from the IBAB Adv.
  We use static ARP instead of dynamic ARP request/reply, not because the latter is more time-consuming, but because dynamic ARP does not work at this stage. Indeed, an ARP request message sent by a switch can be delivered all the way to the controller without difficulty, given that the uplink path has already been created. However, because this request is a broadcast message, its delivery will not trigger source MAC learning in the switches along the uplink path. Consequently, the corresponding ARP reply message cannot be delivered back to the source switch unless the switch has a direct link to the controller. Fig. 3 illustrates an example with three switches S_1, S_2, and S_3 arranged in a chain. Switch S_1 can receive the ARP Reply sent by the controller as they are directly connected. On the other hand, other switches fail to receive their replies since S_1 will drop them (refer to Table 2).

The delivery of an IBAB Reply all the way back to the controller is as follows. First, because s_i's switching table already binds s_i's root port to the controller's MAC address, the IBAB Reply is sent through s_i's root port to the next upstream switch (with respect to $down(s_i)$).

When the message reaches an upstream switch s_j, s_j binds s_i's MAC address with its *in_port* and forwards the message to s_j's root port (refer to Table 2). When the IBAB server receives an IBAB Reply from s_i, the server will recognize its presence and take the s_i's MAC address as its identifier. The controller will also bind s_i's address with the *in_port* of this message. Hence, the IBAB client in s_i and the IBAB server have established a bidirectional path based on L2 identifiers, i.e., MAC addresses. However, for an OF session using IP addresses and higher layer identifiers, the switch should further perform DHCP and other activities.

E. PHASE 3: RUNNING DHCP

A switch initiates DHCP by broadcasting a DHCP Discover message. If the switch has set up its root port, the default rules, as specified in Table 2, will send out the message through the root port. Therefore, a switch that has a direct link to the controller can deliver the DHCP Discover message to the controller as soon as it sets up its root port. Other switches require the assistance of intermediate switches. An intermediate switch that receives these messages will forward them to its root port if the root port has been set up; otherwise, it will drop these messages. Since the switch's uplink path has already been created in Phase 1, the DHCP Discover message can be delivered all the way to the controller.

After receiving the DHCP Discover message, the DHCP server should respond with a unicast DHCP Offer. This message can be delivered to a destination switch s_i because $down(s_i)$ has been created in Phase 2. The subsequent DHCP messages, i.e., DHCP Request and DHCP Ack, can be respectively delivered along $up(s_i)$ and $down(s_i)$.

F. PHASE 4: CREATING OF SESSION

In case that the OF session between a switch and the controller is built upon a TCP connection, the OF client on a switch should create a TCP connection and instantiate an OF session with the controller after the switch has been configured IP layer parameters.

The OF client in switch s_i creates a TCP connection with the controller by executing the three way handshake protocol. To send its first TCP SYN message, s_i needs the binding of the controller's IP address and MAC address. This is straightforward if the corresponding ARP entry has been statically created in Phase 2 with the implementation option of UDP socket. If not, the switch itself may autonomously broadcast an ARP request asking for the controller's MAC address (which is out of the IBAB client's control). In this case, s_i's ARP request will be delivered along $up(s_i)$, which is identical to the way its DHCP Discover message is delivered.

Note that switch s_i sends back IBAB Reply messages to the controller before acquiring s_i's IP address; the source IP address in the IBAB Reply is a fake or default IP address. While the IBAB server already has switch s_i's MAC address, it may acquire s_i's IP address after receiving s_i's ARP request. Alternatively, the controller may broadcast an ARP request to reply to s_i's TCP connection request. In this case, the delivery of the ARP request from the controller to s_i is identical to the delivery of an IBAB Adv message.

After the TCP connection is established, an OF session can then be launched using the TCP connection. The bootstrapping is completed when every switch in the network has established an OF session with the controller. The controller then proceeds to topology discovery and other routine management activities, which falls under the scope of OpenFlow.

G. DISCUSSION

We are now in a position to address several aspects that merit closer investigation. First, one concern may arise when some of data-plane flow rules overlap or conflict with ours for in-band control. If this is the case, there remains no need to introduce any conflict resolution scheme such as in [16] for the following reasons.

1) Flow rules of the in-band control plane (flow rules we need) have a higher priority than those in the data plane. In other words, the former will override the latter if there is any overlapping or conflict.
2) In practice we can designate a VLAN for in-band control as a way to isolate in-band control-plane traffic from data-plane traffic. Since the controller is fully aware of the existence of this VLAN, the controller is enabled to identify any flow rule for the data plane in conflict with those for the in-band control plane (i.e., any data-plane flow rule that is conditioned on the designated VLAN). The controller can thus simply rule out such flow rules or ignore them in that the flow rules of the in-band control plane will override these conflict rules anyway.
3) Flow rules of the control-plane are not visible nor accessible to the controller. As they are de facto hidden to the controller, they are unlikely to be analyzed or modified for conflict resolution purposes.

One-Pass IBAB relies on ARP protocol. As far as ARP vulnerabilities are concerned, ARP spoofing, ARP cache poisoning, and ARP flooding are important types of cyber-attacks linked to other threats. Such vulnerabilities can be redressed with techniques presented in [17], [18] which are of avail to strengthen our architecture. Among others, Sun *et al.* devised a practical scheme [18] that utilizes a cluster of controllers, with reference to a database, to detect forged ARP packets and monitor statistical characteristics of traffic on each port of edge switches. The scheme was shown to efficiently withstand attacks against ARP in an SDN-based cloud computing environment.

On the other hand, we argue that our design of One-Pass IBAB is not weakened unduly due to typical ARP vulnerabilities. Observe that ARP is used in our bootstrapping Phase 2 and Phase 4. Sec. III-D describes the rationale of our development adopting static ARP instead of dynamic ARP request/reply during Phase 2, which rules out the possibility that a malicious node masquerades as a host intercepting traffic on the network. Thanks to static ARP, switches in our paradigm do not involve ARP message exchanges as cache entries exist.

During Phase 4, a new switch s_i in our architecture might opt to broadcast an ARP request for resolving the controller's MAC address (Sec. III-F) Another likely scenario takes place when the controller generates an ARP request in response to s_i's TCP connection request. In either case, the controller has learned of s_i's MAC address upon receipt of the IBAB Reply message earlier during Phase 2 and, thus, is able to validate the new switch. Accordingly, the controller is enabled to prevent any non-s_i hosts from compromising the network. Additionally, in view that an ARP attacker must have direct access to the local network, we may stipulate that immediate neighbor switches whitelist the bootstrapping s_i which has undergone prior phases, so neighbor switches allow ARP packet forwarding for s_i via the root port but drop ARP packets coming from other uncertified local nodes. This delimits scope for ARP vulnerabilities. We mention in passing that nowadays some operating systems like OpenBSD or software tools are available to deal with ARP spoofing.

We remark that our bootstrapping Phases 1 and 2 involve IBAB Adv/Reply in a flavor nearly identical to the All-Path protocol [19] in sense of exploring the network as well as finding the fastest path between two nodes. However, aforementioned potential ARP vulnerabilities are worth noting and warrant treatment if ARP is employed to implement the path-discovery process of the protocol. In comparison, IBAB Adv and Reply messages are UDP-based which can be digitally signed for authenticity protection as well. Further, our design differs from All-Path in that, more than finding a forwarding path as an essential part, our scheme focuses on speeding up the entire bootstrapping processes on a plurality of switches in a pipelined fashion. We contrive means out of several feasible measures to interrelate switches by overlapping network operations of layers 2 up to 4 to the greatest extent possible, resulting in significant performance improvement as shall be corroborated in the next section.

When multiple controllers coexist in the same network, the proposed approach still works if only one controller is active while all others are standby. If two or more controllers are active at the same time, there will be multiple controllers broadcasting IBAB Adv messages. In that case, multiple spanning trees could be formed provided that each switch identifies the controller MAC address when processing relevant IBAB Adv/Reply messages. More explicitly, after a switch receives the first IBAB Adv message and thus

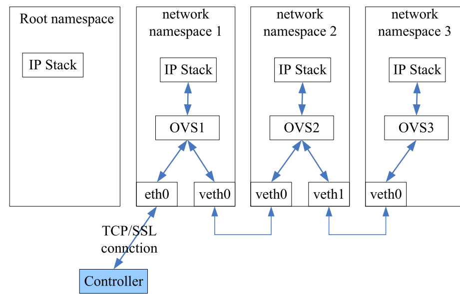


FIGURE 4. Emulation environment (for a three-switch chain).

recognizes the controller MAC address, IBAB Adv from any other controller should be discarded silently.

IV. NUMERICAL RESULTS

We used OVS to emulate SDN switches and measured the bootstrapping time of the proposed approach. We replaced all hidden rules in the OVS kernel with a default rule, which stipulates all incoming IBAB messages to be forwarded to the local port and others to be dropped. Two virtual machines (VMs) were used for our emulations: one equipped with 8 CPUs and 100 GB memory, the other equipped with 24 CPUs and 20 GB memory. The former was used to emulate the controller; the latter was used to emulate the network environment, within which various topologies were created using linux shell scripts. Each OVS had its own network namespace to run its own protocol stack (Fig. 4). Mininet could also be used provided that each OVS has its own network space.

A. TESTING IMPLEMENTATION OPTIONS: RAW VS. UDP

Recall that an IBAB client may use either a raw socket or a UDP socket to send IBAB Reply messages back to the controller. We refer to these two options as Raw and UDP, respectively, and compared their performances. To simplify the effects of network topology, we arranged all switches in a chain topology and varied the number of switches (i.e., the length of the chain) from 10 to 70.

We first measured the control path creation time (Phases 1 and 2) of the whole network. This process begins with the broadcast of the first IBAB Adv message, and ends when the controller receives the last IBAB Reply message. Fig. 5 details the measured results. The control path creation time with Raw is significantly lower than with UDP. This is due to the extra time needed to add a static ARP entry when using UDP socket.

We next measured Phase 3, IP configuration time, and Phase 4, OF session creation time, for each switch in the chain respectively. The IP configuration time of a switch begins when the switch first broadcasts a DHCP Discover message, and ends when the switch configures its IP address. The OF session creation time is the time between the switch sending out its first TCP SYN and receiving the OF feature reply message, the latter of which indicates the creation of

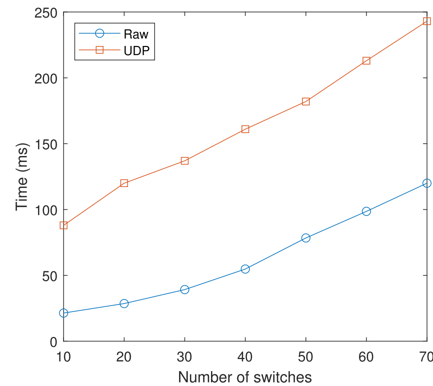


FIGURE 5. The control path creation time of the whole chain.

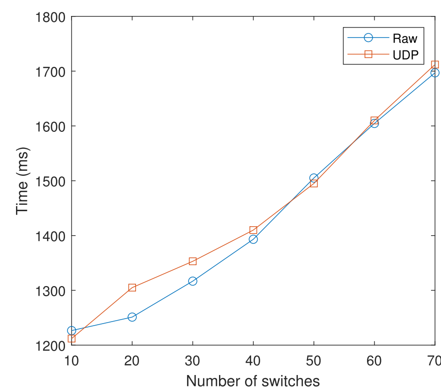


FIGURE 6. The averaged sum of the IP configuration time and the OF session creation time per switch.

its OF session. Let Y_i be the sum of these two components in switch i. Fig. 6 displays the averaged value of Y_i (i.e., $\bar{Y} = \sum_{i=1}^{n} Y_i/n$). The result shows that $\bar{Y}$ increased with the length of the chain. Raw and UDP performed similarly concerning these two time components.

Finally, we measured the time it takes to bootstrap all switches. This begins with the broadcast of the first IBAB Adv and ends at the moment in which the last switch creates its OF session. Fig. 7 displays the results. The bootstrapping time was around 2.0 seconds for a 50-switch chain. For comparison, Sharma *et al.* [2] reported a bootstrapping time of around 100 seconds for a 50-switch chain. Our approach thus yielded a 98% time reduction.

Although Raw had relatively low control path creation time compared with UDP, the resulting bootstrapping time difference between these two is inconsequential. The reason is that control path creation time is relatively short compared with the rest of the bootstrapping process. Once the control path of the whole network is created, the rest of the procedure for all switches can overlap. Let $\hat{Y} = \max_i\{Y_i\}$ and X be the total span of control path creation time. Taking into account the overlapping, the overall bootstrapping time can be roughly estimated as $X + \hat{Y}$ (refer to Fig. 8). Accordingly, $X \ll \hat{Y}$ is the factor that dominates overall bootstrapping time in our approach.

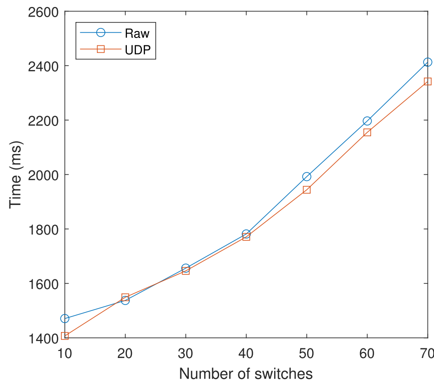


FIGURE 7. The bootstrapping time of a chain.

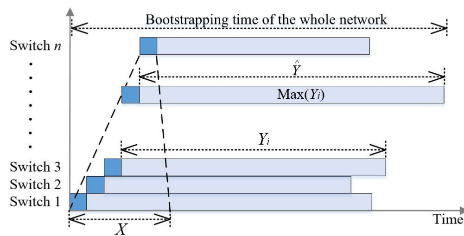


FIGURE 8. Estimating the bootstrapping time of the whole network.

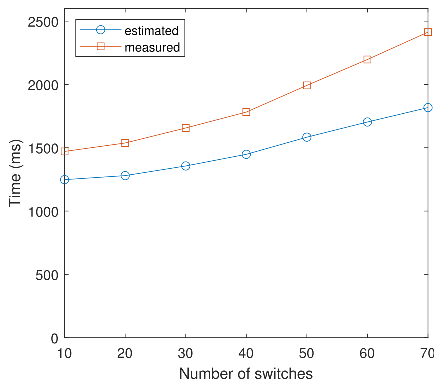


FIGURE 9. The estimated and measured bootstrapping time in Raw.

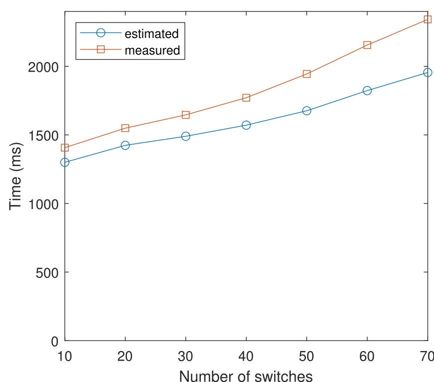


FIGURE 10. The estimated and measured bootstrapping time in UDP.

We used $\bar{Y}$ (instead of $\hat{Y}$) to estimate bootstrapping time. Figs. 9 and 10 respectively demonstrate the estimated

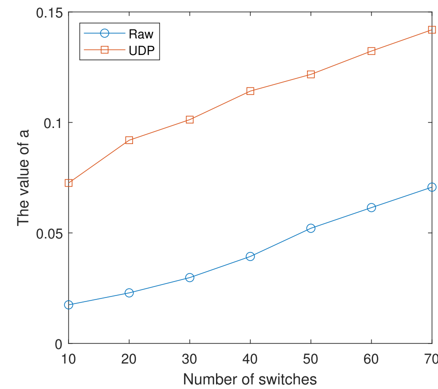


FIGURE 11. The value of *a* in a chain topology.

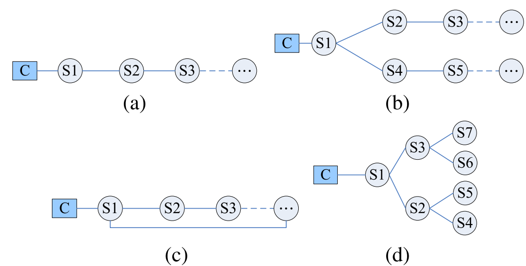


FIGURE 12. Topologies tested (a) chain (b) double-chain (c) ring (d) full binary tree.

and measured bootstrapping time in Raw and UDP. Since $\bar{Y} \leq \hat{Y}$, the estimated time was lower than the measured time as expected.

Although X constitutes only a small portion of the overall network bootstrapping time, it directly determines the time from which all switches start IP configuration and OF session creation with the controller. Let $a = X/\bar{Y}$. We have $a > 1$ for any level-by-level bootstrapping. The overlap between IP configuration and OF session creation among switches is only possible when $a < 1$. In this case, the degree of overlapping is inversely proportional to the value of a. Fig. 11 displays how the value of a changes with the number of switches in a chain.

B. CONTROL PATH CREATION TIME IN VARIOUS TOPOLOGIES

Since control path creation time of the whole network is critical to overall performance, we studied it in four different types of physical topologies: chain, double-chain, ring, and full binary tree (Fig. 12). We further increased the number of switches and measured their control path creation times. We tested the full binary tree topology with 15, 31, 63, and 127 switches. For the other topologies, we tested with 20 to 180 switches with increments of 20. Fig. 13 shows the measured results. Observe that the chain topology had the highest control path setup time. The results of the double-chain topology was nearly half of the chain topology's. This is because the maximum length of the control path in the double-chain topology is only half the length of the control

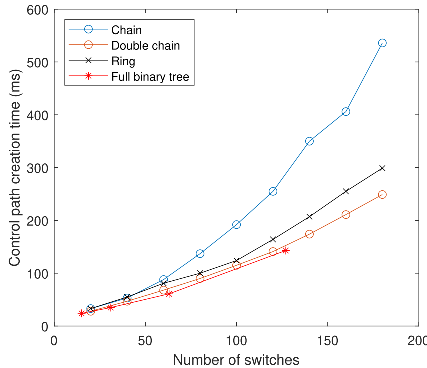


FIGURE 13. Control path creation time in various topologies.

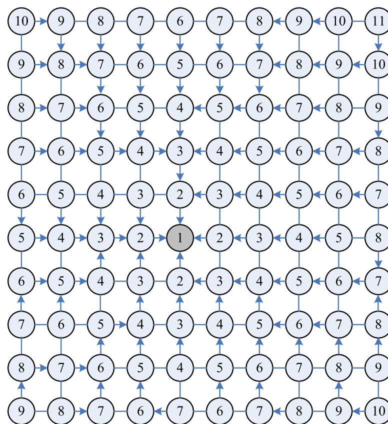


FIGURE 14. The uplink spanning tree in a 10 × 10 crossbar.

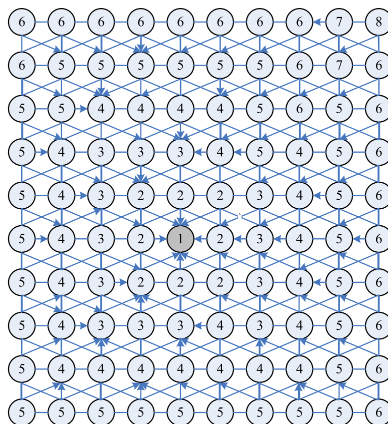


FIGURE 15. The uplink spanning tree in a 10 × 10 2-dim diagonal grid.

path in the chain topology. The ring-topology fared even worse than the double-chain topology. This is because the maximum length of the control path in the ring topology was not always the half of the number of switches due to the race condition of switches when setting up control paths. The full binary tree topology had the best performance, but its superiority is not very significant.

To study the topology of the uplink spanning tree from every switch to the controller, we also tested a 10 × 10 crossbar and a 10 × 10 2-dim diagonal grid. The numbers in Figs. 14 and 15 indicate the lengths of the control paths from each switch to the controller. In these two settings, the controller was placed in the same location as the switch with the minimal length value of 1. In the crossbar, the maximal length value of 11 belongs to the switch in the upper right corner, while the length value for the same switch in the 2-dim diagonal grid is 8. Furthermore, the results show that the control paths for some switches are not the shortest paths for them to reach the controller. This is also due to the race condition of switches when setting up control paths.

V. CONCLUSION

We have presented a faster bootstrapping scheme for OpenFlow switches in which switches set up control paths in a partially overlapping manner. In this scheme, a switch can relay control messages for its neighboring switches to the controller, even if the switch has not yet configured its L3/L4 setting. We have detailed how to create control paths, configure IP addresses, and create TCP connections and OF sessions. Emulations demonstrated a significant reduction in bootstrapping time with this approach.

Future works include an extension to controller-clustering environment, faster failover design for link or path failure [11], control path monitor [14] and autonomic control plane [20].

REFERENCES

[1] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "Automatic bootstrapping of OpenFlow networks," in *Proc. 19th IEEE Workshop Local Metrop. Area Netw. (LANMAN)*, Brussels, Belgium, Apr. 2013, pp. 1–6.

[2] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "A demonstration of automatic bootstrapping of resilient OpenFlow networks," in *Proc. IFIP/IEEE Int. Symp. Integr. Netw. Manage.*, Ghent, Belgium, May 2013, pp. 1066–1067.

[3] L. Schiff, S. Schmid, and M. Canini, "Medieval: Towards a self-stabilizing, plug & play, in-band SDN control network," in *Proc. ACM Sigcomm Symp. SDN Res.*, 2015. [Online]. Available: https://www.semanticscholar.org/paper/Medieval%3A-Towards-A-Self-Stabilizing%2C-Plug-%26-Play%2C-Schiff/05be6379ca710385f83d09ddf31f3d42154f2f8a

[4] L. Schiff, S. Schmid, and M. Canini, "Ground control to major faults: Towards a fault tolerant and adaptive SDN control network," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. Workshop (DSN-W)*, Jun. 2016, pp. 90–96.

[5] M. Canini, I. Salem, L. Schiff, E. M. Schiller, and S. Schmid, "A self-organizing distributed and in-band SDN control plane," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 2656–2657.

[6] O. I. Bentstuen and J. Flathagen, "On bootstrapping in-band control channels in software defined networks," in *Proc. ICC Workshops*, Kansas City, MO, USA, May 2018, pp. 1–6.

[7] R. Katiyar, P. Pawar, A. Gupta, and K. Kataoka, "Auto-configuration of SDN switches in SDN/Non-SDN hybrid network," in *Proc. Asian Internet Eng. Conf.*, Bangkok, Thailand, Nov. 2015, pp. 48–53.

[8] P. Heise, F. Geyer, and R. Obermaisser, "Self-configuring deterministic network with in-band configuration channel," in *Proc. 4th Int. Conf. Softw. Defined Syst. (SDS)*, May 2017, pp. 162–167.

[9] D. Lopez-Pajares, J. Alvarez-Horcajo, E. Rojas, A. S. M. Asadujjaman, and I. Martinez-Yelmo, "Amaru: Plug&Play resilient in-band control for SDN," *IEEE Access*, vol. 7, pp. 123202–123218, 2019.

[10] *IEEE Standard for Local and Metropolitan Area Networks—Bridges and Bridged Networks*, Standard 802.1Q-2018, IEEE, Jul. 2018.

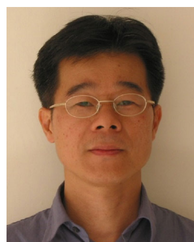
[11] A. S. M. Asadujjaman, E. Rojas, M. S. Alam, and S. Majumdar, "Fast control channel recovery for resilient in-band OpenFlow networks," in *Proc. 4th IEEE Conf. Netw. Softwarization Workshops (NetSoft)*, Jun. 2018, pp. 19–27.

[12] L. Ochoa-Aday, C. Cervelló-Pastor, and A. Fernández-Fernández, "ETDP: Enhanced topology discovery protocol for software-defined networks," *IEEE Access*, vol. 7, pp. 23471–23487, 2019.
[13] F. Wu and A. Tian, "RXstp: A topology discovery mechanism based on rapid spanning tree for SDN in-band control," in *Proc. Int. Conf. Commun., Inf. Syst. Comput. Eng. (CISCE)*, May 2021, pp. 703–706.
[14] I. I. Awan, N. Shah, M. Imran, M. Shoaib, and N. Saeed, "An improved mechanism for flow rule installation in in-band SDN," *J. Syst. Archit.*, vol. 96, pp. 32–51, Jun. 2019.
[15] P. Goltsmann, M. Zitterbart, A. C. Hecker, and R. Bless, "Towards a resilient in-band SDN control channel," Universität Tübingen, Tübingen, Germany, Tech. Rep., 2017.
[16] A. B. Asif, M. Imran, N. Shah, M. Afzal, and H. Khurshid, "ROCA: Auto-resolving overlapping and conflicts in access control list policies for software defined networking," *Int. J. Commun. Syst.*, vol. 34, no. 9, p. e4815, Jun. 2021.
[17] S. Hijazi and M. S. Obaidat, "Address resolution protocol spoofing attacks and security approaches: A survey," *Secur. Privacy*, vol. 2, no. 1, pp. 703–706, 2019.
[18] S. X. Sun, X. Fu, B. Luo, and X. J. Du, "Detecting and mitigating ARP attacks in SDN-based cloud environment," in *Proc. IEEE INFOCOM*, Jul. 2020, pp. 659–664.
[19] E. Rojas, G. Ibañez, J. M. Gimenez-Guzman, J. A. Carral, A. Garcia-Martinez, I. Martinez-Yelmo, and J. M. Arco, "All-path bridging: Path exploration protocols for data center and campus networks," *Comput. Netw.*, vol. 79, pp. 120–132, Mar. 2015.
[20] T. Eckert, M. H. Behringer, and S. Bjarnason, *An Autonomic Control Plane (ACP)*, document RFC 8994, May 2021. [Online]. Available: https://www.rfc-editor.org/rfc/rfc8994.txt

CHIEN-YUNG LI received the M.S. degree in information management from the National Pingtung University, Pingtung, Taiwan, in 2005. He is currently pursuing the Ph.D. degree in computer science with the National Yang Ming Chiao Tung University, Hsinchu, Taiwan. He was a Senior Software Engineer with Alpha Networks Inc., Hsinchu, from 2005 to 2010, and the Cloud Computing of Industrial Technology Research Institute, Hsinchu, from 2010 to 2014. He was a Project Manager with D-Link Corporation, Taipai, Taiwan, from 2014 to 2021. He is currently a Techical Manager with Wistron Corporation, which he joined in June 2021, Hsinchu. His research interests include cloud computing, traditional switch, software-defined networks, and cloud storage.

LI-HSING YEN (Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from the National Chiao Tung University, Hsinchu, Taiwan, in 1989, 1991, and 1997, respectively. Before he joined the Department of Computer Science, National Chiao Tung University, he was with Chung Hua University, from 1998 to 2006, and the National University of Kaohsiung, Taiwan, from 2006 to 2016. His current research interests include distributed computing, wireless networking, and game theory. He has won best paper awards of IEEE WCNC 2013 and ISPA 2015. He has served on the editor boards of *Wireless Networks* (Springer).

KUANG-HUI CHI (Senior Member, IEEE) received the Ph.D. degree in computer science and information engineering from the National Chiao Tung University, in 2001. He is currently a Professor at the Department of Electrical Engineering, National Yunlin University of Science and Technology, Taiwan. Prior to the current position, he was with the Industrial Technology Research Institute, Taiwan. His research interest includes wireless internet.



CHIEN-CHAO TSENG (Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the Southern Methodist University, Dallas, TX, USA, in 1986 and 1989, respectively. He joined the Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan, in 1989, where he is currently a Professor. He is the P.I. of the SDNFV Orchestration for 5G project at the Center for Open Intelligent Connectivity sponsored by the Higher Education Sprout Project by the Ministry of Education (MOE), Taiwan. His research interests include software defined networks, network function virtualization, wireless internet, and heterogeneous networks.