Visual Editor for Streamlining P4-based Programmable Parser Development

1st Muhammad Fajar Sidiq
Department of Informatics
Institut Teknologi Telkom Purwokerto
Purwokerto, Indonesia
fajar@ittelkom-pwt.ac.id

2nd Mega Pranata
Department of Informatics
Institut Teknologi Telkom Purwokerto
Purwokerto, Indonesia
mega@ittelkom-pwt.ac.id

3rd Akbari Indra Basuki
Research Center for Informatics
Indonesian Institute of Sciences
Bandung, Indonesia
akbari.indra.basuki@lipi.go.id

***Abstract*—P4 language enables new protocol development for advanced networking tasks such as dynamic monitoring, custom tunneling & routing, in-switch attack detection, and soon. Nevertheless, developing a P4 program is challenging for those who either lacking programming skills or advanced networking. This paper proposed a visual editor to ease the development of P4-based programmable parser which is the first and fundamental step in P4 development. The editor offers two functionalities, the creation of custom protocols & protocols parser, and code generation for testing purposes. For evaluations, we run a compatibility test to ensure that P4-switch can parse any packet using bit-by-bit extraction defined by the programmable parser. We develop custom protocols having the same bit structure as the standard protocols, such as Ethernet, IP, TCP, UDP, and MPLS, but use different naming. The result showed that the receiver recognized the custom protocols as the standard ones since the P4-switch treat them based on their bit structure. At last, the proposed program can ease and speed up the development of P4-based parser by providing visual editor, and network tester generator.**

***Index Terms*—visual, editor, P4 SDN, parser, new protocol**

I. Introduction

Software-Defined Networking (SDN) revolutionized network management and developments by proposing two main concepts: central network management and network programmability. OpenFlow is the pilot protocol that behaves as a southbound protocol for the central controller and the SDN switches. It lets the developer determine what actions to be applied for the matching packets [1]. With a centralized controller such as Onos [2], Ryu [3], and Opendaylight [4], OpenFlow enables central network management by installing the respective flow rules for each switch.

The second phase of SDN is marked by P4 language that enables the programmability of switches behavior covering how to parse and process the network packets [5]. The rise of P4 SDN is driven by the limited functionality of OpenFlow SDN and unsolvable consensus regarding what protocols that must be supported by the SDN switch. Instead of defining a fixed set of protocols, P4 language lets the user choose or develop their preferred protocol and how to process those protocol. Consequently, it satisfies both, the backward compatibility with the legacy and OpenFlow protocols and the flexibility of developing a new protocol to meet different use-cases. P4-based switch have been widely proposed to solve hard problems in networking, such as In-band network telemetry [6], heavy hitter detection [7], In-switch DDoS detections [8], and soon.

Despite its flexibility, developing a P4 program is hard due to its complexity and the need for extensive testing to ensure that the new protocol is behaving correctly. The complexity of a P4 program is derived from two parts, packet parser, and packet processor. Packet parser requires the user to specifies bit-by-bit protocol structure and how to parse those protocols in the right sequence. The sequence of packet parsers must follow the directed acyclic graphs (DAG) structure to avoid recursive parsing. In packet processing part, the user must determine the match-action table and the states that trigger those tables. Packet processing performs the computation based on the parseable header done by the packet parser. Therefore, it is crucial to ensure that the new packet parser is parsing the packet header correctly.

Refers to the aforementioned problems, it is necessary to ease the development of P4 Programs. Considering that the packet parser is the most critical part of a P4 program, this paper proposes a visual editor that specifically addresses the development of the programmable parser. Particularly in how to test the new parser to work as its purpose. Our proposed visual editor, called Visual Editor for P4-based Programmable Parser (VEP3[1]), breakdown the problem into two parts: visual-based protocol & parser editor, and code generator for testing purpose. The visual editor aims to bridge the gap in the programming skill by providing a visual editor for creating a new parser in the form of a DAG-based state machine. It serves as a visual abstraction layer of how a packet parser works. Meanwhile, the code generator provides testing codes to ease the evaluation of the newly developed parser.

Our proposed program focuses on lessening the learning curve of the P4 language by proposing a visual editor approach. Some existing works of P4-based programs have different functional-

[1]https://github.com/acbari/MiTE-SDN/tree/master/VEP3

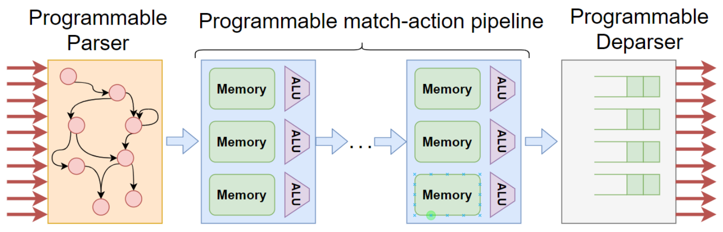


Fig. 1. Internal Structure of P4 Programmable Switch

ities compared to our proposed editor. The work in [9] which is called MiTE is our predecessor works that focus on visual topology editor for P4 and OpenFlow networks. Our proposed editor (VEP3) will be packed under the same bundle as MiTE. Other works [10]–[13] focus on the P4 program verification, validation, or cross-checking to find potential bugs and errors. The closest work is presented in [14] where it can craft packets tester given an input of a P4 file. However, the work did not propose visual editing likes ours. Other work as in [15] aims to assist the development of ASIC-based P4 Switch. Our proposed program targets specifically for emulation testbed, not the real hardware one, aiming for better accessibility of learning P4 language

We organize the paper into 4 sections. First we describe the P4 SDN and its complexity. Second, we describes the technical design of our program in chapter III. It pinpoints how the visual editor and code generator works. Chapter IV discusses the implementation results and evaluations. At last, chapter V presents the conclusion.

II. P4 SDN Complexity

P4 language proposed a programmable switch that can be tailored according to user preferences. The structure consists of three parts, a programmable parser, a programmable processor in the form of match-action pipelines, and a programmable deparser (Fig. 1). Nevertheless, the flexibility and programmability of P4 SDN come at cost of complexity, particularly for those who start to learn SDN or lacking in programming skills.

The authors define three aspects of P4 SDN complexity namely, P4 switch mechanism, the programming of P4 language, and the network testing. Even though the structure of a P4 switch consists of only three parts, the programmable parser becomes the real obstacle for those who do not have a firm knowledge of computer networks. It requires the user to defines which protocols are recognizable by the switch. Thus, the user must define the bit-by-bit sequences of every protocol and how those protocols interact with other protocols under a state machine structure.

The second challenge comes from the programmable packet processor. Besides using a similar match-action structure as the OpenFlow SDN, it has Arithmetic and Logic Units (ALU) that enable more complex arithmetic and logic operations compared to OpenFlow. If the user never learn OpenFlow SDN, it will be quite challenging to understand the programmable processor.

Programmable deparser is the easiest part yet trickiest one since it must be triggered from the P4 codes. Consequently, it can be considered as the second problem (the programming P4 language).

The first hindrance of programming a P4 switch is how to translate the protocol definition and their state machine structure into P4 language. It might be an easy task that can lead to catastrophic failure if not being carefully validated. Some works have proposed a validation tool to prevent error, bug, and invalid configuration [10]–[13]. In our paper, we minimize the error by guiding and limiting the user preferences through visual-based editing. The program ensures that the created parser always complies to the rules of P4 language. We elaborate this mechanism in Section III.

The second problem of programming a P4 program comes from the diverse types of P4 switches. To support vendor-independent approach, different P4 switch has difference feature that can be utilized by importing the module into the P4 code. Thus, the user should not expect that all P4 switches support similar capabilities and functionalities, such as cryptographic hash or encryption modules. The P4 environment lets the vendor advertises those features as their selling points compared to the others.

The latest problem lies in how to test the program in a running network machine. According to our knowledge, only the work in [14] that supports automatic code generation. Given a P4 program, it generated a packet tester to validate the program. Our work proposed a similar but distinct functionality. Instead of testing the new parser in bidirectional communication, our works proposed a compatibility test to validate that the P4 program can parser any packet based on their bit-by-bit sequence and state machine structure. We test the new parser to prove that a new and custom protocol having the same structure as the standard protocol can communicate to each other. Later on, the users can develop custom protocol precisely tailor for their needs.

III. System Design

The flow design of our proposed Parser Editor is shown in Fig. 2. It follows the step of packet parser development that consists of two main stages: parser drafting and testing.

Parser drafting deals with the creation of the underlying building blocks of a parser and how they are being organized as a single parser. Packet parser building blocks consist of two parts: the recognizable protocols, and the extra metadata.

Packet parser editor lets the user organize how the parser will extract the network protocols in bit-by-bit series. The parser editor guarantees the validity of the created parser by ensuring that it works under the rules of a directed acyclic graph (DAG) without a looping pattern. A cyclic parser might result in a never-ending packet parsing or any unintended security threats. To support a

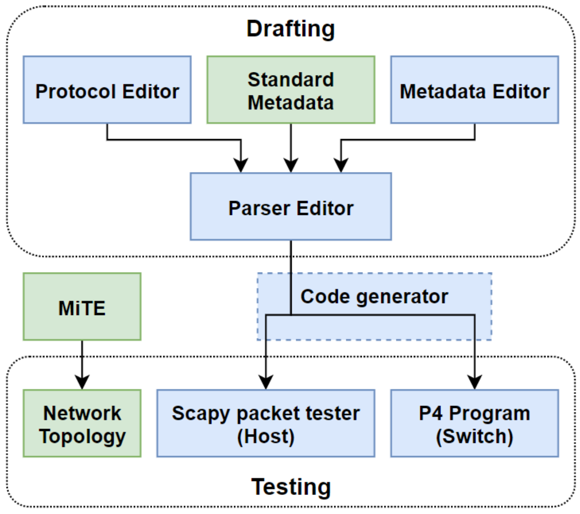


Fig. 2. The flow of programmable parser development and testing

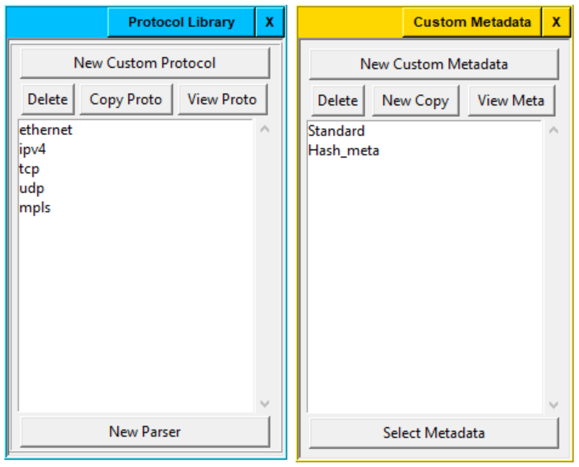


Fig. 3. The building block of a parser: protocols and metadata

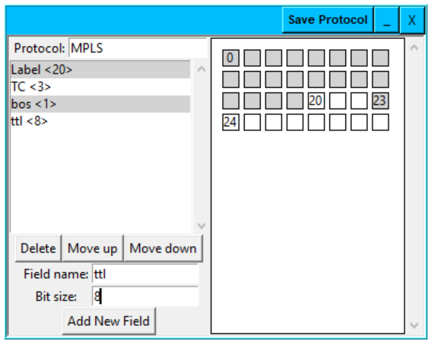


Fig. 4. Protocol drafting window and its visual structure

stacked protocol such as MPLS and VLAN, adding a self-loop is permissible. We called the structure of parser editor as DAGSeL, a combination of a directed acyclic graph with self-loop nodes.

Parser testing involves complete testing of packet parser within a real system, either in a hardware P4 switch or in an emulated one. The proposed editor generates two codes for parser testing: a Scapy-based protocol binding class, and a P4 parser code that can be inserted into an existing P4 program. For the testing, we run it using Mininet. The topology is generated using MiTE program [9] while the P4-parser code and Scapy-based packet sender are generated by the editor.

The following sub-sections explain the detail of the parser editor.

A. Building Blocks

The editor consists of two building blocks, the recognizable protocol, and packet metadata. Recognizable protocols refer to which network protocols are recognized and extractable by the P4 switch. For the sake of educational purposes, we simplify the editor by supporting only two types of protocol: fixed-size protocols, and the stacked ones such as MPLS. Meanwhile, the extra metadata functions as a temporary memory to assist packet processing. The user might create new metadata tailored for their purpose in how to process the packet.

We organize the building blocks into two libraries: protocols library, and metadata library (Fig. 3). Users can create new protocols or metadata through the library window.

In P4 language, each protocol is a set of bit-fields with the total size in bytes. For example, an Ethernet protocol is a set of three bit-fields, a 48-bits of source MAC address, a 48-bits of destination MAC address, and a 16-bits of ethernet type. The protocol's size must be divisible by eight. In this case, the total size of the Ethernet protocol is 112 bits or 14 bytes. Additional padding bits are automatically appended to the new protocol if the total size is not an 8-bit multiplier. To ease new protocol creation, we add the bit-structure visualization into the drafting windows (Fig. 4).

B. Parser Creation (DAGSeL)

The structure of a P4-based packet parser must satisfy a directed acyclic graph structure with an optional self-loop edge for every node (DAGSeL). Our proposed editor enforced the DAG rule during parser creation by forbidding a cyclic interconnection between parser nodes.

Given a directed graph (G) that represents a packet parser, the nodes (V) and edges (E) of G respectively represent the node parsers and the possible transitions between the nodes. Edge e_{ij} is transition between node parser v_i to node parser v_j, where $v_i, v_j \in V$. A node parser v_i specifically parses one network protocol (P_i) with one-to-one projection. A starting node (v_0) is a special node that has zero in-degree and does not represent any network protocol.

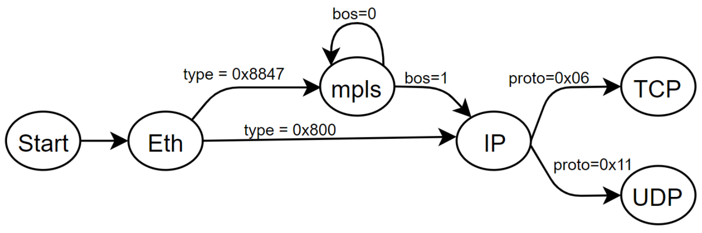


Fig. 5. Programmable parser graph: DAG + self-loop link

A packet parser is able to parse a specific protocol (P_i) if there is a directed path (D) between the starting node (v_0) to the node parser v_i that parse protocol P_i. The path D is a set of edges $e_k, k = \{0, 1, 2, ..., n\}$, where $n = |D| - 1$. The directed path D is considered valid if the edge in D satisfy two requirement as follow. First, the edge $e_k = (v_i, v_j)$ is continues, where $e_k[v_j]$ is equal to $e_{k+1}[v_i]$. Secondly, the edge e_k is a valid parsing transition where $e_k \in E$.

A packet parser is valid if every node parser (v_i) cannot reach itself, except via the self-loop edge (e_{ii}). In another word, there is no such directed path $d_a, |d_a| > 1$ where its starting node $e_0[v_i]$ is the same as its last node $e_n[v_j]$.

Figure 5 depicts the example of a packet parser that can parse five different protocols: Ethernet, MPLS, IP, TCP, and UDP protocol. The parser is a valid one since it does not contain any cyclic path except for MPLS self-loop edge.

The transition between node parser is controlled by the edge attribute (a_k), where each edge or node transition e_k has one attribute a_k. The attribute value is determined by the selector field of the predecessor node. We break down the P4 parsers into two different types: a simple parser, and a selector parser. As its name implies the simple parser does not has a selector field, thus the outgoing edge has a null attribute. Whereas for the selector parser, the user must specify the selector field and all of its transition attributes.

The TCP and UDP nodes in Fig. 5 are the simple node parsers while the rest of the nodes are the selector ones. Ethernet node parser has two outgoing edges to MPLS parser and IP parser with the ethernet type as the selector field. The respective attribute value for both transitions is 0x8847 and 0x800.

C. Parser Testing

The next phase of parser creation is to test the parser in a running setup. We opt for emulation setup by using Mininet. Three configurations are needed to run the emulation: the network topology, sender and receiver program for the hosts, and the forwarding codes for the P4 switches. For the topology, we use a simple topology of one switch and two hosts generated by using MiTE program (Fig. 6).

The remaining two configurations are generated by the proposed editor according to the program stack (Fig. 7). The code

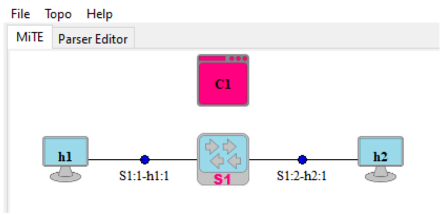


Fig. 6. Simple topology for testing new parser

generator translates the parser structure (DAGSeL) into a protocol binding class by using the Scapy library. The sender and receiver codes for the hosts are crafted by referring to this binding class.

Considering the editor only focuses on the programmable parser, it did not generate the entire P4 program but the parser codes. We modify the existing packet processing codes in [16] to form a fully functional P4 program. The editor replaces the parser code from the tutorial with the generated ones.

Parser testing acts as the ground truth for the new parser whether it had worked correctly or not. To ensure correctness, we use two kinds of testing, positive and negative testing. Positive testing proves that the parser can parse the intended protocols while negative testing proves that the parser is unable to parse unrecognized protocols.

We use five protocols for positive testing namely: Ether, MPLS, IP, TCP, and UDP protocols. For negative testing, we use ARP and ICMP protocols. The editor only generates the positive protocols. As a consequence, the P4 switch only recognized these protocols and cannot parse the ARP and ICMP protocol.

For the host's Scapy codes, we implement custom packet protocol by using protocol binding. We use the same structure as the positive protocol but with different naming. The new names for the custom protocols are in lower case letter as follow: ethernet, mpls, ipv4, tcp, and udp. This step is to test whether the P4-switch will consider the new custom protocol based on their structure or not. If it does, it will treat those protocols as the standard protocols. Thus, it will forward the packet and will be recognized by the receiving host. For compatibility testing, the receiving node (host-2) imports the positive protocols directly from the standard protocol.

IV. Result and Evaluations

A. Visual Editor

The final user interface of the proposed editor is shown in Fig. 8. The left-side pane contains the list of all the created parsers. Underneath, it shows the recognized protocols and custom meta-data used by the selected parser.

The right-side pane displays the parser structure (DAGSeL) that shows how the node parsers are interconnected to each other. The right pane also acts as the container for the drafting windows to create new protocols and metadata.

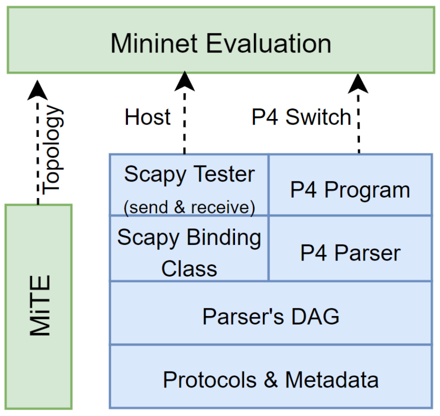


Fig. 7. Program stack for generating the testing codes

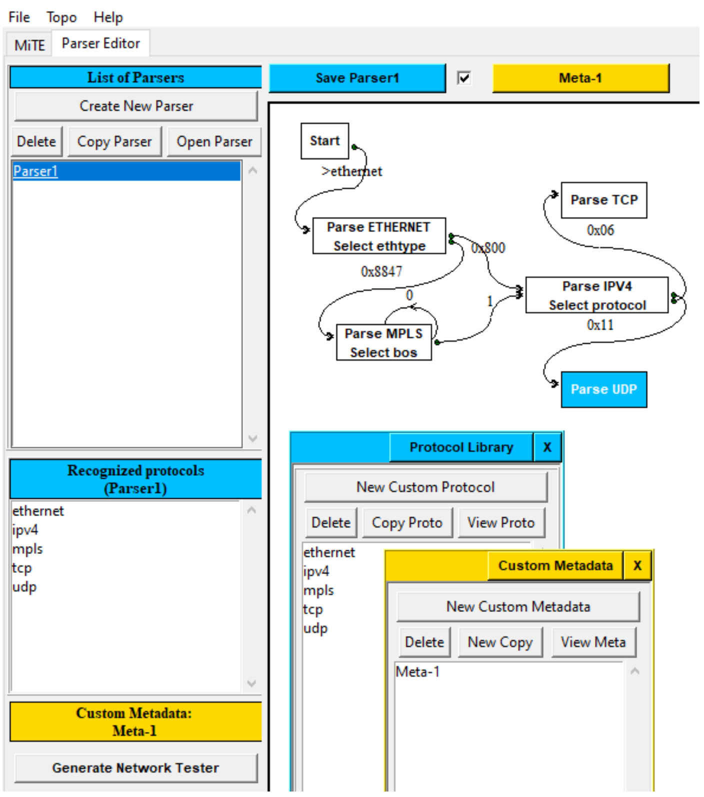


Fig. 8. The UI of the visual editor

B. Packet Forwarding Test

We tested the generated codes by using two routing mechanisms, IP routing, and MPLS forwarding. We modify the routing table of several project in [16] to form as in Table I. For each routing, we sent eight packets from host-1 to host-2. The summary of the forwarding test is listed in Table II. Fig. 9 and Fig. 10 shows the respective result for IP routing and MPLS forwarding. We also debug the setup to ensure that the packet is sent through P4-switch by dumping the ingress interface of the P4 switch (Fig. 11).

For the IP routing test, the P4 switch can parse all protocols except for ARP and ICMP. Consequently, the ARP packets are dropped since the switch cannot parse them. The switch also dropped MPLS packets since IP routing does not implement MPLS forwarding. However, different case is applied to ICMP protocol. In this case, the switch inabilities to parse ICMP protocol does not affect the routing. The switch can forward the packet by using IP forwarding rules since the ICMP protocol is an overlay protocol for the IP protocol.

In a P4-based programmable parser, there is no such layer numbering. Instead, the overlay structure is determined by the transition edges between the nodes. If protocol A is the *ingress* of protocol B, any routing applied to protocol A will *overshadow* protocol B. In this case, the switch will forward the packet even though the switch cannot parse protocol B.

In the MPLS forwarding test, the P4 switch has the same capabilities in parsing the protocol as in the IP routing test. Nevertheless, the switch cannot forward not only ARP packets but also IP packets. It happened due to MPLS forwarding does not implement IP routing. The switch can forward the MPLS packets to host-2 disregarding the overlay protocol that the packet carries on. This results due to the same consequence of *ingress* edge and *overshadowing* mechanism as in the IP routing test.

V. Conclusion

This paper proposed a visual editor for creating a P4-based packet parser that able to parse new protocols based on their bit-by-bit structure. The editor offers visual editing with zero-coding by offering an integrated code generator. It generates a P4-based parser and a set of codes to test the newly generated parser. The editor passed the compatibility test since the generated parser successfully recognizes and routes the packets with custom protocols as long as they have the same structure as the standard protocols.

References

[1] The Open Networking Foundation,“openflow-spec-v1.3.0,”Available at https://opennetworking.org/wp-content/uploads/2014/ 10/openflow-spec-v1.3.0.pdf (2021/05/01).

[2] Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., et al., ”ONOS: towards an open, distributed SDN OS”, In Proceedings of the third workshop on Hot topics in software defined networking (pp. 1-6), 2014.

[3] Tomonori, F. ”Introduction to ryu sdn framework.” Open Networking Summit (2013): 1-14.

[4] Medved, J., Varga, R., Tkacik, A., Gray, K., ”Opendaylight: Towards a model-driven sdn controller architecture,” Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014, IEEE, 2014.

[5] Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., et al., ”P4: Programming protocol-independent packet processors,” ACM SIGCOMM Computer Communication Review 44.3 (2014): 87-95.

[6] Van Tu, N., Hyun, J., Hong, J. W. K., ”Towards onos-based sdn monitoring using in-band network telemetry,” 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), IEEE, 2017.

[7] Sivaraman, V., Narayana, S., Rottenstreich, O., Muthukrishnan, S., Rexford, J., ”Heavy-hitter detection entirely in the data plane,” Proceedings of the Symposium on SDN Research, 2017.

[8] Friday, K., Kfoury, E., Bou-Harb, E., Crichigno, J., ”Towards a unified in-network DDoS detection and mitigation strateg,” 2020 6th IEEE Conference on Network Softwarization (NetSoft), IEEE, 2020.

[9] Sidiq, M. F., Basuki, A., Rosiyadi, D., ”MiTE: Program Penyunting Topologi Jaringan untuk Pembelajaran SDN,” Jurnal RESTI (Rekayasa Sistem Dan Teknologi Informasi) 4.5 (2020): 970-977.

[10] Rakamarić, Z., Emmi, M., ”SMACK: Decoupling source language details from verifier implementations,” International Conference on Computer Aided Verification, Springer, Cham, 2014.

[11] Liu, J., Hallahan, W., Schlesinger, C., Sharif, M., Lee, J., et al. ”P4v: Practical verification for programmable data planes,” Proceedings of the 2018 Conference of the ACM Special Interest Group on data communication, 2018.

[12] Neves, M., Freire, L., Schaeffer-Filho, A., Barcellos, M., ”Verification of p4 programs in feasible time using assertions,” Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, 2018.

[13] Stoenescu, R., Dumitrescu, D., Popovici, M., Negreanu, L., Raiciu, C., ”Debugging P4 programs with Vera.” Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. 2018.

[14] Rodriguez, F., Patra, P. G. K., Csikor, L., Rothenberg, C., Laki, P. V. S., et al. ”BB-gen: A packet crafter for P4 target evaluation.” Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos. 2018.

[15] Intel,“Intel®p4studio,”Available at https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html (2021/05/01).

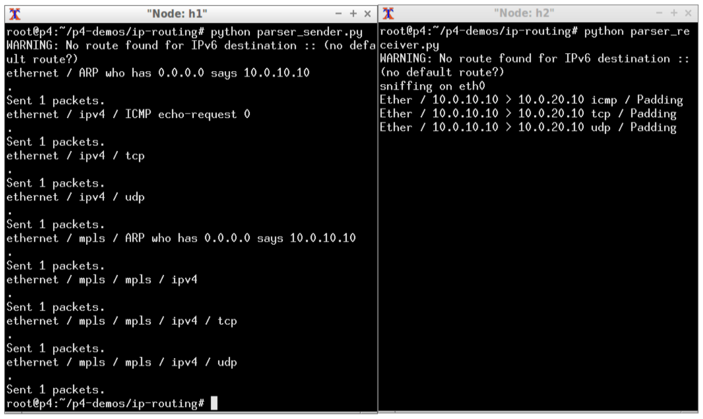
[16] Osiński, T.,“P4-Research/p4-demos: Prototyping networks with P4,”Available at https://github.com/P4-Research/p4-demos (2021/05/01).

TABLE I
ROUTING TABLE CONFIGURATIONS

Tables	Action	Match value	Out
mplslookup_table	forward	label = 100	Port2
ingress.routing_table	ipv4_forward	10.0.20.0/24	Port2

TABLE II
PACKET ROUTING RESULTS

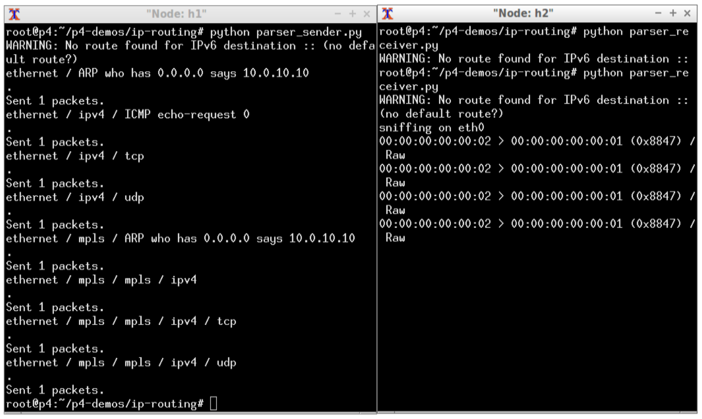
Protocols sequence					Routing	
					IP	MPLS
ethernet	ARP				X	X
ethernet	ipv4	ICMP			V	X
ethernet	ipv4	tcp			V	X
ethernet	ipv4	udp			V	X
ethernet	mpls	mpls	ARP		X	V
ethernet	mpls	mpls	ipv4	ICMP	X	V
ethernet	mpls	mpls	ipv4	tcp	X	V
ethernet	mpls	mpls	ipv4	udp	X	V



```
root@p4:~/p4-demos/ip-routing# python parser_sender.py
WARNING: No route found for IPv6 destination :: (no defa
ult route?)
ethernet / ARP who has 0.0.0.0 says 10.0.10.10
.
Sent 1 packets.
ethernet / ipv4 / ICMP echo-request 0
.
Sent 1 packets.
ethernet / ipv4 / tcp
.
Sent 1 packets.
ethernet / ipv4 / udp
.
Sent 1 packets.
ethernet / mpls / ARP who has 0.0.0.0 says 10.0.10.10
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4 / tcp
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4 / udp
.
Sent 1 packets.
root@p4:~/p4-demos/ip-routing#
```

```
root@p4:~/p4-demos/ip-routing# python parser_re
ceiver.py
WARNING: No route found for IPv6 destination ::
(no default route?)
sniffing on eth0
Ether / 10.0.10.10 > 10.0.20.10 icmp / Padding
Ether / 10.0.10.10 > 10.0.20.10 tcp / Padding
Ether / 10.0.10.10 > 10.0.20.10 udp / Padding
```

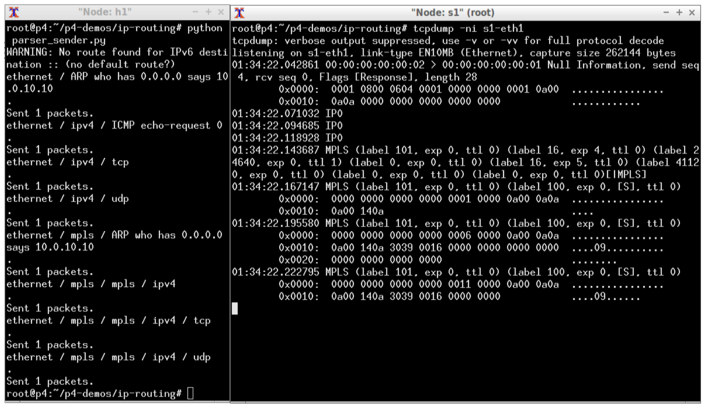
Fig. 9. IP routing from host-1 to host-2



```
root@p4:~/p4-demos/ip-routing# python parser_sender.py
WARNING: No route found for IPv6 destination :: (no defa
ult route?)
ethernet / ARP who has 0.0.0.0 says 10.0.10.10
.
Sent 1 packets.
ethernet / ipv4 / ICMP echo-request 0
.
Sent 1 packets.
ethernet / ipv4 / tcp
.
Sent 1 packets.
ethernet / ipv4 / udp
.
Sent 1 packets.
ethernet / mpls / ARP who has 0.0.0.0 says 10.0.10.10
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4 / tcp
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4 / udp
.
Sent 1 packets.
root@p4:~/p4-demos/ip-routing#
```

```
root@p4:~/p4-demos/ip-routing# python parser_re
ceiver.py
WARNING: No route found for IPv6 destination ::
root@p4:~/p4-demos/ip-routing# python parser_re
ceiver.py
WARNING: No route found for IPv6 destination ::
(no default route?)
sniffing on eth0
00:00:00:00:00:02 > 00:00:00:00:00:01 (0x8847) /
 Raw
00:00:00:00:00:02 > 00:00:00:00:00:01 (0x8847) /
 Raw
00:00:00:00:00:02 > 00:00:00:00:00:01 (0x8847) /
 Raw
00:00:00:00:00:02 > 00:00:00:00:00:01 (0x8847) /
 Raw
```

Fig. 10. MPLS forwarding from host-1 to host-2



```
root@p4:~/p4-demos/ip-routing# python
 parser_sender.py
WARNING: No route found for IPv6 desti
nation :: (no default route?)
ethernet / ARP who has 0.0.0.0 says 10
.0.10.10
.
Sent 1 packets.
ethernet / ipv4 / ICMP echo-request 0
.
Sent 1 packets.
ethernet / ipv4 / tcp
.
Sent 1 packets.
ethernet / ipv4 / udp
.
Sent 1 packets.
ethernet / mpls / ARP who has 0.0.0.0
says 10.0.10.10
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4 / tcp
.
Sent 1 packets.
ethernet / mpls / mpls / ipv4 / udp
.
Sent 1 packets.
root@p4:~/p4-demos/ip-routing#
```

```
root@p4:~/p4-demos/ip-routing# tcpdump -ni s1-eth1
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s1-eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
01:34:22.042861 00:00:00:00:00:02 > 00:00:00:00:00:01 Null Information, send seq
 4, rcv seq 0, Flags [Response], length 28
        0x0000:  0001 0800 0604 0001 0000 0000 0001 0a00  ................
        0x0010:  0a0a 0000 0000 0000 0000 0000            ............
01:34:22.071032 IP0
01:34:22.094685 IP0
01:34:22.118928 IP0
01:34:22.143687 MPLS (label 101, exp 0, ttl 0) (label 16, exp 4, ttl 0) (label 2
4640, exp 0, ttl 1) (label 0, exp 0, ttl 0) (label 16, exp 5, ttl 0) (label 4112
0, exp 0, ttl 0) (label 0, exp 0, ttl 0) (label 0, exp 0, ttl 0)[|MPLS]
01:34:22.167147 MPLS (label 101, exp 0, ttl 0) (label 100, exp 0, [S], ttl 0)
        0x0000:  0000 0000 0000 0000 0001 0000 0a00 0a0a  ................
        0x0010:  0a00 140a                                ....
01:34:22.195580 MPLS (label 101, exp 0, ttl 0) (label 100, exp 0, [S], ttl 0)
        0x0000:  0000 0000 0000 0000 0006 0000 0a00 0a0a  ................
        0x0010:  0a00 140a 3039 0016 0000 0000 0000 0000  ....09..........
        0x0020:  0000 0000 0000 0000                      ........
01:34:22.222795 MPLS (label 101, exp 0, ttl 0) (label 100, exp 0, [S], ttl 0)
        0x0000:  0000 0000 0000 0000 0011 0000 0a00 0a0a  ................
        0x0010:  0a00 140a 3039 0016 0000 0000            ....09......
```

Fig. 11. In-switch debugging using Tcpdump