IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 13 September 2024, accepted 26 September 2024, date of publication 8 October 2024, date of current version 21 October 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3476244

RESEARCH ARTICLE

An NFV-Based Framework for Autonomous Deployment of New Protocols in SDN Networks

ABBAS KHATER, SEYED AMIRMASOUD NOOHI, MASSOUD REZA HASHEMI, AND ZEINAB ZALI

Electrical and Computer Engineering Department, Isfahan University of Technology, Isfahan 84156-83111, Iran

Corresponding author: Massoud Reza Hashemi (hashemim@iut.ac.ir)

ABSTRACT Network programmability has always been a topic of interest to researchers. After introducing the concept of SDN, control-plane programmability is achieved. However, data-plane programmability remains a challenge in the context of network programmability. Several studies have attempted to deploy data plane programmability in different ways to accommodate and support arbitrary network protocols or to modify the currently supported protocols' functions. P4 introduced flexible parsing of arbitrary protocol header fields using software-programmable switches. In this study, we propose an autonomous SDN-based framework that achieves the same goal of flexible parsing of arbitrary protocol header fields in real-time using software switches based on the NFV concept. We describe a method based on the proposed framework that makes it possible to add proprietary network protocols or modify and upgrade an existing protocol. We introduce a software switch architecture that can implement data plane programmability based on the proposed method. In this switch architecture, the switches need not be pre-programmed to be able to parse and process the packets rather the desired parsing of the arbitrary protocol header fields and actions for each match are automatically programmed into the switches as needed. A proof-of-concept implementation verifies the expected operation of the method and shows that the network can be autonomously programmed in real-time, and new or customized protocols, above the link layer, can be deployed without network operator involvement.

INDEX TERMS SDN, NFV, network programmability, data plane programmability, P4, autonomous programmable network, software switch.

I. INTRODUCTION

The Internet has created a digital community with almost everything interconnected and accessible from anywhere. Despite their widespread use, traditional networks are complicated to manage. It is difficult to adjust the network based on pre-defined policies or reconfigure it to respond to faults and changes. What makes matters even worse is that the current network utilizes a vertically integrated architecture, where the control and data planes come together in the routers. The design and deployment of many network applications require flexible control that could provide better support for advanced services in addition to basic internet connectivity.

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

In many cases, achieving these goals requires custom packet processing in the data plane in addition to flexibility in the control plane. However, in traditional network architectures, changing anything in the IP requires the replacement of all data plane hardware. Consequently, over the past 20-30 years, several attempts have been made to refine the architecture of the internet network layer to provide efficient and dynamic programming of advanced network functions. To this end, many approaches have been explored, with varying degrees of flexibility and programmability.

In this context, Software-Defined Networking (SDN) has been proposed, which provides network programmability by separating network control logic from the routers and switches, improving network control focus, and providing new capabilities for network programmability [1], [2], [3], [4]. In SDN networks, the network control logic is

implemented in a controller that communicates with network devices. The most popular and common communication protocol between the controller and network devices is OpenFlow [5].

SDN focuses on control plane programmability, whereas the data plane is used mostly for packet forwarding with limited programmable options according to the OpenFlow protocol. OpenFlow only supports predefined protocols, has limitations in the flow table fields, and cannot flexibly add new header fields. The number of match fields has been continuously updated to add support for new fields such as IPv6 addresses, MPLS, and VLAN headers.

Adding new forwarding protocol features requires an overhaul of both the switch and controller, and in the worst case, results in a complete redesign of the switch chipset and hardware. Although OpenFlow has already covered more header fields, it has failed to support many well-known protocols. On the other hand, as new protocols continue to emerge (L2.5, L3, and L4 Protocols), the proliferation of new header fields shows no signs of stopping. For example, data center network operators increasingly want to apply new forms of packet encapsulation (e.g., NVGRE, VXLAN, and STT), so, according to [8] and [9] major revisions to OpenFlow standardization are expected, which makes the standard unstable and eventually unwieldy. In addition, vendors could not wait for OpenFlow standards to be developed. This instability leads to a slower pace of standard adoption. For this, network operators resort to deploying software switches that are easier to expand with new capabilities. The Open Network Foundation has not released any new OpenFlow protocol specification since 2015, and it seems that it is not evolving anymore, so development cannot continue in this way and new and more flexible solutions such as our proposed solution are preferred.

Nevertheless, the major problems that OpenFlow-based networks face can be summarized as follows:

1) Limitation of OpenFlow protocol in flexible packet processing rules.
2) Dependency on the OpenFlow protocol version to identify the header fields.
3) Scalability problem (e.g., controller processing power, flow table size, OpenFlow channel, etc.).
4) Focusing on control plane programmability rather than the data plane programmability.
5) Inability to process new protocol headers in packets.
6) Inability to forward packets of a customized protocol type over the network.
7) Dependency of OpenFlow-based switches on a fixed parsing of the header fields that supports only specific protocols.
8) The need to change the switches when the version of OpenFlow is upgraded.
9) Undesirable issue of rapid development and frequent updating of OpenFlow protocol for switch manufacturers and vendors.

Besides SDN, Network Function Virtualization (NFV) enables network operators to implement network elements, known as network functions, in software and run on standard server hardware. Each one of these elements is traditionally a separate hardware device operating on packets at the data plane. For example, in a network, the existence of separated devices such as firewalls, load balancers, etc. is common. It is evident that the presence of separate devices for each application is costly and has many management complexities [6], [7]. So, NFV aims to make networks more flexible, scalable, and cost-effective by decoupling network functions from dedicated hardware and running them as virtualized instances on general-purpose servers, often within virtual machines or containers. Implementing each of these components and network functions in the network as software, called Virtualized Network Functions (VNF), enables more efficient resource utilization, easier deployment and management, and greater agility in scaling and adapting network services to changing demands, as they can be deployed, scaled, and managed independently.

Deploying NFV is more trivial in SDN-based networks. NFV complements SDN (and vice versa), but they are not dependent. NFV can also be implemented without SDN, though the integration of SDN and NFV technologies will yield better results in function.

With the emergence of new insights such as SDN, NFV, and data plane programming, networks are becoming more flexible in terms of network protocols and even functions beyond the current protocols at the network level [8], [9].

Network programmability, or more accurately, control plane programmability has been achieved by SDN. However, data plane programmability is still a challenge in the context of network programmability. So, as the networks become more flexible and more functional than before, and because of the expectations from the new networks, new protocols are being proposed in addition to existing ones. Being able for network operators to utilize different protocols and even proprietary protocols in their networks, gives it a great advantage and makes it even more functional and flexible. In this case, the network can be programmed in the data plane so it has high flexibility to support arbitrary protocols. For this, rather than repeatedly extending the OpenFlow specification, the researchers have argued that future switches should support flexible mechanisms for parsing packets and matching header fields, allowing controller applications to take advantage of these capabilities. In this regard, several studies have tried to deploy data plane programmability in the network, in different ways. Some of the research reported in the literature has focused on deploying data plane programmability on software switches, such as POF [8], and P4 [9]. In these solutions, the boundaries of the packet header fields are programmed in the switches by the software executed in the switch. Then the switch identifies and parses the header fields accordingly.

Following the path of POF, and then P4 in solving the OpenFlow problems by flexibly parsing the header fields, in this paper, we propose a framework based on a different

approach from POF and P4, that uses a novel method to achieve the same goal of flexibly parsing the header fields in the switches. We also use software switches however with the novel capability that allows the controller to seamlessly deploy new protocols or newer versions of existing protocols in the switch. In this proposed method, unlike P4, the switches do not need to be pre-programmed to be able to forward packets of new protocols, instead, if a new packet of a new protocol arrives at a switch, the parser module is configured in real time by the controller through fetching a set of parameters from a server and setting them in the switch. These parameters simply identify the boundaries of the packet header fields according to the desired protocol. Therefore, with these parameters, the packet header fields are specified, then the packet can be processed and routed by matching the parsed fields and performing the corresponding actions. Also, new actions can be fetched, in the same way to operate on the processed packets without requiring new hardware or manually programming the switch.

In addition, it is noticeable that the proposed method is backward compatible with the legacy SDN networks because it can deploy conventional network protocols and process their packets, in the same manner.

The capability of autonomously implementing on-demand changes in the switches results in a dynamic and adaptive network where new protocols with arbitrary packet handling can be deployed in the data plane. As a result, using such switches in the network helps reduce OPEX and CAPEX by reducing the deployment time and operating costs of changes, as well as saving on replacement costs of the equipment itself.

In this proposed method for data plane programmability, initially applicable to protocols above the link layer, we achieve the following advantages:

1) The ability to quickly and easily define new network protocols or features in existing protocols without changing the switch hardware.
2) Independency from particular OpenFlow-based switches that otherwise have to be changed when the network protocol is upgraded or changed.
3) Need not to pre-define the protocols in the switches.
4) The ability to automatically deploy new protocols in the network without involving the network operator.
5) The compatibility with current and existing protocols in the network.

In the remainder of this paper, in section II, an overview of related works is presented. Then the proposed framework is described in detail in section III. An implementation of the framework is presented in section IV. In section V some experiments are shown as a proof of concept and the proposed method is evaluated. Finally, section VI, concludes the article.

II. RELATED WORKS

In SDN networks with the OpenFlow protocol, the focus is on control plane programmability. The data plane in these networks merely receives match rules and action commands from the control plane and applies them to the arriving packets. Because of OpenFlow's inability to flexibly add new header fields, researchers made suggestions on how OpenFlow should evolve in the future. Multiple new header fields have been added to OpenFlow incrementally, and the tables have become multipath. However, the demand to add new header fields to support newly designed protocols and features continues to grow. So, it has been suggested that instead of repeatedly extending OpenFlow's features, something should be done to make future switches support flexible mechanisms for parsing packets and matching header fields.

Consequently, several solutions were proposed to achieve data plane programmability some of which focused on designing programmable switches. Some of the designed switches are based on hardware [11], [12], [13], and some are based on software [14], [15], [16], [17], [18], [19], [20], [21], [22], [23]. Even some new architectures were proposed for data plane programmable switches [24], [25], [26]. Also, a new network architecture called Deeply Programmable Networks was presented in [27] and [28].

Among the various approaches for data plane programmability solutions, an effective one is designing data plane languages for programming the data plane [8], [9], which results in considerable flexibility in the network. According to [9], recent chip designs show that such flexibility can be achieved with specific ASICs at terabit speeds. However, programming this new generation of switches based on these chips is very complicated because each chip has its low-level interface, similar to microcode programming. Therefore, some researchers believe that future generations of OpenFlow should allow the controller to tell the switch how to work and operate, rather than being limited to specific switching capabilities. In this way, POF [8] and P4 [9] languages have been introduced to program the network switches arbitrarily and flexibly.

POF is a data plane programming language that aims to remove any dependency on protocol settings in switches and provides the possibility of implementing new functions at the data plane. Network switches using POF do not need to know the format of the packet and do not know about any protocol type. However, their task based on controller instructions, is only extracting and collecting search keys from the packet header, performing table searches, and then executing related instructions. The controller extracts the packet header fields through a sequence of public key collection and search instructions in the table, so the packet parsing is directed by the controller through a sequence of generic key assembly and table lookup instructions. A search key is simply defined by one or more tuples {offset, length}, where the offset represents the start bit of the key in the packet, and the length represents the number of bits of the key. The southbound interface defined in POF, can be treated as a promising enhanced version of OpenFlow. The limitation of this method is that it still depends on a set of instructions supported by the switch. Therefore, it is not possible to perform operations and processing on packages that are not supported by existing instructions. Considering these limitations, a more

comprehensive solution is required to deploy a desired protocol and perform a desired operation in a convenient software environment without the need for specific hardware or software that supports particular instructions.

P4 is a high-level programming language to program protocol-independent packet processors, that works in conjunction with SDN control protocols like OpenFlow protocol [9]. The idea of P4 is that, rather than have the switch tell us the limited set of things it can do, P4 gives us a way to tell the switch what it should do, and how it should process packets. P4 lets us define what headers a switch will recognize or parse, how to match on each header, and what actions we would like the switch to perform on each header [29]. In other words, P4 defines new parsers in the switches to be able to parse the packet header fields of new packets to match their values with the flow table entries. The flexible parsing in P4 is done by programming the switches. In SDN-based networks with OpenFlow, P4 tells the switch how packets should be processed and creates the flow tables while OpenFlow is still used to fill the flow tables. In this way, OpenFlow and P4 can work together. While OpenFlow is designed for SDN networks in which the control plane is separated from the forwarding plane, P4 is designed to program the behavior of any switch or router, whether it's controlled locally from a switch operating system, or remotely by an SDN controller [29].

When designing the P4 language for hardware-based switches three goals were intended: (1) reconfigurability, which is achieved by the ability of programmers to determine the method of parsing and processing packets in the switch, (2) protocol independence, and (3) target platform independency, which enables the programmer to program data plane independently of existing hardware specifications. This means that the compiler should take care of the details of the switch and consider its capabilities while compiling the controller program to a switch program. Later, to better achieve the P4 goals, they introduced a new programmable software switch, called PISCES [18] that supports the P4 language and can run the compiled programs of P4. Recent reports on designing software switches [30], [31], [32], [33], [34], [35], [36], [37], [38], [39] show that software switches on sufficiently high link speeds can achieve desirable processing speeds. But, with all the benefits that P4 has, it lacks an autonomous approach to program the network automatically without network operator involvement which requires more time and effort.

BPFabric [10] is another SDN-based architecture for implementing arbitrary functions at the data plane that cannot be achieved using OpenFlow, ranging from traditional routing and forwarding functions to more complex middlebox-like functions, such as statistic gathering and reporting, packet tracing, network telemetry, load balancing, and anomaly detection. BPFabric operates independently from protocol, target platform, and language. Functions are implemented through a High-Level Language (HLL), then converted to eBPF instructions through the compiler and installed on the switch. However, one of the limitations of this approach is that it is still dependent on a set of instructions, and the operations need to be supported by the instruction set.

Following these proposed methods, a programmability enhancement for SDN networks using POF was reported in [40]. In this method, POF is leveraged to enhance the network programmability further such that the forwarding plane becomes protocol-independent and can be dynamically reprogrammed to support new protocol stacks seamlessly. They achieve this by developing a POF controller, using the POX controller with some additional features added to it, and designing and implementing a software-based POF switch, by leveraging the Intel data plane development kit (DPDK) [41].

PNPL [42], is a framework at the control plane for programming the entire SDN POF data plane, that allows users to program high-level network policies over the data plane. PNPL provides an easy-to-use programming paradigm that includes a header specification language and a set of protocol-agnostic programming APIs. With PNPL, a programmer can arbitrarily define network protocols and compose network policies over the self-defined protocols with high-level abstractions, without the need to configure individual POF switches explicitly, but only describe the protocol header formats and network-wide forwarding policy. PNPL's runtime system takes the user program as input and automatically produces and maintains forwarding pipelines in POF switches. The pipeline efficiently parses the self-defined protocol headers and enforces the programmer's network policy.

In all of the proposed methods, there is a common limitation which is that the networks must be programmed in advance to send packets, and if a packet from a new and arbitrary protocol arrives at a switch that has not been programmed before, the packet cannot be processed.

Compared to the above solutions the framework and the switch architecture proposed in this paper aim to remove the hardware limitations and programming instructions dependency for the parsing of the packet header fields by running a software-based parser module with adjustable parsing boundaries which are configured automatically in the switch by the controller on the fly as needed, and along with the deployment of custom software modules, to define the new protocol on the switch.

III. PROPOSED FRAMEWORK

This section describes the proposed framework to achieve autonomous data plane programmability. As the basis of the proposed framework, an SDN-based network is considered with a controller that manages the switches in the data plane.

The proposed framework consists of multiple layers that are described briefly in section A as a high-level view of the framework, then each layer is described consecutively in the 3 subsections B, C, and D. Then the packet structure considered in this framework is introduced in subsection E. After that, a simple scenario about packet processing and forwarding is described in subsection F to show the steps

it takes for a packet to pass the network. Finally, the use of the NFV concept in the proposed framework is justified in subsection G.

A. HIGH-LEVEL VIEW OF THE FRAMEWORK

We introduce an NFV-based service layer on top of the existing SDN network, as seen in Fig. 1, called the Protocol Definition Layer (PDL). PDL consists of an overlay network of Protocol Repository server (PR) holding information about any network protocol, old or new, that is expected to be needed by the network along with their related actions that can be executed on the packets in the switches. The controllers access the PR when needed and get the required protocol information, including the Packet Parser parameters (PP). PP consists of the parameters needed to identify the header fields of the packets, whose role is to define and parse the packet header fields of the respective protocol. Because each protocol has its own header fields, so each protocol has its own parameters.

The traditional control plane of the SDN networks in this framework has the extra capability to allow the addition of new protocols by defining their parsing parameters, their match rules, and related actions. The parameters and match rules and actions for a new protocol are programmed into the control plane as a VNF. The parser module is executed with these parameters in the controller where it operates solely on the first received packet of any new flow that is used to determine the route.

The switches in this framework are also programmable by configuring their packet parser's parameters. These parameters are programmed into the switches by the controller. In addition, the match rules and their related actions are also programmed into the switches as VNFs. The parsing module on the switch with the parameters acts on every incoming packet of the same protocol type to match it with the flow entries in the flow tables. This parser supports various actions performed on the incoming packets for different purposes according to the relevant network protocol. The proposed network switches used in this framework are software-based switches that provide considerable flexibility in the network. As mentioned earlier, according to recent studies desirable processing speed can be achieved with software switches with sufficiently high link speeds, therefore, the proposed switches can be used in all access networks and most enterprise networks where data plane programmability and flexibility are needed the most.

Briefly, the control plane in this framework is programmed based on the concept of Network Function Virtualization (NFV), so that the actions and instructions performed on the packets of a specific protocol can be added as a VNF to the controller. Then, the network switches in the data plane are programmed by the controller to handle new protocols and perform new actions and instructions over the passing packets.

The communication between the controller and the switches is done through an appropriate communication

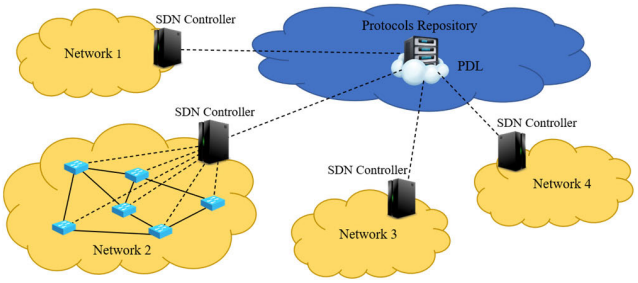

FIGURE 1. Proposed high-level framework.

protocol, somehow similar to OpenFlow in terms of its messaging system that supports our proposed features.

For simplicity, initially, we focus on arbitrary protocols in the layers above the link layer assuming that Ethernet is used in the link layer. Nevertheless, Ethernet is the dominant and common link layer protocol at least in wired networks and main network technologies nowadays rely on using Ethernet protocol [43], [44]. So, the packets are expected to have the same layer 2 header (Ethernet header) as in the traditional packets whose "*ethernet type*" field is used to identify the protocol type of the next layer. So, the switches should have the Ethernet protocol module built-in to have the ability to process the Ethernet header of the incoming packets.

In practice, when a packet with a new protocol type arrives (like MPLS, VXLAN, NVGRE), the switch identifies its protocol type through the "*ethernet type*" field included in the layer 2 header and then sends it to the corresponding parser. But if this protocol type is not yet defined, the switch requests the relevant protocol PP from the controller. The controller extracts the protocol type ID from the packet, coded in the "*ethernet type*" field, and fetches its corresponding parameters from the PR. Then, it runs the parser module to process the packet using the received parameters that specify its fields and consequently determines the path. After that, the controller sends the received protocol parser (PP) to the switches on the path. The switches also store the PP and use it to parse and process packets of that type.

B. SWITCH ARCHITECTURE

The proposed switch used in this framework is a software switch that runs a set of modules to process the arrived packets. The first one of these modules is the built-in module to parse the layer 2 header, especially the Ethernet header, and extract the "*ethernet type*" field to specify the protocol type of the next layer. The second module is the parser module that gets parsing parameters as input to identify the next layer header fields. The other modules are the instructions and actions performed on the packets. Some of these actions are generic actions that are designed with a switch and performed in all types of packets (e.g., forwarding packet to output port) and some of them are custom actions that are performed on corresponding protocol, according to its fields. These custom actions are loaded from the PR server when the protocol is deployed on the switch. A separate memory space

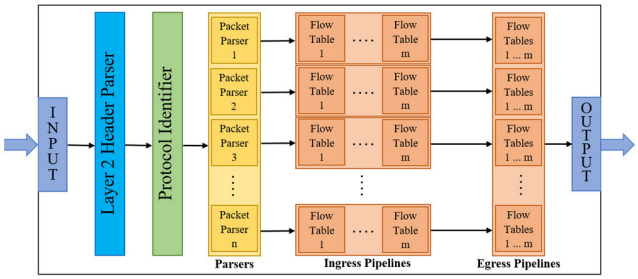

FIGURE 2. The block diagram of the proposed switch.

is dedicated to each protocol for maintaining a table of flow entries containing matching fields and their relevant actions.

The main building blocks of the proposed switch architecture are illustrated in Fig. 2. As can be seen, the switch comprises the *layer 2 parser* to parse the layer 2 header, especially the Ethernet layer header, and fetch the protocol ID from the "*ethernet type*" field, a *protocol type identifier module* with a *protocol identifier table* for detecting the protocol type of received packets using ID, and several stored processing parameters, called *Packet Parsers (PPs)*, for parsing the packets' fields. Also, attached to each PP, the memory space needed by the module where the *flow tables* are stored is shown.

With the arrival of the first packet of any protocol type, the Ethernet layer is parsed and the protocol ID is fetched. If the protocol is not previously defined, a copy of the packet is sent to the controller requesting the PP of this type of packet. Then the switch receives the PP from the controller and stores it. Flow tables according to the new protocol are also created and stored. The protocol identifier table is also updated, adding a new protocol type along with the protocol ID code. Then with the arrival of the next packets of this type of protocol, the switch will be able to parse and process the fields of those packets. The parsed header also has a field that specifies the next layer which also in turn may require its specific parser that can be fetched in the same way recursively. While in this paper we will focus only on the protocols between layers 2 and 3 as proof-of-concept examples.

In summary, for the switch to handle the protocols, old or new, the switch should have the corresponding PP, and if it is not deployed in the switch yet, the switch will request it when required, so, when each packet enters the switch, its protocol type is specified using the *protocol type identifier module*. Then it is sent to the parser module with the corresponding parser parameters. Finally, the fields are specified and assigned to the relevant pipelines along with Ethernet header fields for matching, and the corresponding action of the matched field in the flow entry is applied.

The switch operation process on each incoming packet is shown in the flowchart in Fig. 3.

C. CONTROLLER MODULE

The controller in the proposed framework not only performs the normal functions of conventional SDN controllers, but it

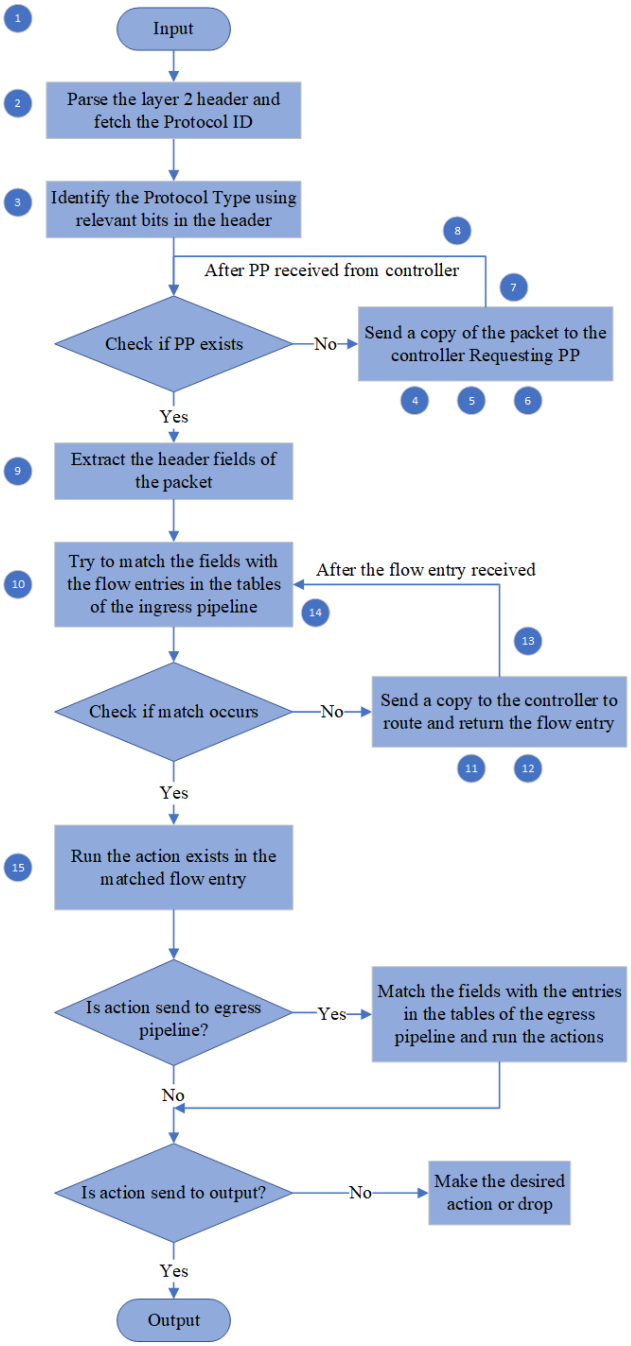

FIGURE 3. Flowchart of the operation process inside the switch.

also performs some additional and specific basic functions required for the data plane programmability using a dedicated module. These specific functions are, *Protocol Identifier* that detects the type of the protocol of incoming packets and sends them to the corresponding PP, *Protocol Fetcher* that fetches the required PPs from PR and stores them locally by sending a request to the PR and waiting for the reply through the Northbound Interface, the *Protocol PPs*, and the *Data Plane Programmer* that programs the switches by sending the fetched PP to them through the Southbound Interface. Fig. 4, shows the block diagram of the proposed controller module.

The main task of the controller module is programming the software switches. For this purpose, after fetching and storing the required PP, the controller updates the *protocol identifier table* by setting the address of the PP for this new type of packet and sends a copy of this PP to the network switches through the communication protocol. Briefly, this controller module adds the PP to the switch and allocates memory for it to have the ability to parse the new type of packets. It is worth noting that the controller module refers to the PR server only once for each new protocol, since the controller after getting the new protocol parser, saves it and will not refer to the server again. Also, the switches will not refer to the controller for this issue after getting the parser.

One of the basic operations of any SDN controller is routing. Hence, the controller eventually performs routing based on the packet header fields specified by the related PP. Furthermore, to fill the flow tables in the switches in the network, the controller must find the route and then add the considered packet information in the tables of the switches across the route as a flow entry. So, after making routing decisions using PP, the controller generates a flow entry to match the correspondent packets and sends it to the switches on the path to add it to their flow tables.

D. PROTOCOL REPOSITORY (PR)

Protocol Repository is an NFV-based server that is used for maintaining the packet parsers (PPs) for all protocols used in the network. To introduce a new protocol, it is required to register its PP in the repository server. A unique protocol ID is assigned to each PP through this registration.

To register a new custom protocol in the PR, the protocol designer must register its PP in the repository along with its custom actions. Then the repository assigns an unused protocol ID (PID) code to it. The assigned PID code is listed in the table and is considered an ID code for the new protocol. After that, when using this protocol, the relevant PID code must be indicated in the packet's header. Also, if an update or new version for an existing protocol is released it will take a new ID as happens in the traditional network (e.g. IPv4 and IPv6 each get a new code that is 0×800 and $0 \times 86DD$ respectively).

Briefly, the PPs of various protocols are stored in this repository and accessed through the ID code assigned to the protocol in the registration phase. Each packet must contain this ID in the "*ethernet type*" field in its header, indicating its protocol type.

To fetch a PP from the PR, each controller has to recognize the repository server and send it a protocol lookup request using the PID. If the requested protocol is found, it will be sent to the controller as a response, while if it is not found, that means that the requested PP is not registered yet, an error message response is sent to the controller.

The PR can be incorporated with the controller in enterprise or local networks, especially for proprietary protocols. For a broader scope for global protocols and coverage, a cloud-based approach or any implementation of the PR as an overlay will be necessary.

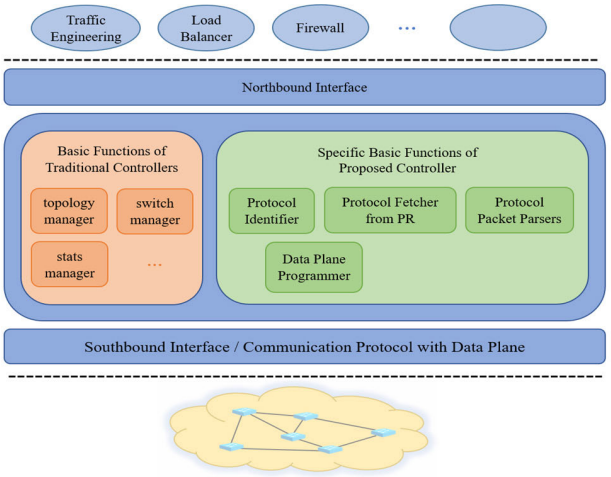

FIGURE 4. Proposed controller block diagram.

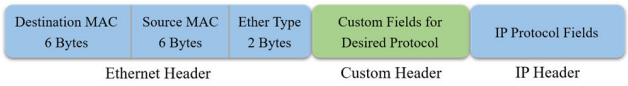

FIGURE 5. Packet structure of a new designed protocol.

E. THE PACKET STRUCTURE

In the proposed framework the switches can process packets of an arbitrary network protocol by adding the PP of the protocol to the PR. The PP specifies the header length and its fields, which vary according to the protocol. For example, MPLS, VXLAN, and NVGRE packets each have different header fields and lengths from others. The software implementation of the switch should flexibly support the arbitrary size of the packet, its header, and the header fields.

A typical packet's structure is shown in Fig. 5. As seen in the figure, the packet has the conventional Ethernet layer and IP layer headers, with another layer's header between them, which is designed based on the concept of the desired protocol defined by the protocol designer. It can include any arbitrary fields according to the designer's intent and goal. Some of these fields are used to route the packet across the network.

The "*ethernet type*" field in the Ethernet header of the packet is suggested to be reused for the PID code to be added to the packet. The PR assigns the unused codes of the "ethernet type" field to new protocols. In this way, the header fields defined in the next layer header of the packet are processed using the PP related to the specified PID code that is invoked to extract the fields of that layer.

F. PACKET FORWARDING SCENARIO

Here, according to the flowchart in Fig. 3, we describe the scenario of packet processing and forwarding, which occurs in the proposed framework, in steps as follows:

1) The first packet of the flow enters the switch.

2) The Ethernet header is parsed and the PID that exists in the "*ethernet type*" field is fetched.
3) The protocol type identifier module specifies the protocol type of the packet (flow).
4) If the protocol type is not previously defined in the switch, the packet is sent to the controller along with a request for PP for this protocol type.
5) The controller specifies the protocol type of the packet and fetches its PP from the PR server if available.
6) The controller also performs the routing mechanism and makes decisions on actions required for these types of packets (flow).
7) The controller sends the required PP of this packet to the switch; then, the PP is stored in the switch.
8) The controller sends the matching fields for these packets (flow) with the required action(s) to the switch(es) and inserts them into the corresponding table.
9) On the other hand, in step 4, if the protocol type is already defined in the switch, the packet is sent to the parser module along with the corresponding PP parameters for processing and extracting the fields.
10) After the fields are extracted, they are matched with the flow entries in the relevant flow tables.
11) If no match is found with the flow entries in the tables, then the packet is sent to the controller to be routed.
12) The controller performs routing and makes the required decision for that packet and the subsequent packets of the same flow.
13) The controller sends the matching fields for packets of this flow with the required action as a flow entry to the switches and inserts them into the corresponding table.
14) Then the fields are matched with the flow entries.
15) If matching occurs, then the action(s) intended for these packets of the flow are applied.
16) However, if the required PP is not found in the PR, the packets of that type will be dropped.

G. NFV CONCEPT IN THE PROPOSED FRAMEWORK

In the proposed framework, the concept of network function virtualization has been used to deploy new network protocols and their related actions in the controller. The network protocols deployed in the controller are also deployed in the switches by the controller and the related actions are used in the switches to be performed on the passing packets.

In OpenFlow design, actions are added to the flow entries in the flow tables in the switches, and executed on the matched packets. In this case, new actions come with new releases of the OpenFlow, and the switch implementation should be updated and changed to support these new actions. In our proposed method, there is no need to change the switch because these actions are software-based modules related to the PP of each protocol, and they can be added or updated at any time.

This model of performing various actions on matched packets makes the network more flexible such that it can support changes on protocols and adding new actions without changing the switches.

IV. IMPLEMENTATION ISSUES

In the implementation of the proposed switch architecture, the switch should be able to read the packets from the network card, which in turn, needs low-level access in the hardware, and then process them according to the relevant protocol.

Consequently, the switch reads the packets' bytes from the network card, then extracts the fields using its related parsing module and applies the specified action(s) by matching the fields with the flow tables.

We implemented the proposed switch using the DPDK library, which can be used in several environments. Using DPDK, we developed a module with 2 functions, to send and receive packets in bytes to/from network card (port). Along with that module, we developed a component in the switch that uses the developed module and assigns an instance of it in parallel (using a thread) for each port in the switch. Besides that, the parser module, which is used to extract and process the packets using parameters, exists and is considered as a class, that consists of 4 sub-classes: (1) extracting fields from the packet (parser), (2) encapsulating fields on a packet (packet builder), (3) processing the packet in the switch (matcher), (4) processing the packet in the controller (router). The processing function class in the switch, that is called matcher, is used for matching the fields of the packets with the flow tables and only exists in the switch, while the processing function class in the controller, that is called router, is used for routing the packets and only exists in the controller. The parser's parameters, which we named them PPs, are stored in the PR. An example of these PPs that is related to the IP protocol, can be found in [45].

As a result, the proposed switch that we call NSwitch is a fast and flexible programmable software switch with a class for each protocol that consists of the required parameters to process the related fields. The switch is programmed by defining new classes. For each class, the related PID code is added to the identifier table in the switch and the related PP is added as a class to the class list. The implementation source code of the proposed switch is available in [46]. In this switch, the dependency on compilers is removed and a dynamic storage to store protocol lists, as well as flow tables, is built in. With these dynamic tables, which can be seen in Fig. 6, each row of the tables can be managed by the controller in the SDN infrastructure, and finally, this feature makes it possible to add/edit/remove protocols in real time.

As it is depicted in Fig. 6, each row of a protocol table consists of two parts. The first part is the key, which contains a unique code, and the second part is the value part, which consists of information about the protocol layer number, the layer structure, the location of each part of the layer in the match-action table, and a number which indicates whether there is a next layer to extract or not. After we extract the packet completely, we should match the packet fields with the match fields in the table in Fig. 7.

In the flow table in Fig. 7, there is a key and value, where the key is the hash of the value, while the value contains two parts that are the match and the operation that should be performed on that packet. The numbers represent the number of bits for each header that are used in the matching process.

Also, multiple techniques have been used in the implementation of this switch to achieve a very high speed and eliminate possible overheads. These techniques include:

1) Using pointers to avoid packets being copied at each stage.
2) Extracting only part of the layers that are needed for routing.
3) Calculating the checksum of packets only in the output port.
4) Optimizing coding techniques such as using move semantics.

Over the network consisting of proposed switches, we implemented the proposed controller to control the switches. The proposed controller has the same PPs required in the switch, for extracting and processing the packets of defined protocols. One of these modules, which is specific for the controller is a module to find out the destination and route the packets. So, to route a type of packet, the protocol type should be identified. To identify the protocol type of the packet, the controller uses the identifier table and finds the required protocol. After that, the controller has all the data needed to process the packets of defined protocols to make routing decisions.

Also, we implemented an essential element in the proposed framework, which we named PR, which stores all the defined network protocols. In PR, there is a table to identify the protocols. The table is formed of 2 fields: (1) the protocol type ID that is used in the "*ethernet type*" field in the Ethernet header of the packet, and (2) the PP of the protocol along with its related actions which are designed by the protocol designer who wants to use it in the network.

However, a third field can be added to the table to specify the layer that the desired protocol is proposed for, so more layers can be supported hierarchically, and the match-action process can be performed on them. For example, in the case of IP/MPLS, the MPLS layer can be deployed in addition to the IP layer, and in the case of SST, the SST layer can be deployed in addition to the Transport layer.

Finally, in the proposed framework, the communication between the switches (data plane) and the controller (control plane) is done through an appropriate communication protocol, somehow similar to OpenFlow in terms of its messaging system, with new messages and features to support our proposed framework. In the communication protocol, both sides use standard messages to have the ability to exchange data with each other. We propose a generic communication protocol to make communication between the controller and switches independent of the network protocol design and its parsing fields and actions. We suggest two significant types of messages in this protocol that are sent from the switches to the controller: (1) PP-request to request the PP of the new protocol for the newly arrived packet when its PP does not exist in the switch, (2) Route-request to request the packet's intended route in the network if its PP exists in the switch. However, other types of exchanging messages between both sides are assumed to be as in a generic communication protocol. In general, the communication protocol is independent from the proposed framework and its implementation is similar to the existing messaging framework in OpenFlow with some new message types added to it that need to be designed accordingly.

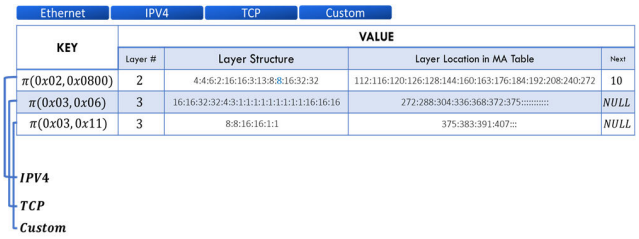

FIGURE 6. Protocol table example of the new designed switch.

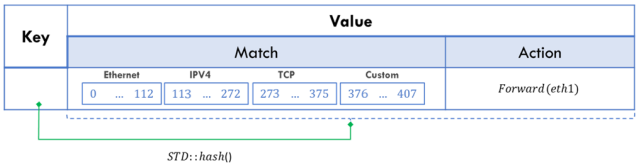

FIGURE 7. Flow table example of the new designed switch.

V. PROOF OF CONCEPT AND FUNCTIONAL EVALUATION

As a proof-of-concept and to demonstrate the proposed method's functionality, we designed a small network consisting of 7 switches, a controller, a PR server, and two hosts. The designed network is shown in Fig. 8. Initially the switches just have a layer 2 parser. However, it is supposed that the required protocols are registered on the PR and an unused PID is allocated to each one of them, and their PPs are saved on the PR server connected to the network.

We ran tests using virtual servers that execute the switch program code as switches and a separate server as the controller on which the controller program code is executed. In addition, the PR is deployed in another server that maintains the PPs.

In the test scenarios, H_1 wants to send packets with the desired protocol type to H_2, so after setting the IP header fields, it sets the Ethernet header fields with the related PID code in the "*ethernet type*" field. When the first packet arrives at S_1, its Ethernet layer is parsed, and the protocol type ID is fetched, but because the switch does not have the required PP, it sends a request to the controller to get the desired PP. The request is sent by a packet that consists of a copy of the arrived packet, the request type, and the ID of the requested protocol type (PID). The request type in this case is 0, which shows that the PP of the specified protocol type is requested, and the arriving packet needs to be routed.

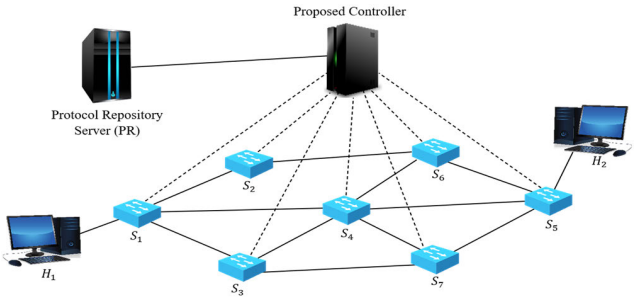

FIGURE 8. Designed network to evaluate the proposed framework.

If the request type is 1, it indicates that the arrived packet only needs to be routed.

In this case, because the request type is set to 0, assuming that the controller does not have the requested PP, it sends a request to the PR server and gets the desired PP. Then, by using the PP, the controller finds the route of the packet and sends the PP along with the related flow entry to the switches on the path to forward the packets of the flow. The switch stores the PP, updates the protocol identifier table by adding the new protocol ID, and sets the flow entry on the flow table. The selected path, specified by the routing algorithm (e.g., Dijkstra) that is run in the "Process Class" that exists in the parser module, is passed respectively through S_1, S_4, and S_5.

A. FIRST EXPERIMENT

In the first experiment, traditional IPv4 packets are sent to the network. As we know the "*ethernet type*" field of IPv4 packets is equal to 0×800. When the first packet arrives at the first switch on the network, the switch can only parse the Ethernet header and does not have the IPv4 header parser to process it, so after fetching the PID that exists on the "*ethernet type*" field that is, in this case, equal to 0×800 it sends the packet to the controller with a request for its L3 parser (that is, in this case, IPv4 parser). Then the controller fetches the relevant parser parameters (PP) from the PR and sends them to the switches on the network, enabling the switches to process this type of packets. As a result, the packets pass through the network as expected.

As we can see, we assumed that the IPv4 protocol is newly designed and did not exist in the switches before, then after a new packet of that type arrived in the network, ipv4 protocol is fetched from the PR and added to the switches in the network, so the arriving packets could be processed and pass the network.

If our network were a traditional OpenFlow-based network with OpenFlow switches, it would not be able to support new protocols without releasing a new version of OpenFlow and changing the switch hardware. Also, if our network were a P4-based network, although we can define the new protocol of the arrived packets in the switches, it would not be able to fetch and define the newly designed protocol in real-time and automatically, as the packets are passing. Instead, the protocol should be defined in the switches beforehand along with some effort from the network operator.

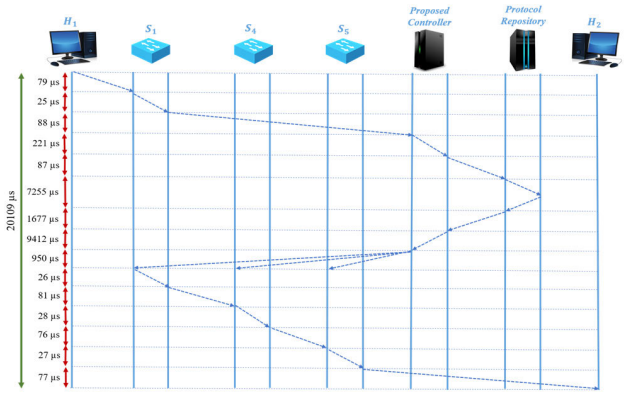

FIGURE 9. Time spent and process steps of the first arrived packet to the proposed framework.

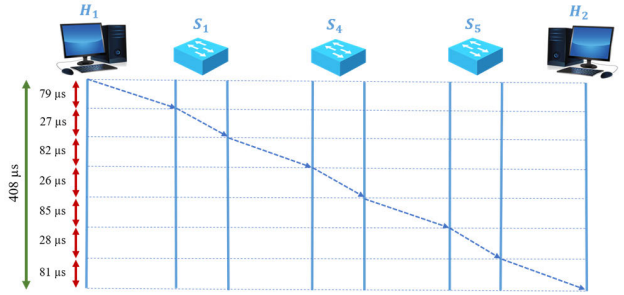

FIGURE 10. Time spent and process steps of the next arrived packets to the proposed framework.

To compare the functionality and performance of the proposed framework against OpenFlow-based SDN networks, we create the same network using OVS and floodlight controller, then some packets are sent from H_1 to H_2.

As we can observe from the log taken from the switches in the proposed method, shown in Fig. 9, the switches were able to process and forward the packet until it reached the destination. In addition to the processing steps, the figure shows the time spent to fetch the required PP and process the packet. The steps shown in this figure are only applied over the first packet of the flow whenever the required PP does not exist.

Fig. 10, shows the steps taken and the time spent to process and forward the subsequent packets of the new protocol after it is added to the switches.

The log taken from the switches in the OpenFlow-based network are shown in figures Fig. 11 and Fig. 12.

The time spent for the first packet of the flow to reach the destination in our proposed framework, as shown in Fig. 9 is 20.109 milliseconds (ms), while in the OpenFlow-based network it is 13.116 ms, as shown in Fig. 11. The additional delay in the proposed method is due to the process of retrieving the parser from the PR, which happens only once when the switch and controller do not have the parser. After that, as we can see from figures Fig. 10 and Fig. 12 the time spent for the subsequent packets in our proposed framework is less than that of the first packet and is very

close to the one in the OpenFlow-based network. So, there is only a small overhead in the case that the related protocol does not exist in the switch, so it should be fetched from the PR, and this case happens only after defining the new protocol, especially since the network is assumed to be fixed and not with dynamic behavior. Therefore, this overhead can be considered insignificant because we rarely have a newly designed protocol in fixed protocol networks.

Similarly, the main difference between our proposed framework and the P4-based network is in the protocol deployment stage. In our proposed method the protocol is deployed automatically, while in P4 the protocol should be deployed and added to the switches by the operator. After this stage, the packets of a newly deployed protocol type can be processed and forwarded in both networks in a similar manner.

So, in reality, supporting new protocols in our proposed method is faster than others, because the time required to deploy a new protocol in our framework is only a few milliseconds and it is done automatically (for the above example it is approx. 7 ms), so the operator involvement and effort required is none. While the effort required in a P4-based network depends on the number of the switches that need the new protocol to be added to them and their locations, which causes spending more time than our proposed method. Moreover, the operator involvement and effort needed to add a new protocol to the switches in the traditional OpenFlow-based network is not comparable, since, the OpenFlow should be changed and updated first then the switches should be replaced with ones that support the new OpenFlow version.

In addition, in the scenarios of frequent changes in the protocol, if we want to compare the deployment time order of these changes between our proposal and other works, we also can see that our time order is in milliseconds in the worst case, but for the other works, it takes seconds to compile, upload, and deploy a new protocol in the switch.

B. SECOND EXPERIMENT

In the second experiment, traditional MPLS packets are sent to the network. The "*ethernet type*" field of MPLS packets is equal to 0×8847. When the first packet arrives at the first switch on the network, the switch not knowing the MPLS header parser to process it, fetches the PID of the next layer that exists on the "*ethernet type*" field that is, in this case, equal to 0×8847 and sends the packet to the controller with a request for its parser (that is, in this case, MPLS parser). Then the controller fetches the relevant parser (PP) from the PR and sends it to the switches on the network, enabling them to process this type of packet.

This example shows the difference between our proposed framework and the traditional networks. When the MPLS protocol was proposed first, the existing routers could not process it. So, the manufacturers changed the hardware and the software of the routers adding this new feature. Also, in OpenFlow-based networks, especially in the early versions of OpenFlow, the switches did not support MPLS,

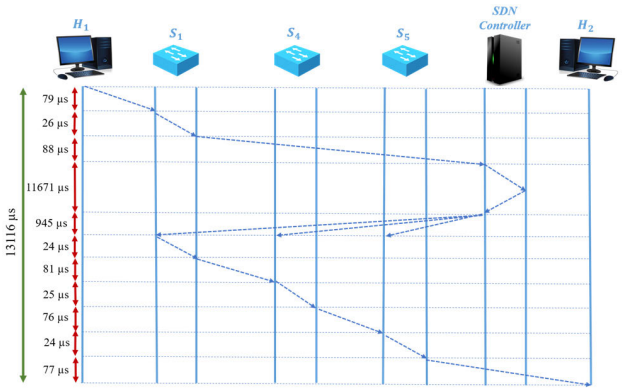

FIGURE 11. Time spent and process steps of the first arrived packet to the OpenFlow-based network.

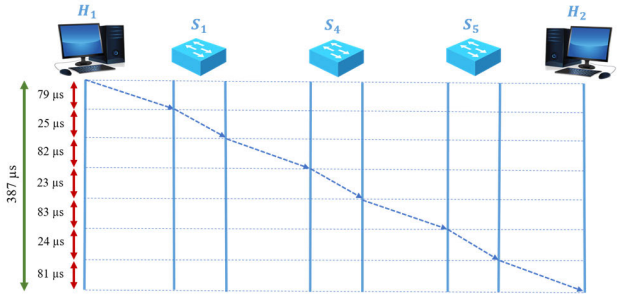

FIGURE 12. Time spent and process steps of the next arrived packets to the OpenFlow-based network.

after that, the OpenFlow version was updated and manufacturers changed the switch design to support the updated version of OpenFlow, then the new switches were able to process MPLS header. In P4, although the MPLS protocol can be easily defined without changing the switch, it should be programmed and added to the switches by the operator beforehand.

In our proposed method, there is no need to change the devices in the network and there is no need for the network operator involvement because new protocols can be deployed in the switches in real-time automatically as the packets are passing through the network.

C. THIRD EXPERIMENT

In the third experiment, a new protocol is designed and registered in the PR. The newly designed protocol for this evaluation is assumed to be a protocol that exists between layer 2 and layer 3 and consists of 3 fields X, Y, and Z. In this test, the packet's structure in the designed protocol is shown in Fig. 13. As can be seen in the figure, this packet has conventional Ethernet and IP layer headers while it also has a new layer header between them, which is designed based on the concept of the testing protocol used in this experiment. The fields considered in this test are the *hostname of the destination*, the *hostname of the source*, and the *type of the next layer* respectively.

In this test scenario, in H_1 after setting the IP header fields, it sets the value of the "*Destination Name*" field to 2 and the

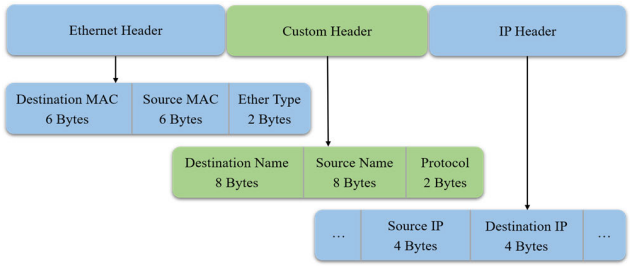

FIGURE 13. Packet structure of the test protocol.

value of the "*Source Name*" field to 1, then it sets the Ethernet header fields with the related PID code in the "*ethernet type*" field. Then, it starts sending packets with the desired protocol type to H_2. The network in this experiment acts similarly to the first and second experiments, also the steps and time spent for the packets to pass the network are similar to what is shown in figures Fig. 9 and Fig. 10. In addition, as the previous experiments and protocols, the new header fields can be involved in the decision-making process. However, as mentioned before, the routing process is handled in the controller only for the first packet of a new flow and if this packet involves a new protocol there will be an extra overhead due to the PP installment.

After the optimizations made on the proposed switch software, we were finally able to reach the speed of 13.128 Gbps for 64-byte packets, which is equivalent to 20 million packets per second (PPS), in the infrastructure where our existing network is 60 Gbps. In terms of the maximum rate that can be processed, we were able to fill the entire 60-Gbps link with 1500-byte packets. In this way, with better infrastructure, theoretically, 100-Gbps speeds can also be achieved [35], [36], [37], [38], [39].

In addition to the mentioned experiments as existing use cases, any desired protocol (new or traditional) can be defined in the proposed method without any administrative involvement, such as Google's QUIC, Tencent's HARP, and so on. The only involvement the human should do is define the parameters to process this type of packet as a class and register it on the PR.

VI. CONCLUSION

Network programmability has long been an attractive objective that is now partially met by SDN, mostly in the control plane. A flexibly programmable data plane is still an active research topic.

In this paper, a new NFV-based framework was proposed for SDN networks that makes the data plane autonomously programmable using software switches. Using the method proposed based on this framework, new and customized network protocols can be autonomously deployed in the network without the intervention of the operator and without the need to replace the network devices or change the switch hardware for new protocols.

In the context of the proposed framework a software switch design, and a controller definition were proposed, and a protocol repository was suggested that can be used to provide newly defined protocols with their actions based on the NFV concept. The desired protocol's actions are deployed in the controller similar to a VNF and then they are deployed in the switches by the controller. The same concept is extended to modify and update the match-action fields of the previously deployed protocols in the switches by the controller. Other than these capabilities the controller is similar to the traditional SDN controllers with the same functions.

Briefly, in our proposed framework, network switches do not need to be pre-programmed to forward packets, new protocols only need to be registered in the Protocol Repository in the PDL. Based on this, with the arrival of the first packet of any protocol type, if the protocol type is not recognized as a previously defined protocol, a copy will be sent to the controller requesting the PP of this type of packet. Then the controller fetches the PP from the PR and sends it to the switches. After that, the switches receive the PP from the controller and store it in their modules list. The protocol identifier table is also updated, adding a new protocol type along with the protocol ID code. Then with the arrival of the next packets of this type of protocol, the switches will be able to parse and process the fields of these packets.

A proof-of-concept implementation of the proposed framework showed that the proposed method works well as expected and can deploy an arbitrary protocol autonomously in the network controller and switches, and as a result, can process and forward packets of the new protocol throughout the network.

Based on the functional evaluation, we can see that the proposed method has achieved the desired advantages as expected. The evaluation shows that we can define new protocols, modify them, or add new features to them, and pass the packets of any protocol type (a new protocol or an existing protocol with new features) from the network without replacing the switches. And all of these are done without the network operator's involvement.

On the other hand, we can see that this proposed method does not have the limitations of other methods. As we previously mentioned, POF still depends on a set of instructions supported by the switch, and specific hardware is also needed to support these instructions, while our proposed method does not depend on switch instructions nor needs specific hardware, it depends on a software class that runs in the switch to parse the passing packets. However, although P4 can define new protocols without the previous limitations, it still requires the network operator's involvement to program these protocols in the switches and it should be done beforehand and prior to the arrival of the new protocol packets.

In terms of performance, deploying the new protocol in the switches on our proposed framework happens only for the first packet of the newly designed protocol that arrives at the switch. We showed that the delay in this case is in an acceptable range especially in comparison with P4-based networks that have more operator involvement and effort,

however, in either case, this delay is not something that affects the data plane performance.

So, our proposed method's contribution compared to solutions like P4 is mostly in the protocol deployment stage. However, P4 is proposed to work in traditional networks in addition to SDN, while our proposed framework is designed for SDN-based networks.

In SDN networks, running network functions on a packet can be performed in the controller, but sending the packets to the controller causes a significant delay in forwarding the packets through the network in addition to the delay that can be caused by processing them at the controller. Also, sending several packets to the controller to perform the required function on them leads to a bottleneck at the controller. While some other network functions have to be performed on the data plane as packets are passing through network devices (Like In-band Network Telemetry (INT)). Therefore, the NFV concept used in this framework can be extended to support any desired packet processing (network functions) in the data plane, as VNF, when the packets are passing through the switches, as was expected originally by the idea of Active Networks [47], [48], [49], given that the pipeline timing can be flexibly adjusted to accommodate the new processes. However, this feature requires switch architectures with enough processing power that support such flexibilities. This is a future work of this project.

The limitations of our proposed method are that the switches should have the layer 2 protocol defined in the switches (as a built-in protocol) and we should reuse a field from its header to have the ability to specify the next layer protocol to process the packets. While supporting new link layer protocols can be investigated as a future work. Finally, a common requirement to deploy all of the proposed solutions for data plane programmability, including our solution, is replacing the network switches to support the proposed solutions' capabilities.

The proposed method like any other method has to deal with security aspects. These issues are not addressed in this paper and can be investigated in future works.

REFERENCES

[1] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015, doi: 10.1109/JPROC.2014.2371999.

[2] T. Sloane. (May 2, 2013). *Software-Defined Networking: The New Norm for Networks*. Accessed: Oct. 3, 2024. [Online]. Available: https://opennetworking.org/sdn-resources/whitepapers/software-defined-networking-the-new-norm-for-networks

[3] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for SDN? Implementation challenges for software-defined networks," *IEEE Commun. Mag.*, vol. 51, no. 7, pp. 36–43, Jul. 2013, doi: 10.1109/MCOM.2013.6553676.

[4] K. Govindarajan, K. Chee Meng, and H. Ong, "A literature review on software-defined networking (SDN) research topics, challenges and solutions," in *Proc. 5th Int. Conf. Adv. Comput. (ICoAC)*, Chennai, India, Dec. 2013, pp. 293–299.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, doi: 10.1145/1355734.1355746.

[6] B. Yi, X. Wang, K. Li, S. K. Das, and M. Huang, "A comprehensive survey of network function virtualization," *Comput. Netw.*, vol. 133, pp. 212–262, Mar. 2018, doi: 10.1016/j.comnet.2018.01.021.

[7] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015, doi: 10.1109/ACCESS.2015.2499271.

[8] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane," in *Proc. 2nd ACM SIGCOMM Workshop Hot Topics Softw. Defined Netw.*, Aug. 2013, pp. 127–132.

[9] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "p4: Programming protocol-independent packet processors," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014, doi: 10.1145/2656877.2656890.

[10] S. Jouet and D. P. Pezaros, "BPFabric: Data plane programmability for software defined networks," in *Proc. ACM/IEEE Symp. Architectures for Netw. Commun. Syst. (ANCS)*, Beijing, China, May 2017, pp. 38–48.

[11] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA—An open platform for teaching how to build gigabit-rate network switches and routers," *IEEE Trans. Educ.*, vol. 51, no. 3, pp. 364–369, Aug. 2008, doi: 10.1109/TE.2008.919664.

[12] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "NetFPGA: Reusable router architecture for experimental research," in *Proc. ACM Workshop Program. Routers Extensible Services Tomorrow*, Aug. 2008, pp. 1–7.

[13] A. Bitar, M. S. Abdelfattah, and V. Betz, "Bringing programmability to the data plane: Packet processing with a NoC-enhanced FPGA," in *Proc. Int. Conf. Field Program. Technol. (FPT)*, Queenstown, New Zealand, Dec. 2015, pp. 24–31.

[14] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000, doi: 10.1145/354871.354874.

[15] M. Honda, F. Huici, G. Lettieri, and L. Rizzo, "mSwitch: A highly-scalable, modular software switch," in *Proc. 1st ACM SIGCOMM Symp. Softw. Defined Netw. Res.*, Jun. 2015, pp. 1–13.

[16] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, and P. Shelar, "The design and implementation of open vSwitch," in *Proc. 12th USENIX Conf. Netw. Syst. Design Implement.*, May 2015, pp. 117–130.

[17] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, "PISCES: A programmable, protocol-independent software switch," in *Proc. ACM SIGCOMM Conf.*, vol. 38, Aug. 2016, pp. 525–538.

[18] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy, "SoftNIC: A software NIC to augment hardware," Dept. EECS, Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2015-155, May 2015.

[19] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, "Dataplane specialization for high-performance OpenFlow software switching," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 539–552.

[20] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Conf. Operating Syst. Design Implement. (OSDI)*, Nov. 2016, pp. 203–216.

[21] A. Agrawal and C. Kim, "Intel Tofino2—A 12.9Tbps P4-programmable Ethernet switch," in *Proc. IEEE Hot Chips 32 Symp. (HCS)*, Palo Alto, CA, USA, Aug. 2020, pp. 1–32.

[22] Intel. (2024). *Intel Tofino: P4-Programmable Ethernet Switch ASIC That Delivers Better Performance At Lower Power*. Accessed: Oct. 3, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html

[23] Intel. (2024). *Intel Tofino 2: Second-Generation P4-Programmable Ethernet Switch ASIC That Continues To Deliver Programmability Without Compromise*. Accessed: Oct. 3, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series/tofino-2.html

[24] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian, "Fabric: A retrospective on evolving SDN," in *Proc. 1st Workshop Hot Topics Softw. Defined Netw.*, Aug. 2012, pp. 85–90.

[25] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, "Toward software-defined middlebox networking," in *Proc. 11th ACM Workshop Hot Topics Netw.*, Oct. 2012, pp. 7–12.

[26] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. V. Lakshman, "Application-aware data plane processing in SDN," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw.*, Aug. 2014, pp. 13–18.

[27] A. Nakao, "Deeply programmable network; emerging technologies for network virtualization and software defined network (SDN)," in *Proc. ITU Kaleidoscope, Building Sustain. Communities*, Kyoto, Japan, Apr. 2013, p. 1.
[28] K. Yamada, Y. Kanada, K. Amemiya, A. Nakao, and Y. Saida, "VNode infrastucture enhancement—Deeply programmable network virtualization," in *Proc. 21st Asia–Pacific Conf. Commun. (APCC)*, Kyoto, Japan, Oct. 2015, pp. 244–249.
[29] N. McKeown, and J. Rexford. (May 18, 2016). *Clarifying the Differences Between P4 and OpenFlow*. Open Networking Foundation. Accessed: Oct. 3, 2024. [Online]. Available: https://opennetworking.org/news-and-events/blog/clarifying-the-differences-between-p4-and-openflow
[30] E. Kaljic, A. Maric, P. Njemcevic, and M. Hadzialic, "A survey on data plane flexibility and programmability in software-defined networking," *IEEE Access*, vol. 7, pp. 47804–47840, 2019, doi: 10.1109/ACCESS.2019.2910140.
[31] R. Bifulco and G. Rétvári, "A survey on the programmable data plane: Abstractions, architectures, and open problems," in *Proc. IEEE 19th Int. Conf. High Perform. Switching Routing (HPSR)*, Bucharest, Romania, Jun. 2018, pp. 1–7.
[32] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *J. Netw. Comput. Appl.*, vol. 212, Mar. 2023, Art. no. 103561, doi: 10.1016/j.jnca.2022.103561.
[33] B. Amutha, R. Jeya, and H. Gondaliya, "Data plane programmability: Enabling the vision of programmable networks," *Int. J. Adv. Sci. Technol.*, vol. 29, no. 8, pp. 240–248, 2020.
[34] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87094–87155, 2021, doi: 10.1109/ACCESS.2021.3086704.
[35] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the performance of state-of-the-art software switches for NFV," in *Proc. 15th Int. Conf. Emerg. Netw. Exp. Technol.*, Dec. 2019, pp. 68–81.
[36] Q.-H. Nguyen and Y. Kim, "Performance of OVS-DPDK in container networks," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Jeju, South Korea, Oct. 2020, pp. 437–440.
[37] Y. Zhao, L. Iannone, and M. Riguidel, "Software switch performance factors in network virtualization environment," in *Proc. IEEE 22nd Int. Conf. Netw. Protocols*, Raleigh, NC, USA, Oct. 2014, pp. 468–470.
[38] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and latency of virtual switching with open vSwitch: A quantitative analysis," *J. Netw. Syst. Manage.*, vol. 26, no. 2, pp. 314–338, Apr. 2018, doi: 10.1007/s10922-017-9417-0.
[39] J. Xiao. (May 2017). *New Approach To OVS Datapath Performance, Open VSwitch Open Source Day At OpenStack Summit*. Accessed: Oct. 3, 2024. [Online]. Available: https://www.openvswitch.org/support/boston2017/1530-jun-xiao.pdf
[40] S. Li, D. Hu, W. Fang, S. Ma, C. Chen, H. Huang, and Z. Zhu, "Protocol oblivious forwarding (POF): Software-defined networking with enhanced programmability," *IEEE Netw.*, vol. 31, no. 2, pp. 58–66, Mar. 2017, doi: 10.1109/MNET.2017.1600030NM.
[41] DPDK. (Sep. 18, 2024). *DPDK: Data Plane Development Kit*. Accessed: Oct. 3, 2024. [Online]. Available: http://dpdk.org
[42] X. Wang, Y. Tian, M. Zhao, M. Li, L. Mei, and X. Zhang, "PNPL: Simplifying programming for protocol-oblivious SDN networks," *Comput. Netw.*, vol. 147, pp. 64–80, Dec. 2018, doi: 10.1016/j.comnet.2018.09.018.
[43] J. F. Kurose and K. W. Ross, "The link layer and LANs," in *Computer Networking: A Top-Down Approach*, vol. 4, 8th ed., London, U.K.: Pearson, 2020, ch. 6, sec. 4, p. 485.
[44] B. A. Forouzan, "Ethernet protocol," in *Data Communications and Networking*, vol. 1, 5th ed., New York, NY, USA: McGraw-Hill, 2012, ch. 13, sec. 1, p. 362.
[45] S. A. M. Noohi. *IPv4 Protocol As a PP in the Switch Used for Proposed Architecture*. Accessed: Oct. 3, 2024. [Online]. Available: https://github.com/amirmnoohi/Packenger/tree/master/Packenger/Headers/Ethernet_Protocols/IPv4.py
[46] S. A. M. Noohi. *NSwitch: Fast and Flexible Software Switch*. Accessed: Oct. 3, 2024. [Online]. Available: https://github.com/amirmnoohi/NSwitch
[47] J. T. Moore, M. Hicks, and S. Nettles, "Practical programmable packets," in *Proc. 20th Conf. Comput. Commun. (INFOCOM)*, vol. 1, Anchorage, AK, USA, Apr. 2001, pp. 41–50.
[48] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 5, pp. 81–94, Oct. 2007, doi: 10.1145/1290168.1290180.
[49] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden, "A survey of active network research," *IEEE Commun. Mag.*, vol. 35, no. 1, pp. 80–86, Jan. 1997, doi: 10.1109/35.568214.

ABBAS KHATER received the B.Sc. and M.Sc. degrees in computer engineering from the Department of Electrical and Computer Engineering, Isfahan University of Technology (IUT), Iran, in 2012 and 2015, respectively, where he is currently pursuing the Ph.D. degree. His Ph.D. research investigates data plane programmability in SDN networks. His research interests include software-defined networking, data plane programmability, computer networks, and the IoT.

SEYED AMIRMASOUD NOOHI received the B.Sc. and M.Sc. degrees in computer science from Isfahan University of Technology, Iran, where he developed a strong foundation in Operating Systems, Computer Networks, and Virtualization. His research interests are centered on distributed systems, cloud computing, and performance optimization techniques in large-scale systems. He has actively contributed to several academic research projects, with a focus on designing scalable, efficient systems architectures and improving the reliability of virtualized environments. His work often involves low-level system programming, kernel development, and network simulations.

MASSOUD REZA HASHEMI received the Ph.D. degree in electrical and computer engineering from the University of Toronto, Canada, in 1998. From 1998 to 1999, he was a Postdoctoral Fellow with the University of Toronto. He was a Founding Member and the Lead Systems Architect with AcceLight Networks, in 1999, where he developed some of the key system elements of a multi-terabit multiservice core switch. Since 2003, he has been with Isfahan University of Technology, where he is currently a Full Professor. He was the Head of the IT Center, Isfahan University of Technology, from 2005 to 2013. From 2016 to 2017, he was a Visiting Scholar with the University of Toronto on sabbatical leave. His current research interests include digital twin, smart edge, and intelligent edge computing.

ZEINAB ZALI received the B.Sc., M.Sc., and Ph.D. degrees in computer engineering from Isfahan University of Technology (IUT), Isfahan, Iran, in 2006, 2009, and 2019, respectively. In 2014, she was a Visiting Student with the Telematics Laboratory, Politecnico di Bari, Italy. She is currently an Assistant Professor with the Department of Electrical and Computer Engineering, IUT. Her research interests include computer networks, cloud and edge computing, data analysis, and intelligent services in cellular networks.