Instruction based Packet Processing Scheme for Programmable SDN Data Plane

Ki-Wook Kim, Pill-Won Park, Sung-Gi Min

Department of Computer and Radio Communication Engineering, Korea University, Seoul, South Korea, {wowook, pillwon79, sgmin}@korea.ac.kr

Abstract

Software Defined Networking (SDN) decouples a network equipment’s control plane from its data plane, the underlying packet forwarding system. It makes a network to be easily programmable, and improves network flexibility especially on its control plane. However, the SDN technology so far cannot provide data plane programmability, the programmability for in-network processing (such as Network Address Translation or Deep Packet Inspection) and new non-IP protocols, in run-time. To provide the data plane programmability in run-time, two parts are needed: a method to describe the new data plane operation and a flexible network data plane to accept the operation in run-time. We propose a new packet processing scheme to support the data plane programmability in SDN data plane. The proposed scheme is based on well-known BSD packet filtering method. The implemented prototype shows that a controller application can write and insert a new arbitrary packet processing rule to the prototype engine in run-time.

Keywords: SDN, Programmable Data Plane, Packet Processing, in-network data processing, BSD packet filter

1. Introduction

Network Virtualization has made a large contribution toward an expansion of cloud service and 5G market. It is used to address frequent network configuration request from multiple tenants and leads to improvement in network control technologies such as Software Defined Networking (SDN). SDN separates the control and data plane of networking architecture. This separation introduces the standard form of data plane architecture and flexible control plane.

With the flexible control plane, control applications running on the controller can dynamically deploy new network architectures to a running network slice easily. However, when the new network architecture requires new operations on the data plane, it cause the extension of the control interface specification (e.g. extension of the OpenFlow protocol) and the data plane to support the new operations. As a matter of course, a flexible data plane which can adopt the new operations in run-time is essential for the dynamic deployment of the new network architecture.

Unfortunately, the most of existing SDN data planes are not flexible enough for the dynamic installation of new network operations. They are modeled after looking the existing network equipments. Legacy network equipments are tightly coupled with a set of protocols they handle and they are based on the table lookup mechanism using TCAM (Ternary Content Addressable Memory). The table lookup mechanism has a fixed set of selectable fields of packet headers, and use them for flow classification. The packet handling action for an incoming packet depends on its classified flow. There is no way to extend these selectable classification fields or handling actions at run-time to process new types of packets such as new non-IP protocols or in-network processing functions (e.g.
Network Address Translation or Deep Packet Inspection), so they are the functional boundary of a running data plane. To keep pace with the flexibility of control plane, the data plane should have a way to extend the functional boundary dynamically.

We introduce the Instruction-based Packet Processing (IPP) scheme as a SDN data plane to enhance the programmability of the data plane. The scheme includes the IPP language and the IPP Virtual Machine (IPP VM). The language is used to write an IPP code, a procedure of network operation for a data plane. The VM is a packet processing architecture for the data plane, which supports dynamic loading of the programmed network operations at run-time. The proposed scheme is based on Berkeley Packet Filter (BPF), a well-known software packet filtering method. It inherits flexibility and stability of the BPF architecture, while extends the BPF language to adopt it to the SDN switch. With the new processing scheme, a SDN controller can insert a new arbitrary packet processing rule to a data plane as a plug-in.

2. Related Work

There are several proposals introducing the programmability into the SDN data plane, such as [1-3, 5, 19, 20]. In spite of their programmable features, their language is targeted to table based packet processing model (like fig. 1, the reference data plane model defined at the P4). It abstracts the parse-match-action pipeline operations in a dedicated hardware. It has ingress and egress tables, and all of the network programs are translated to the packet processing rules and placed at one of these tables. It aims to allow programmers to describe match-action tables that are dynamically populated by well-typed rules.

![Figure 1. Abstracted Forwarding Model in P4.](image)

In 1993, Steven McCanne and Van Jacobson introduced a novel way of filtering packets in the kernel. It is called BPF [13]. The BPF has been used widely at the network applications such as libpcap and tcpdump. The BPF defines the virtual filter machine and the language for the machine. The language contains a few instructions for fetching data from the packet, performing arithmetic operations, and comparing data. A packet filter is defined by a code using the language. The BPF virtual machine executes the packet filter code to decide whether an incoming packet is acceptable or not. In [12], the BPF is used simply to match incoming packets. Several mechanisms [6-8] have been proposed to improve speed and expressivity of the BPF, but they do not extend the functionality of the BPF. Even though the BPF is excellent for packet matching, functional extensions are needed to make it suitable for versatile packet processing such as modifying or generating packets.
3. The Instruction-based Packet Processing (IPP) Scheme

The IPP scheme aims to introduce a programmable SDN data plane. It consists of the IPP language and the IPP Virtual Machine (IPP VM). The IPP language is used to describe the packet processing procedures for network operations. A procedure written by the IPP language is called as an IPP code. The IPP VM runs the IPP codes, and it supports dynamic installation of IPP code at runtime.

3.1. IPP Virtual Machine

To avoid collisions between our player robots and the obstacles like the opposite player robots, we need to determine the avoidance direction and the avoidance motion of our player robots. Therefore, we used Complex Measure (CM) and the Fuzzy Logic based Avoidance Motion Selector (FL-AMS) [6] to solve the given problem.

![The IPP Virtual Machine Architecture.](image)

The IPP VM is composed of a scheduler, an execution engine, a loader, packet buffers and a runtime storage (Fig 2). The straight-line arrows in the figure represents data flows, while the dotted arrows are for the control flows.

- **Scheduler and Execution Engine**: The scheduler is a component of the VM which schedules the tasks of execution engine. When a new packet arrives at input packet buffer, the scheduler queues the installed IPP instances in order by their priority to the execution engine. The execution engine runs the scheduled tasks.

- **IPP Loader**: The IPP loader installs the IPP code into the VM on behalf of an IPP application. When an IPP code is received by the IPP loader, it compiles the IPP code into the executable form, which can be executed by the execution engine. The executable form of the IPP code is called IPP instance. The IPP instance includes the runtime data section for a non-volatile data. They are used to remember the state of the IPP instance. Then the IPP loader stores the IPP instance at the runtime storage, and it will be executed when a packet arrives to the input packet buffer.
Input and Output Packet Buffers: The Input and Output Buffer consist of their own packet queues and schedulers. When an incoming packet is arrived at the IPP VM, the Input Buffer scheduler receives it. It queues the packet into its packet queue and generates a packet-in event. The Output Buffer stores the outgoing packets. Packets in the Output Buffer are transmitted outside of the VM by the Output Buffer scheduler.

Runtime Storage: The Runtime Storage maintains the installed IPP instances and their runtime data. The runtime storage is a non-volatile storage which has independent lifecycle to the IPP code, which means that an IPP instance can aggregate data over multiple packets. It is used to remember the data processing state for future execution.

3.2. IPP Language

The IPP language is a low-level language for the IPP VM. It consists of a series of directives, instruction statements and comments. Directives inform the VM about how to install the IPP code. Instruction statements are symbolic machine codes of the IPP VM, and they are translated to executable codes in the VM.

The IPP language is derived from the BPF, especially LSF (Linux Socket Filter [14]). It inherits many language features of the BPF, such as instruction statement syntax, instruction set, scratch memory (addressable registers with a limited size) and BPF extensions (platform-dependent variables provided by the Linux kernel). However, the BPF (including LSF) is specialized to packet matching. It is not suitable to describe in-network operations such as packet forwarding, packet modification and data aggregation. The IPP language extends the LSF instruction set to include these in-network operations.

The table 1 shows the directives and the extended instructions. The loader decodes the directive statements in an IPP code ahead of translating its instructions. Directives describe the configurations needed for constructing an IPP instance from the code. The first table shows several basic directives. The setid directive explicitly sets the identifier value of the instance. The setpriority directive sets a priority value for the instance, and it is used to schedule the execution order of IPP instances for the given event. The addvar and setarray directives are used to declare non-volatile data variables. In the BPF, a non-volatile memory exists, called the scratch memory. The scratch memory is pre-defined and has fixed size, and it is not suitable to code various operations. The addvar and setarray are used to supplement the scratch memory, by making each instance declares the required memory in its own code.

The table 2 shows the newly defined instructions. The instructions are listed by their names (mnemonics) and their operands. The out instruction enqueues a packet to the output buffer of a designated port. The out instruction simply enqueues a packet to the output buffer of a designated port. The fwd instruction associates a priority to an outgoing packet to make sure that the forwarding is occurred only once for a packet. If several outgoing packets with different priorities exists for an incoming packet, the packet with the highest priority will be forwarded.

The ign and norm instructions are used to process a packet with typical drop or forward actions. If the ign instruction is executed while running an instance, the packet forwarding instructions in other instances, which have lower priority value than the instance, are ignored for the same incoming packet. This instruction is mainly used to drop a packet. If the norm instruction is executed in any instance for an incoming packet, the packet will be processed by the vendor dependent packet processing method (such as IP routing or VPN tunneling) after the IPP processing finished.

To modify fields in an incoming packet, wrt, wrth are introduced. Before their operations, the original packet is duplicated for safe concurrent packet processing. If there are several IPP codes for an incoming packet, several modified packets may be created by them. The npkt instruction allocates memory for a new packet. Note that there are much more instructions inherited from the BPF (refer to [14] for them). They are fetching data from the packet, performing simple arithmetic operations, and comparing data.
Table 1. Directives in the IPP Language.

<table>
<thead>
<tr>
<th>Directive statement</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>%setid &lt;id&gt;</td>
<td>Set the instance ID explicitly</td>
</tr>
<tr>
<td>%setpriority &lt;priority&gt;</td>
<td>Set priority value of the instance</td>
</tr>
<tr>
<td>%addvar &lt;name&gt; &lt;init value&gt;</td>
<td>Declare an integer variable</td>
</tr>
<tr>
<td>%setarray &lt;arr name&gt; &lt;size&gt;</td>
<td>Declare an array</td>
</tr>
</tbody>
</table>

Table 2. New Instructions in the IPP Language.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operands</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out</td>
<td>port</td>
<td>Packet out to port</td>
</tr>
<tr>
<td>fwd</td>
<td>port</td>
<td>Packet forward to port</td>
</tr>
<tr>
<td>norm</td>
<td></td>
<td>Vender-dependent packet processing</td>
</tr>
<tr>
<td>ign</td>
<td></td>
<td>Ignore the lower priority instances</td>
</tr>
<tr>
<td>wrt</td>
<td>offset</td>
<td>Write word A to packet offset</td>
</tr>
<tr>
<td>wrth</td>
<td>offset</td>
<td>Write half-word A to packet offset</td>
</tr>
<tr>
<td>npkt</td>
<td>size</td>
<td>Allocate a new packet buffer</td>
</tr>
</tbody>
</table>

4. Implementation and an Example IPP Code

We implemented the IPP VM as an alternate packet processing engine of Open vSwitch (OVS, [17]) for easy prototyping and testing. The implemented IPP engine is available on our project website [18]. It alternates the packet classifying and forwarding codes in the OVS datapath module to the IPP VM, and a device is added to make loader interface of the IPP VM.

A simple example IPP code, in figure 3, is written to test the implemented engine. It considers a case of packet load-balancing between two available paths for a packet destination. Path capacity of the path 1 is about double of the one of path 2. To get the maximum amount of flow bandwidth, the IPP code describing a network operation that classifies incoming packets into two groups and forward them to each ports, is injected as a sender component of the channel bonding pair.

At first, the code indicates the loader to setup a non-volatile variable for packet counter and initialize it when installing this code. Then the following codes verifies the incoming packet by its ethertype and addresses. It increases the packet counter value, and it decides a path for the packet using the counter value. Finally it forward the packet to the selected path.

5. Conclusion and Future Work

This paper proposes the design and implementation of the IPP scheme, as a new SDN data plane scheme. It is based on the well-known BPF architecture to inherit its programmability and stability, proven over many years. It provides run-time programmability to the administrator (such as SDN controller) through the IPP language and the IPP VM.

The IPP language is used to program an arbitrary network operation procedure as an IPP code. The IPP language is similar to the language of the BPF, so it is very familiar to most of network engineers. The IPP VM is a packet processing machine which runs the IPP code to process packets, and it supports run-time injection of new IPP code. With these two features, an administrator can write and install his own network operation code to a running SDN data plane.

Through the prototype implementation, we assured ourselves that new network operations can be programmed as IPP codes, and it can be injected to a running IPP engine in run-time. We continue to investigate the benefits of our approach in various cases on the one hand, and we are making a much more flexible data plane by adding event-driven features to the IPP scheme.
; set counter
%addvar count 0

; check EtherType=IPv4.
ldh [12]
jne #0x800, drop

; check SrcIP=0x00000001.
ldh [26]
jne #0x0000, drop
ldh [28]
jne #0x0001, drop

; check DstIP=0x00000002.
ldh [30]
jne #0x0000, drop
ldh [32]
jne #0x0002, drop

; increases packet counter value
ld #count
add #1
st #count

; decide path and forward the packet.
; if (counter%3) = 0 or 2) goes to path 1,
; and (counter%3) = 1) goes to path 2.
mod #3
jneq #1, out1
jmp out2
drop:
ign
ret #0
out1:
fwd #1
ret #0
out2:
fwd #2
ret #0

Figure 3. An Example IPP Code.

6. References


