ConForm: In-band Control Plane Formation Protocol to SDN-Based Networks

Marcelo Silva Freitas, Romerson Oliveira, Diego Molinos,
Juliano Melo, Pedro Frosi Rosa, Flavio de Oliveira Silva
Faculty of Computing
Federal University of Uberlandia
Uberlandia, Brazil
{msfreitas, romerson, diego.molinos, julianoco, pfrosi, flavio}@ufu.br

***Abstract*—Although OpenFlow-based SDN networks make it easier to design and test new protocols, when you think of clean slate architectures, their use is quite limited because the parameterization of its flows resides primarily in TCP/IP protocols. Besides, despite the many benefits that SDN offers, some aspects have not yet been adequately addressed, such as management plane activities, network startup, and options for connecting the data plane to the control plane. Based on these issues and limitations, this work presents a bootstrap protocol for SDN-based networks, which allows, beyond the network topology discovery, automatic configuration of an in-band control plane. The protocol is designed to act only on layer two, in an autonomous, distributed and deterministic way, with low overhead and has the intent to be the basement for the implementation of other management plane related activities. A formal specification of the protocol is provided. In addition, an analytical model was created to preview the number of required messages to establish the control plane. According to this model, the proposed protocol presents less overhead than similar *de-facto* protocols used to topology discovery in SDN networks.**

***Index Terms*—Bootstrap, In-Band Control Plane, Clean Slate Architectures, Protocol Design, Self-establishment**

I. Introduction

In Software Defined Networking (SDN), the control and data planes are decoupled. Network intelligence and state management are logically centralized, and the underlying network infrastructure is transparent to applications [1]. This decoupling is implemented, for example, by OpenFlow technology, which allows the creation of distinct flows based on OpenFlow protocol parameters.

The SDN philosophy breaks the old vertical integration of the network, in which the control and data functions coexist in a single plane. In spite of this, the discussion about deployment of the control plane, out-of-band or in-band, initially untreated, has recently gained more prominence among research in the area.

In out-of-band mode, controllers are connected to the switches through dedicated links used exclusively for traffic of control messages. But for many use cases, this is not economically feasible. In in-band mode, control and data packets share the same infrastructure, making it necessary to initialize the network, that is, establish a controller communication for the controlled switch via a shared infrastructure [2], [3].

As the network paradigm is changing, all aspects related to it, including network management aspects such as bootstrapping, resilience, security, monitoring, among others, must also change to allow its evolution [4]. In this sense, since startup activity is related to the management plane, it can be stated that a bootstrap protocol is one that operates in the management plane to configure the control plane, acting during a pre-operational stage of the network.

In addition, although OpenFlow-based SDN networks facilitate the design and testing of new protocols, when you think of clean slate architectures, their use is quite limited. This is due to the fact that the programming capability allowed by the OpenFlow technology is oriented to the implementation of flows in the network elements based only on the protocol headers of the TCP/IP stack. Besides that, several disadvantages of the OpenFlow switch are highlighted in [5] and limitations of OFDP (OpenFlow Discovery Protocol) is discussed in [6]. Finally, there is no flexibility for link layer programming, i.e., no programmability is applicable to the Medium Access Control (MAC).

Motivated by the issues and limitations described above, this article presents **ConForm** (**Con**trol plane and network **Form**ation), a network formation protocol for SDN-based network architectures, which allows, beyond topology discovery, automatic bootstrapping of an in-band control plane, establishing control paths, like a spanning tree, from the controller to every network element. With the purpose of be SDN-generic and independent of TCP/IP protocols, the proposed protocol was implemented over layer two.

The remainder of the document is structured as follows: section II presents the state-of-the-art related to SDN bootstrapping methods. Section III presents the formal specification of **ConForm**. Section IV develops an analytical model to the protocol and finally, section V offers a discussion about the protocol, some concluding remarks and makes suggestions for future work.

II. Related Work

This section presents an overview of the literature regarding topology discovery and control plane configuration on SDN-based networks. The problem of automatic initialization and

configuration of SDN-based networks has been addressed in several ways.

The automatic initialization procedure proposed by [2] is a pioneering work on this theme, which basically follows three steps: (i) assignment of switch-controller connection identifiers, such as switch IP and controller IP; (ii) instantiating an OpenFlow session with the controller, establishing a route from the switch; and (iii) network topology discovery. The solution uses Dynamic Host Configuration Protocol (DHCP), Address Resolution Protocol (ARP), OpenFlow Protocol, and Link Layer Discovery Protocol (LLDP). Therefore, it is about a method and not a novel protocol specific to bootstrapping. Furthermore, resilience aspects as presented in **ConForm** are not considered.

The work [7] presents a model for distributed control plane project that stabilizes itself from any initial setup. Its main contribution is the design of an In-Band control system that coordinates distributed controllers in a self-organizing way, to perform the bootstrapping of connectivity between controllers and switches. The method is based only on OpenFlow. The controller installs rules on neighboring switches, which can be used to install rules on two-hop switches, successively expanding the controller domain to manage remote switches. Virtual LANs (VLANs) are used to implement configurations on the intermediate switches. In the control plane, two distinct spanning trees are created: a spanning tree bidirectional by region, which spreads across the area managed by a controller; and another spanning tree that enable each controller to reach any other controller. Every switch initially broadcasts its connection attempts addressed to the controller anycast address. This address needs to be previously configured in each switch.

Current SDN controllers use the OpenFlow Discovery Protocol (OFDP). It is the *de-facto* protocol for discovering the underlying network topology. After each switch has been manually configured with the controller address and port, an initial handshake establishes the connection between the SDN controller and the switches, and OFDP is started. The SDN controller begins by sending LLDP frames encapsulated in Packet-Out messages to each active port on each switch in the network. By default, after receiving an LLDP packet from ports other than the controller port, each active switch must send a Packet-In message that contains the received LLDP to the controller. Send out a packet for each port of the network make OFDP inefficient. Thus, several works like [8] and [9], have analyzed this protocol and proposed variations to solve its flaws and limitations with respect to efficiency, security and functionality.

As can be seen, the solutions above depends on OpenFlow, directly or indirectly, and other traditional protocols. So it does not fit with clean slate architectures projects. **ConForm** proposes a new and independent protocol, acting in layer two, decentralized and efficient in compose a control tree only with authenticated switches.

III. CONFORM PROTOCOL SPECIFICATION

A protocol specification must consider five essential elements to reach a consistent and well formed protocol. In this sense, the five **ConForm** elements are presented in this section: assumptions about environment, services, vocabulary, formatting and procedure rules [10].

A. Assumptions about Environment

ConForm is designed to setup the in-band control channels from the controller to every network element. It is assumed that there is a logically centralized controller and that control and data planes are decoupled. In addition, it assumes no standard interfaces between the planes. Therefore, the protocol is designed for a generic SDN-based network.

In this sense, the network is composed by SDN switches interconnected with each other by full-duplex links. Furthermore, a controller must be initially running, ready to process requests, and connected to one of the switches. More specifically, the protocol proposed in the current version is capable of configuring control channels for a SDN local network, i.e., the region comprised of switches under the domain of one Controller. It is understood that interconnection of Controllers may involve political aspects beyond physical connectivity.

Additionally, since the proposed protocol is a clean slate one, and acts in layer two, its implementation would be facilitated if the network element could be programmable at data plane level [11].

B. Protocol Services and Vocabulary

The **ConForm** protocol design was driven by efficiency and effectiveness requirements with a minimum number of messages to minimize the impact on switch design and Controller overhead. Table I shows the relation between the protocol services and their respective messages. The messages follow the taxonomy of confirmed services, *Request/Indication* and *Response/Confirmation*, i.e., message *MSG_request*, for example, has the same content as message *MSG_indication*, differing semantically from the moment it occurs.

C. Message Formatting

The message format is shown in Fig. 1. The meaning of fields and PDUs will be further gradually explained in the next section. **ConForm** messages were designed to be encapsulated in Ethernet frames. However, frame address fields carry the identifiers of the source and destination entities instead of the Ethernet MAC addresses. This decision keeps compatibility with current network cards, allowing the reuse of available technology and preserving investment.

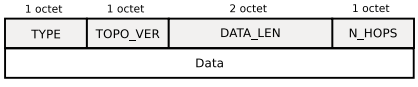

Figure 1. Message format.

Table I
PROTOCOL SERVICES AND VOCABULARY

Service	Messages	Functional Description
Switch Registration	reg_req reg_resp+ reg_resp-	Confirmed service used by switches to request registering and authentication to the Controller. Through this service, each switch learns the Controller ID and the port that leads to it.
Topology Update	upd_req upd_resp	Confirmed or Unconfirmed service used by switches to send information about their neighbors to the Controller.
Switch Advertisement	adv	Unconfirmed service used by switches to gather current state information about neighbors.
Network Reboot	reboot	Unconfirmed service used by Controllers to restart bootstrapping of switches.

D. Procedure Rules

In this section, the services from Table I are better explained.

It is assumed that the switch is manually configured by engineers involved in network management in two moments. First, when it comes to making the physical connectivity of the equipment, at which point the equipment is powered up and turned on. Second, when the analyst configures the device's ID (unique) and Cryptographic Key, that only the Controller can validate. These are the only procedures at which the equipment is manipulated. From that point on, there is no longer any need for manual setup. The next version of the protocol, is intended to eliminate any manual configuration.

The protocol rules are specified through a Finite State Machine (FSM). The diagram in Fig. 2 formally describe the behavior of the protocol services. Each transition is represented by the combination of input and output in the format $\frac{input}{output}$. The symbols ? and ! represent, respectively, the actions of *Receiving* and *Sending* messages.

1) Registration and Authentication: When a switch is powered on, it sends the message **reg_req** through all its connected ports, that is, those ports that have a wired device attached. The message **reg_req** contains, in its payload, the switch key for authentication.

An `UNREGISTERED(S0)` switch remains in semi-operational mode and can only send the message **reg_req** and should discard any other messages.

As Fig. 2 shows, when a switch sends a registration request (**!reg_req**), its state changes to `WF_REG_CONF(S1)` state. This is the state at which the switch is waiting for (WF) confirmation to its registration request. If there is a `timeout` in state `S1` or a corrupted message (`error`) is received, the switch returns to the state `S0` and another request will be sent.

The periodic sending of **reg_req** messages is interrupted if a **reg_conf+** is received, when the switch goes to `REGISTER (S3)` state. Upon receiving such a confirmation, the switch stores the (`ControllerID`), i.e. the source

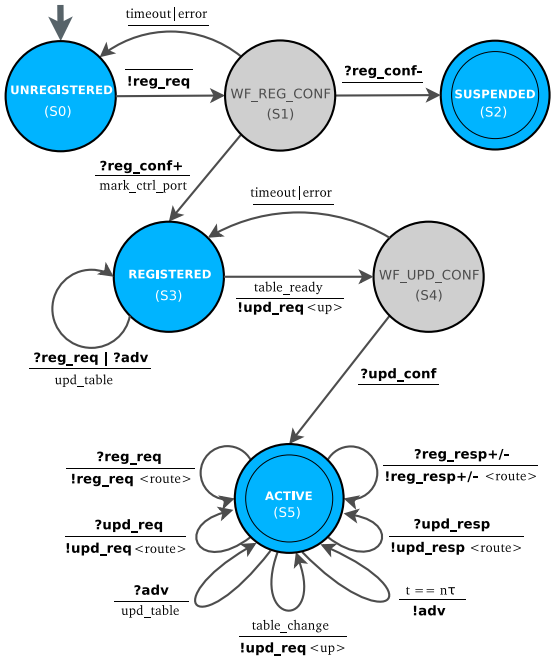

Figure 2. Switch automaton.

address of the message, and the port by which the confirmation arrived (`ControllerPort`). This will be important to routing and relay functions of active switches, as will be described in section III-D4. Otherwise, if registration fails, a **reg_conf-** message is received. For security reasons, this message hides the Controller ID, since the switch has not been authenticated and should not know about that ID. The Controller should log the event. In case of receiving this negative confirmation, the switch disables all ports and stays `SUSPENDED(S2)` until a manual intervention of the network management team takes place and turn the switch again to initial state `(S0)`. According to our experience of more than twenty years in the telecommunication industry and wide area networks, this possibility is very unlikely and the cases experienced have required intervention from the security office.

2) Topology Update: In `REGISTERED` state, the switch cannot forward messages, thus, other higher ranking switches cannot be registered yet. A higher ranking means they are farther from the controller. However, in this state the switch can receive **reg_req** or **adv** messages from its neighbors. The sender ID is extracted from this messages, and the switch neighbors table is updated (`upd_table`). Thus, when all neighboring switches are identified and inserted into the neighboring table, the switch sends the payload-encoded table into a **upd_req** message addressed to the Controller. Eventually, this table reaches the Controller where its information is processed and used to compose the logical topology representation. Next, the Controller sends a **upd_resp** to the switch. When switch receives this confirmation, its state finally moves to `ACTIVE(S5)` and then all of its relay functions are

enabled. As shown in Fig. 2, the update is confirmed service in the state `(S3)`. For this reason, the state `(S4)` represents the waiting for confirmation.

3) Advertisement: Entering the `ACTIVE` state triggers the advertisement service on each connected port, that is, the switch advertise the neighbors of its presence. Besides that, it is advertised of the presence of neighbors. This service is a *one-hop keep alive* signaling. Messages from this service have no confirmation because they are periodically transmitted from one switch to each of its neighbors through fast links, frequently fiber optical links, where error rates are very low.

The **`adv`** messages are sent after specified time intervals, $n\tau$. When a switch receives this message, it uses the information in its header and payload to update the neighbors table. Upon a change in the neighbors table is detected, the update service (**`upd_req`**) must be requested to the Controller, but in state `S5` there will be no confirmation message. Update messages are also sent to the Controller if a switch does not receive an advertisement message from a known neighbor switch after a specified time, $m\tau$. This is necessary to detect when a neighboring switch has some kind of problem and stops sending the **`adv`** message.

Still analyzing the Fig. 2, it should be noted that, active switches also must relay Topology Update service messages `<up>` to the Controller as well as `<down>` to other switches.

4) Routing: To enable switches ranked higher than zero to succeed in their attempts to reach the Controller, there must be a logical chaining of active switches to `<route>` registration requests toward the Controller as well as to `<route>` Controller responses toward higher ranked switches.

With **ConForm** protocol this goals are accomplished using marked ports and routes embedded in messages PDUs. Let's look at the example from Fig. 3 which illustrates an unregistered (U) switch S2 trying to be registered. To do this, requests from switch S2 must be forwarded by active (A) switches S1 and S0 to the Controller. To explain how requests are routed upward and responses downward, the registration processes will be reviewed from a routing point of view.

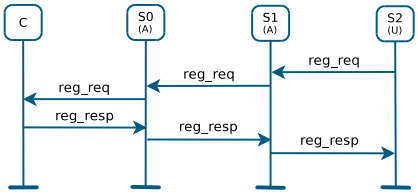

Figure 3. Registration procedure of a switch with rank=2.

The registration request (**`reg_req`**) of switch S2 will be transmitted to its neighbors. This message have within its PDU the ID of switch S2. One of the requests reaches active switch S1 which adds its own ID to the PDU message and forward it through the port `ControllerPort` to the next switch S0. Switch S0 proceeds in the same manner, by adding its ID in PDU message and dispatching it to Controller. The Controller extracts the list of IDs from the received PDU and stores this list (indeed a route) in a data base. This database will be used to build a graph representing the network topology. If there is an alternative route that connects the S2 switch to the Controller, another request from S2 containing the other route will be received by the Controller. If so, both routes will be stored, but the Controller will use the inverse of the smallest to compose the response to switch S2.

Each switch on the downward route will pull its own ID from the list in the reply message, look up the next ID in the list, use that ID to index the its neighbors table, and forward the message to the next down switch. Finally, the port by which the message was sent will be marked as a `DownstreamPort` in table.

The routing mechanism described above is required for the switch Registration service as well as Topology Update service.

5) In-band Control Flow - ICF: It can be seen that, as a consequence of the way the registration process was designed, there will be a deterministic switch activation sequence throughout the physical network topology, with the Controller acting as a pivot, as shown in Fig. 4. Switch S0, directly connected to the Controller, will always be the first to receive registration response. It becomes active and begins relaying messages from its neighboring switches. Then these neighbors will be activated and so on. This process will result in an activation wave that propagates until all switches in the network has been activated. Note that this process is driven by state machines within each switch and not by the Controller. Despite the name (Controller), in this network formation phase, it acts as an information assembler for the purpose of composing the representation of the logical topology.

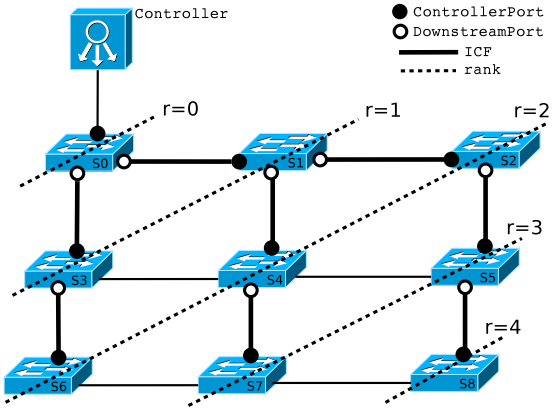

Figure 4. Activation wave and the ICF - In-band Control Flow.

When all switches in the Controller domain are in the Active state, the ICF will be fully formed. ICF consists of a logical set of port numbers formed by all `ControllerPorts` and all `DownstreamPorts`.

It is noteworthy that the ICF tree, rooted in the Controller, is an optimized spanning tree. This is true because the Controller

always responds to registration requests by choosing the shortest route down to the switch.

E. Adaptability to topology changes

By analyzing the **ConForm** specification against resiliency requirements, it can be stated that the protocol is capable of reacting appropriately to some important scenarios, such as:

1) Network rebooting: in the event of a generalized power outage, for example, on return, the switches will restart in the `UNREGISTERED` state and the entire procedure will be repeated. In this case, the `TOPO_VER` (Topology Version) field value will be incremented by the Controller.

2) Deployment of new switch: If a new unregistered switch is added to the infrastructure, connected to an active switch, requests from new switch will be routed through a chain of active switches up to the Controller. Both, the new switch and its neighbor, where it is connected, will send update messages and Controller will update the topology.

3) Switch/port fault: when a switch/port fails, the neighboring switches stop receiving **`adv`** messages. Thus, as soon as a switch periodically detects missing messages **`adv`**, it sends a message **`upd_req`** to the Controller to update the network state. If the Controller deems it necessary, a new bootstrapping could be started, through Network Reboot service, as an attempt to correct the absence of this switch/port.

Noting that for all the cases described above, **ConForm** is intended to support network resiliency in detecting and reporting failures. Procedures for fixing failures are beyond the scope of the protocol and can be considered as part of a management plane. It is the responsibility of the network (domain) Controller to reconstruct the logical representation of the network topology, including ICF recovery. In the current version, **ConForm** provides only the message **`reboot`**. However, other messages intended to repair the network in a less invasive way may be implemented in the future.

IV. Analytical Model

In this section, we model the behavior of a switch from its registration until it becomes active. An analytical model was developed to determine the time required for a switch to become active. Our goal is not to offer a highly accurate model. Therefore some simplifying assumptions were introduced. For this purpose, the probabilistic model shown in Fig. 5 was derived from the switch automaton in Fig. 2.

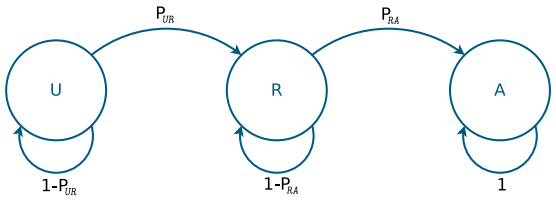

Figure 5. Markov chain representing switch behavior.

The behavior of a switch was modeled through a Discrete Time Markov Chain. Specifically, three states were considered: U, R and A, that respectively stands for `UREGISTERED`, `REGISTERED` and `ACTIVE` states. The probability matrix of the Markov chain is as follows:

$$P = \begin{bmatrix} 1 - P_{UR} & P_{UR} & 0 \\ 0 & 1 - P_{RA} & P_{RA} \\ 0 & 0 & 1 \end{bmatrix}$$

Since the matrix has an absorbing state (A), the average number of steps before entering state A can be evaluated as:

$$t = \frac{1}{P_{UR}} + \frac{1}{P_{RA}}$$

Each switch goes from state U to state R with probability P_{UR}, which is the probability that the switch will register successfully. For this to occur, after sending a registration request, the switch must receive a positive confirmation, without error or timeout:

$$P_{UR} = P_{reg\_conf}.P_{no\_error}$$

Let P_e be the probability of an error or timeout, and let P_{fa} the probability of authentication failing. The probability that the switch receives a positive registration confirmation depends on whether the authentication does not fail. Hence:

$$P_{UR} = (1 - P_{fa}).(1 - P_e)$$

For the sake of simplicity of analysis, we assume that all authentications are successful and therefore P_{fa} will be 0.

Similarly, the probability that the switch state will go from R to A, that is the probability of receiving positive update confirmation, after sending update request, is:

$$P_{RA} = P_{upd\_conf} = (1 - P_e) = P_{UR}$$

The probability P_e depends on the distance from the switch to the Controller. This means that the more hops a message makes on its journey to reach the Controller, the more likely errors or timeouts occur. Let r (rank) be the number of switches (hops) in the path; then this probability could be expressed, for instance, as:

$$P_e(r) = \rho + (1 - e^{-\rho r})$$

Where ρ expresses the error rate for the switch directly connected to the controller.

The average number of steps t for a switch to reach the active state, calculated above, from the matrix, will be interpreted as the number of time intervals. For simplicity, this time will be associated to the mean time to a message to traverse a link between two switches.

From this reasoning, we produce another strategy for deterministically calculating the number of time intervals until a specific switch becomes active, that is, instead of sticking to the average of the probabilistic method above, we also calculate the exact number of time intervals for activation as a function of the switch rank.

Initially, let's think about the required number of sent messages until the activation of S2 from Fig. 3 takes place. The **`reg_req`** message is sent/forwarded three times up to

C, as the **reg_resp** message is sent three times downward. Similarly, the messages **upd_req** and **upd_resp** count six forwardings. That is, twelve message forwardings are needed until S2 has been activated. However, messages required to S0 and S1 activation also must be accounted for S2 activation.

Extrapolating the reasoning above, using simple induction and arithmetic series, one can deduce the number of messages sent $M(r)$ and the time intervals $T(r)$ required for activation of a specific switch with a generic ranking r:

$$M(r) = 4.(r+1)$$

$$T(r) = 4.\sum_{a=1}^{r+1} a = 2.(r+1).(r+2)$$

For this deduction, only messages that are not discarded were considered. For example, while a switch is not yet registered, at each time interval, it sends registration requests. That messages were not considered.

It is also worth mentioning that it was considered the best case to calculate $M(r)$ and $T(r)$, i.e. there are no errors or timeouts along the path from the switch to the Controller.

Finally, we can say that the Controller *sends* only two distinct messages, namely **reg_resp** and **upd_resp**, to activate each switch. Therefore, when analyzing Controller overhead, it is comparable to that offered by OFDPv2, which is an enhanced version of the OFDP protocol. According to [8], with OFDPv2, the number of Controller LLDP Packet-Out messages is reduced to just one per switch, or N in total, with N being the number of switches on the network. In our case, the Controller needs to respond two messages, that is, it will be $2N$, but still $O(N)$. The difference comes from the cost of in-band control channels.

About Packet-In messages received by the controller, yet according to [8], with OFDPv2, the number is proportional to active inter-switch links L; specifically, $2L$. But in our case, this number is always $2N$ too, that is, inferior to most cases.

V. Concluding Remarks and Future Work

The main contribution of this paper is to present an SDN bootstrapping control protocol and to specify how it configures a control plane within a regular (in-band) SDN infrastructure.

The protocol can build an ideal spanning tree as a logical multicast domain, bypassing closed loops on infrastructure connections. Network performance is not affected as the most intense message exchanges occur only at the time the control plane is formed. In addition, the deterministic manner in which the network is discovered is a desirable aspect for making predictions about performance and scalability.

A protocol for automatic bootstrap is very welcomed for resilience, as mentioned earlier. Also, the recovery time of the control plane is significantly lower compared to human intervention, thus reducing the cost of operation (OPEX). Another key aspect of being emphasized is safety because manual interference is restricted only to the moment the switch is physically connected to the network and powered on.

In addition to the analytical model presented in this document, future work should allow the implementation of a simulator to test **ConForm**. To do this, the OMNet++ [12] platform will be adopted.

In newer versions, the scope of the protocol should be expanded to consider the presence of multiple Controllers in the network and the cooperation between them.

Languages as P4 [11] introduced the notion of data plane programmability, enabling faster development of novel protocols. In this sense, future work is to use P4 to deploy a switch capable of treating **ConForm** primitives. This will be very useful to test, for example, scalability and timing questions of the bootstrapping.

Acknowledgment

This project was built with the support of MEHAR team. We would like to thank all who contributed with our research. This work has been partially funded by Brazilian agency CAPES.

References

[1] O. N. Foundation, "SDN Architecture Overview," ONF, Palo Alto, CA, Technical Recommendation Version 1.0, draft v08, 2013.

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

[3] A. Jalili, H. Nazari, S. Namvarasl, and M. Keshtgari, "A comprehensive analysis on control plane deployment in SDN: In-band versus out-of-band solutions," in *2017 IEEE 4th International Conference on Knowledge-Based Engineering and Innovation (KBEI)*, Dec. 2017, pp. 1025–1031.

[4] S. Abdallah, I. H. Elhajj, A. Chehab, and A. Kayssi, "A Network Management Framework for SDN," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, Feb. 2018, pp. 1–4.

[5] A. Kalyaev and E. Melnik, "Fpga-based approach for organization of sdn switch," in *2015 9th International Conference on Application of Information and Communication Technologies (AICT)*, Oct 2015, pp. 363–366.

[6] A. Azzouni, N. T. M. Trang, R. Boutaba, and G. Pujolle, "Limitations of openflow topology discovery protocol," in *2017 16th Annual Mediterranean Ad Hoc Networking Workshop (Med-Hoc-Net)*, Jun. 2017, pp. 1–3.

[7] L. Schiff, S. Schmid, and M. Canini, "Ground Control to Major Faults: Towards a Fault Tolerant and Adaptive SDN Control Network," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, Jun. 2016, pp. 90–96.

[8] F. Pakzad, M. Portmann, W. L. Tan, and J. Indulska, "Efficient topology discovery in OpenFlow-based Software Defined Networks," *Computer Communications*, vol. 77, pp. 52–61, Mar. 2016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0140366415003527

[9] A. Nehra, M. Tripathi, M. S. Gaur, R. B. Battula, and C. Lal, "SLDP: A secure and lightweight link discovery protocol for software defined networking," *Computer Networks*, vol. 150, pp. 102–116, Feb. 2019. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1389128618307916

[10] G. J. Holzmann, *Design and validation of computer protocols*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1991.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and others, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2656890

[12] A. Virdis and M. Kirsche, *Recent advances in network simulation: the OMNeT++ environment and its ecosystem*. Springer International Publishing, 2019, oCLC: 1082225057.