A Packet I/O Architecture for Shell Script-based Packet Processing

Yohei Kuga[1], Takeshi Matsuya[1], Hiroaki Hazeyama[2], Kenjiro Cho[3], Rodney Van Meter[4], Osamu Nakamura[4]

[1]The Graduate School of Media and Governance, Keio University, 5322 Endo Fujisawa, 252-0882, Kanagawa, Japan
[2]The Graduate School of Information Science, Nara Institute of Science and Technology, Takayama 8916-5, Nara, Japan
[3]IIJ Research Lab, Takebashi Yasuda Bldg 3F. 3-13 Kanda Nishikicho, Chiyoda-ku, Tokyo 1010054, Japan
[4]Faculty of Environment and Information Studies, Keio University, 5322 Endo Fujisawa, 252-0882, Kanagawa, Japan

Abstract: We propose a new scripting model for rapid and easier development of packet processing using shell scripts. In this paper we present EtherPIPE, a character network I/O device, that allows the programmer to access network traffic data as a file through UNIX commands. By setting a UNIX pipe "|" from or to EtherPIPE's output or input with UNIX commands, packets can be easily processed, executing functions such as packet filtering, packet capturing, generating arbitrary packets, and rewriting header information. In order to prove the utilities of our model, we have developed FPGA-based EtherPIPE adapter using a commodity FPGA card and a character device driver featuring new offloading functions. With our prototype implementation, packet scripting works at 1Gbps wire-speed, receiving packets with precise hardware timestamps. This paper argues for use cases of the EtherPIPE, and discusses enhanced formats of character devices for easier network scripting.)

Key words: network I/O; Ethernet; Shell script; Software-Defined Networking; device driver; network adapter

I. INTRODUCTION

In order to realize Software-Defined Networking (SDN), many APIs and libraries have been proposed, such as OpenFlow and its controller [1-3], and API for Commercial Network Device [4-5]. Also, some libraries have been developed that provide high performance networking by connecting directly with the network device [6-7]. If we could use the UNIX shell commands for network processing, the scripting environment doesn't depend the software libraries, and we think the script environment becomes new lightweight network processing tools.

However, current network devices on a UNIX-like OS are implemented as special device with a specific API, and can not be used in an shell environment. One of the reasons why network I/O is usually implemented as a network device rather than a character device is the need to access different layers in a packet such as the Ethernet frame, the IP packet and the UDP/TCP datagram.

The BSD socket is one abstraction allowing network devices to access raw packets, and behaves as glue or a buffer to read/write a payload as characters. The BSD socket provides an interface to control packets with flexibility and good performance, however, we have to obey the programming paradigm of the BSD socket that forces us to write a long passage of code for accessing raw packets. Of course, we can inject packet streams more easily by using a virtual network device [8] or one of several libraries to access raw sockets [9]. These APIs provide easy access to raw socket but still require programming specific to networking. Following the characteristics and friendliness of these APIs, we would like to provide a simple network I/O for network processing on an OS in the same manner of device files for storage devices.

We propose the EtherPIPE network scripting framework in this paper. EtherPIPE provides a character device interface for a network device. A network device on EtherPIPE is abstracted as a device file on an OS, and packets on the network device are transformed to files or streams on the device file. EtherPIPE also serves as truly simple input/output functions for scripting packets like file processing on a UNIX command line. In EtherPIPE's network scripting, various packet processing can be achieved by I/O redirection through standard input (<), standard output (>) and pipe (|). We believe that the EtherPIPE network scripting framework brings a more flexible/lightweight programming paradigm that allows us to develop a packet processing application for a SDN.

Many features of the network have been fixed in hardware until the advent of SDN. With the advent of SDN, we can now develop new network functions quickly and flexibly using a commodity PC and its software.

To provide further flexibility to software-based development in networking, we developed EtherPIPE as an Ethernet device driver for a commodity FPGA network card on Linux. Our software and FPGA circuit are available as open source (https://github.com/sora/ethpipe). Combining EtherPIPE with hardware offloading functions of the FPGA or other network processors, more powerful network scripting or network processing can be achieved.

The rest of this paper is composed of the following sections; Section 2 discusses the primitive functions on network processing and defines primitives that must be supported in the EtherPIPE. Section 3 shows our concept of the network scripting that we try to achieve through the EtherPIPE. In Section 4, we explain device formats and interfaces of the EtherPIPE. Section 5 shows our implementation. We present examples of applications by EtherPIPE network scripting in Section 6. Section 7 dicusses the potential of EtherPIPE interface and performance of our implementation, and discuss extensions and limitations of EtherPIPE in Section 8. After referring to related work in Section 9, we conclude this paper in Section 10.

Our EtherPIPE allows shell scripting to deal with network devices and network I/O in the same manner as file devices and file I/O. It enables the programmer to access network traffic data as a file through UNIX commands.

II. Primitive Functions for Packet Processing

As a network scripting framework, primitive network processing functions should be provided by EtherPIPE as commands on an OS. Before designing the EtherPIPE network scripting framework, we explore the primitive functions needed to program network applications as shell scripts, and try to define a primitive function set.

We focus on network applications on the data-link layer as a first step. Note that we refer to Ethernet frames, IP packets and/or TCP/UDP datagrams, all as "packets" in the following sentences. We use the terms "Ethernet frames", "IP packets" and "TCP/UDP datagrams" when we would like to distinguish the data format on each layer.

Typical network applications on the data-link layer are as follows, network test and diagnosis tools such as packet generators and packet capture, and network elements such as Ethernet switch and tunnel gateways. These applications can be composed of five primitive functions 1) packet generation (packet sending), 2) packet capturing (packet receiving), 3) forwarding, 4) packet filtering, and 5) header modification.

Packet Sending and Receiving

All network equipments minimally require the functions of packet sending and/or packet receiving. TCP connections normally employ both packet sending and packet receiving.

Table I *Translate functions from packet processing to Ethernet character device*

Packet processing	Ethernet character device
receive packets	read from input device
send packets	write to output device
forward	copy from inputs to outputs
filter	search data pattern
modify headers	translate·characters

An application uses read(2) and write(2) on a socket file descriptor for a TCP connection.

Packet capture tools such as tcpdump [10] or WireShark [11] require only the packet receiving function. On the other hand, packet generator applications, such as pktgen [12] or scapy [13], are mainly composed of the packet sending function.

OpenFlow defines *Packet-in* and *Packet-out* functions [14]. Various OpenFlow libraries provide *Packet-in* and *Packet-out* APIs. Using these APIs, an open flow controller can send and receive arbitrary packets from OpenFlow switches.

Filtering

Filtering function is required by a firewall, port mirroring or network injector. Berkeley Packet Filter (BPF) [15], OpenBSD Packet Filter (PF) [16] or IPFW [17] are supported in various BSD Operating Systems.

Forwarding

Repeaters, switches and routers require a forwarding function to interconnect an input port and an output port. Several OSes support the forwarding function in the kernel. Recently, Linux brctl(8) and Open vSwitch [18] provide more flexible forwarding control.

Header Modification

Header modification is a key function to achieve forwarding, routing or encapsulation.

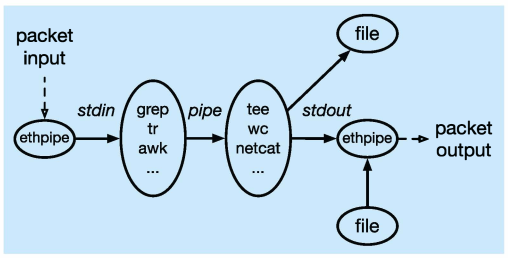

Fig.1 *Network scripting*

Filtering tools on various OSes or *Flow-Mod* action of Open vSwitch enable users to modify protocol headers. Usually, a header modification function is hidden in the kernel space. The RUMP (Runnable Userspace Meta Program) kernel of NetBSD [19] provides a header modification environment in the user space.

III. ETHERPIPE NETWORK SCRIPTING

We define "network scripting" to be processing packets in the UNIX shell interface or shell programming. In this section, we explain the design of the EtherPIPE network scripting framework to achieve the basic functions for network processing mentioned in Section 2 on the UNIX shell interface.

Table 1 shows the correspondence between basic functions and commands on the UNIX shell interface. In the UNIX shell programming framework, an application can be written by a chain of device files, files, and commands, concatenated by redirection expressions, using stdin (<), stdout (>) and pipe (|).

To handle packets in the UNIX shell programming manner, input and output for packets must be expressed in character devices that can be connected by stdin and stdout. Character device type network I/O uses the read(2) system call to a device file for packet receiving, and the write(2) system call to send packets to the device file.

The forwarding function can be simply achieved by copying data from the output of a device file to the input of the another device file. Also, redirection and concatenating commands will provide forwarding packets to multiple ports.

Each packet is a simple string, therefore, a combination of grep(1) and tr(1) will give a filtering function and a header modification function. Using regular expressions in grep(1) or sed(1), we will describe a complex search pattern in a line shell script.

Figure 1 shows an overview of network scripting. Concatenating character device type network I/O and shell commands, we can process packets as files in a UNIX shell interface. Of course, the network scripting inherits the UNIX shell programming paradigm, so a custom network scripting command can be reused in another network script. Through our network scripting, interactive processing against network streams can be realized.

IV. Design of Device Formats

The EtherPIPE device tree and its name space are shown in Figure 2. For simplicity, we consider the case where only one multi-port network card is inserted into a computer. We locate a network device in */dev* as is the case with other devices.

Therefore, EtherPIPE provides abstracted character device files under */dev/ethpipe/*. Each physical port on a network card is labeled as an independent device such as */dev/ethpipe/0* or */dev/ethpipe/1*.

EtherPIPE creates two device interfaces at each physical port; one is the **shell interface** and the other is the **raw interface**. The shell interface is designed to access packets by using ASCII for network processing on a shell. The device name is described by only port number in */dev/ethpipe/*.

The raw interface enables to access packets in a binary format for high-bandwidth network processing. The device name on the raw interface is labeled with '**r**' + port number. For instance, physical port '**0**' and '**1**' can be described as in Figure 2.

4.1 Shell interface

The shell interface is used for network scripting in the shell. Figure 3 presents the format of the shell interface. Packets are presented in ASCII, one packet per line and Ethernet header fields and payload in hexadecimal notation are separated with space characters by the kernel driver. Using the shell interface, we can parse packets by traditional command-line tools.

This shell interface on EtherPIPE is very simple, however, two alternatives for the ASCII expression are considered. One is expressing MAC addresses and IP Addresses in ASCII, the other is expressing all protocol headers in ASCII. Of course, adopting these expressions on EtherPIPE will give more control to network scripting, these expressions sacrifice processing time, data size and overhead on kernel drivers. Considering these trade-offs on ASCII expression, we take a simple ASCII expression described in Figure 3.

4.2 Raw interface

The raw interface is used for network processing in high-bandwidth network connections. Figure 4 shows the format of the raw interface. This interface has metadata that indicates a hardware timestamp, a frame length, a five-tuple hash and Ethernet frame data. All EtherPIPE metadata are computed by hardware, and can be used by network processing software in the user space.

The hardware timestamp is described in a 64-bit counter value with 8 nanoseconds resolution taken when the head of a packet arrives at the network hardware. Wireshark [11]

```
/dev/
  |-- ethpipe/
        |-- 0    # Shell IF port 0
        |-- 1    # Shell IF port 1
        |-- r0   # Raw IF port 0
        |-- r1   # Raw IF port 1
```

Fig.2 *EtherPIPE device name*

```
[DST_MAC] [SRC_MAC] [Eth_Type] XX XX XX XX ...
[DST_MAC] [SRC_MAC] [Eth_Type] XX XX XX XX ...
[DST_MAC] [SRC_MAC] [Eth_Type] XX XX XX XX ...
```

Fig.3 *Shell interface format*

```
+----------------------------+
|     Magic code (2 Byte)    |
+----------------------------+
|    Frame length (2 Byte)   |
+----------------------------+
|                            |
|    Hardware timestamp      |
|         (8 Byte)           |
|                            |
+----------------------------+
|      Five-tuple hash       |
|         (4 Byte)           |
+----------------------------+
|    Ethernet frame data     |
|           ...              |
+----------------------------+
```

Fig.4 *Raw interface format*

and its capture format PcapNG [20] support a nanosecond timestamp with NIC hardware counters. We use the same 64-bit timestamp by converting the EtherPIPE raw format to the PcapNG format.

The five-tuple hash is a hash value of <*source IP address, destination IP address, protocol number, source port number, and destination port number*> for the high throughput packet processing. Network processing often uses five-tuple hash values for identifying unique IP flows on routers, firewalls and load-balancers.

V. Implementation

Figure 5 shows an overview of our implementation design. We developed an implementation of EtherPIPE as a character device driver for a commodity FPGA network card on Linux, and implemented almost all of the EtherPIPE primitive functions.

Our FPGA implementation supports basic transmit/receive function for 1000BASE-T and hardware offloading functions such as hardware timestamp and five-tuple hash of the EtherPIPE Raw interface. Our design of the FPGA logic supports ring buffers for transmitting and receiving packets, and the bus master mode to transfer data between the NIC and the main memory.

We developed the FPGA logic using the LatticeECP3 versa development kit [21] by

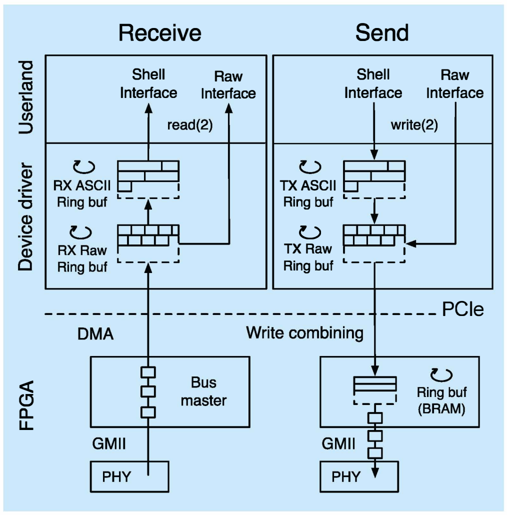

Fig.5 *Implementation of EtherPIPE adapter*

Lattice Semiconductor Inc. The Lattice ECP3 versa kit has two 1000BASE-T interfaces and one PCI Express interface, therefore, it can be used to test the forwarding case by EtherPIPE network scripting.

5.1 Device driver

The EtherPIPE device driver has two 1MB ring buffers for Raw and Shell interface used for the sending and receiving device of each port. The EtherPIPE is a general character device, so userspace applications can send and receive packet data with any buffer size of write(2)/read(2) system call. The buffer size of write(2)/read(2) depends on the implementation of user commands.

When sending a packet using the Shell interface, the device driver gets the packet data from userspace and writes it to the TX ASCII ring buffer in the device driver. Next, the device driver converts the ASCII packet data in the TX ASCII ring buffer to the binary format of the Raw interface, and writes to the TX Raw ring buffer. Finally, the device driver reads the binary packet data from the TX Raw ring buffer and writes the packet data to the FPGA sending buffer using memcpy(3). When it finishes writing the packet data, the device driver updates the write pointer of the FPGA sending slot to the frame size. If the sending ring buffer still has data, the device driver repeats the sending process.

When sending a packet using the Raw interface, the device driver writes the binary data to the TX Raw ring buffer directly.

5.2 Hardware implementation

The receive logic is the same design as an ordinary NIC. The FPGA becomes a bus master and writes packet data directly to the host RX Raw ring buffer.

We implemented the transmission logic using PCI Express PIO (Programmed I/O) write. The TX device driver writes packet data to the FPGA using PCIe with write combining. Our prior experiments showed that normal PCI Express PIO writing performs at 40 MB/s but PIO writing with PCIe write combining performs at about 175 MB/s, or over 1Gbps.

We developed the FPGA TX logic with only a 32KB ring buffer in order to simplify the circuit. The Data format of the Raw interface has the frame size of the packet in the head of the data, so the FPGA TX logic can find the data boundary. In the design of our TX logic, the device driver doesn't need to update the write pointer of the FPGA ring buffer with respect to writing each packet. The device driver only updates the write pointer when it has finished all of the data in the TX Raw ring buffer in the device driver.

VI. Applications

This section shows examples of network scripting by EtherPIPE. Mainly, we explain the examples of primitive functions mentioned in Section 2.

6.1 Packet capture and generation

Command 1: packet generation

```
$ cat packet.dump > /dev/ethpipe/r1
```

Command 2: packet capture

```
$ cat /dev/ethpipe/r0 > packet.dump
```

Command 3: decapsulating Ethernet header and store IP packets

```
$ cut -d' ' -f4- /dev/ethpipe/0 > ip-packets.dump
```

Command 4: packet capture with PcapNG format

```
$ ethdump < /dev/ethpipe/r0 > dump.pcapng
```

Command 5: Port mirroring

```
$ cat /dev/ethpipe/0 \
| tee /dev/ethpipe/0 > /dev/ethpipe/1
```

Packet monitoring or packet analysis often employs tcpdump(1) or wireshark to store packets in Pcap format or PcapNG format files. EtherPIPE is suited to capture packets or to generate packets from both the Raw and Shell interfaces.

Command 1 shows an example to generate (replay) packets from the Raw interface. Simply reading a Raw format file by cat(1) and redirecting stdout to the raw interface to Port 1, packets will be sent through Port 1.

Command 2 describes packet capturing via shell scripting. In contrast to Command 1, the packet capturing scripts redirects the raw interface to a file.

Command 3 removes the Ethernet header of each packet from Port 0 and stores the IP packets into a file by redirection of stdout. Because of the shell interface of the character device (*/dev/ethpipe/0*), EtherPIPE enables cut(1) to separate the Ethernet header from a packet.

ethdump in Command 4 is our original shell command to store packets in the PcapNG format with hardware offloading of a FPGA network card. Through ethdump, a nanosecond-accuracy timestamp will be contained in each PcapNG format packet.

Port mirroring can be composed of chains of tee(1) and the shell interfaces of EtherPIPE like Command 5. The example of Command 5 mirrors received packets from Port 0 to Port 0 and Port 1. Adding a set of tee(1) and pipe (|), the number of the destination ports can be extended.

6.2 Mac address filtering

Command 6: filtering dstmac

```
$ gawk '$1=="001122334455"{print $0}' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 6 forwards packets from Port 0 to Port 1 on the network interface card only when the destination MAC address of a packet matches 00:11:22:33:44:55 by gawk(1).

6.3 Decapsulation and Encapsulation

Command 7: VLAN tagging

```
$ sed -e 's/^\([^ ]* \)\{2\}/&8100 00 01 /' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 8: VLAN untagging

```
$ sed -e 's/8100 00 01 //' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 9: VLAN translation

```
$ sed -e 's/8100 00 02 /8100 00 01 /' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 3 decapsulates the Ethernet header from a packet. An 802.1Q VLAN operation can be described by sed(1); VLAN tag-

```
#!/bin/bash
#
# usage: learning_switch.sh </dev/ethpipe/0

MY_PORT="0"
TEMP_DIR="/tmp/"

while true
do
  read FRAME

  DMAC=${FRAME:0:12}
  SMAC=${FRAME:13:12}

  # regist SMAC LEARNING TABLE
  if [ ! -f $TEMP_DIR$SMAC ]; then
    if [ $((0x$SMAC & 0x010000000000)) -eq 0 ]; then
      echo $MY_PORT >$TEMP_DIR$SMAC
    fi
  fi

  # search port number by DMAC
  if [ -f $TEMP_DIR$DMAC ]; then
    exec 3< $TEMP_DIR$DMAC
    read PORT 0<&3
    exec 3<&-
    echo $FRAME >/dev/ethpipe/$PORT
  else
    # flooding
    if [ ! $MY_PORT == "0" ]; then
      echo $FRAME >/dev/ethpipe/0
    fi
    if [ ! $MY_PORT == "1" ]; then
      echo $FRAME >/dev/ethpipe/1
    fi
    if [ ! $MY_PORT == "2" ]; then
      echo $FRAME >/dev/ethpipe/2
    fi
    if [ ! $MY_PORT == "3" ]; then
      echo $FRAME >/dev/ethpipe/3
    fi
  fi
done
```

Fig.6 *A learning switch script*

ging, VLAN untagging and VLAN translation are depicted in Command 7, 8 and 9 respectively.

6.4 Overlay tunneling

Command 10: L2 over TCP tunneling

```
[192.168.0.1] $ nc -l 9999 < /dev/ethpipe/0 \
> /dev/ethpipe/0
10.0.0.1] $ nc 192.168.0.1 9999 \
< /dev/ethpipe/0 > /dev/ethpipe/0
```

Command 11: ssh tunneling

```
$ cat /dev/ethpipe/r0 \
| ssh sample.com “cat >/dev/ethpipe/r0”
```

Several types of overlay tunneling can be described in EtherPIPE.

In Command 10, L2 over TCP tunnel is achieved by nc(1). nc(1) of the node 192.168.0.1 (the first line) reads packets from Port 0 and encapsulates read packets into a TCP (port 9999). In the second line, node 10.0.0.1 connects stream to 192.168.0.1 TCP port 9999, decapsulates packets, and throws decapsulated packets into port 0 of node 10.0.0.1.

On the other hand, Command 11 forwards captured packets to *sample.com* through ssh(1). This example shows a unidirectional ssh tunnel. If a bidirectional ssh tunnel is required, the same setting should be configured on *sample.com*.

6.5 Learning switch

Next, we show an example of shell scripting of complicated packet processing using broadcast transfer.

Fig. 6 shows the code of learning switch which has four ports. By using the network scripting environment, a full-function Learning switch can be developed in less than 50 lines. The example is developed using existing shell and UNIX commands. If a network script isn’t fast enough, you can use the APIs to write simple C utilities that can then be used in shell scripts.

In particular, computationally intensive functions and common network functions such as Forwarding DataBase (FDB) and routing database should be implemented as a command using in network script.

6.6 Exisiting network tools compatibility

Command 13: A virtual device for connecting an EtherPIPE and OS network stack

```
$ tappipe pipe0 </dev/ethpipe/0 \
> /dev/ethpipe/0 &
$ ifconfig pipe0
$ tcpdump -i pipe0
```

More advanced packet processing needs the functionality of various layers. For instance, the negotiation of ARP packet with network device and IP address setup is a function of both layer 2 and layer 3.

On the other hand, EtherPIPE is an input-and-output device of the packet data of a physical Ethernet port, and does not have those upper layer functions. It is necessary to make a network stack for each network application in the construction of a complicated network application.

Tappipe [22] is our EtherPIPE application which connects the Linux network stack with EtherPIPE. EtherPIPE provides raw packet data. It's easy to connect the OS network stack to our EtherPIPE application simply by converting the frame format.

Command 13 show usage of tappipe. In the first line, tappipe makes a virtual Ethernet device using physical port 0 on EtherPIPE. By specifying an EtherPIPE device as stdout and stdin of tappipe, the device is created using Linux TAP.

Because pipe0 is at the virtual Ethernet device for OS, it can offload the setup of an IP address, and the work of a data link layer called the negotiation of ARP to the existing network stack. Moreover, existing network software such as ping, dhclient, and wireshark can be used with an EtherPIPE port. By using Network scripting and tappipe together, we can narrow our focus to scripting our new network functionality.

VII. Performance analysis

We discuss the performance of our implementation and the potential of network scripting.

7.1 Performance of shell scripting

In order to test the performance of our hardware and software separately, we evaluate the performance of general network scripting with a dummy driver. There are two major factors. One is memory access throughput because entire packets are passed between UNIX commands through standard I/O. The other is character-based processing (e.g., string matching) used in UNIX commands. We have developed a dummy device driver for EtherPIPE. When reading from the device, the driver returns a dummy shortest (64 byte) Ethernet frame data pre-populated in the device driver. When writing to the driver, the driver simply copies the data into a buffer in the driver. The driver does not take it into account the timing constraints of the Ethernet specification (e.g., interframe gap). The measurements were performed on a PC with an Intel Core i5 760 running at 2.8 GHz and a ramdisk (tmpfs).

Table 2 shows the throughput of typical applications of network scripting using the dummy driver. The results show that simple packet capturing by cat(1) achieves more than 10 Gbps, but header rewriting using grep(1) and cut(1) is much slower.

Modern PCs have enough memory bandwidth (e.g., 10.6 GB/s for DDR3-1333) so that memory copy is not a bottleneck on simple network scripting. Moreover, we can take advantage of multi-core processors and their shared cache when using piped commands.

On the other hand, string matching used in header rewriting requires us to process data byte-by-byte. Many UNIX commands are line-oriented and need to check every byte in search for newline characters. Also, a string match stops when a match is found, but needs to search to the end of a packet when no match is found. It becomes even worse when regular expressions require backtracking. Therefore, the performance of network scripting is heavily influenced by the string matching rules used in a command.

To further improve the performance, one possible way is to provide a special command to extract specific header fields and apply commands only to the extracted field. For example, to apply grep(1) to only the Ethernet headers in packets, it would look something like "*epcmd --extract etherheader --command 'grep PATTERN'*". Another way is to keep the command syntax but add hardware-based offloading functions to make use of GPU, FPGA or other parallel processing methods.

7.2 Performance of EtherPIPE adapter

EtherPIPE can be used to rapidly build simple

Table II *Measuring Shell command-based packet proceedings using dummy driver*

dummy driver (frame size: 64B)	throughput (MB/s)	throughput / 1GE line-rate
capture[1]	1,097	11.52
MAC address filtering (one rule)[2]	833	8.75
MAC address filtering (five rules)[3]	184	1.93
decap ethernet header[4]	8	0.08

[1] cat /dev/ethpipe > dump
[2] grep "^002222222222" /dev/ethpipe > dump
[3] grep "^001111111111|^002222222222 ... ^005555555555" /dev/ethpipe > dump
[4] cut -d' ' -f4- /dev/ethpipe > dump

1GE line-rate (excluded Ethernet preamble and Interframe Gap): 95.24 MB/s
CPU: Intel Core i5 760 2.80 GHz

network tools such as packet capturing and generating tools. In particular, the performance of "*cat /dev/ethpipe/0*" and "*cat ./pkt > /dev/ethpipe/0*" shows primitive throughput, and is important in network scripting. Therefore, we compared the throughput of our FPGA-based implementation with that of tcpdump(1), by "*cat /dev/ethpipe/0*".

This evaluation has two purposes. One is to confirm the data transfer performance of our hardware design and implementation. The other is to show the accuracy of the hardware timestamp in our FPGA implementation.

We compare the performance of the FPGA-based EtherPIPE implementation with that of tcpdump(1) on a commodity NIC. The capturing machine is equipped with Intel Core i3-3220 and Intel 82579LM Gigabit Ethernet PHY.

The sender uses the Linux pktgen tool to send 10,000 packets with 64 or 1518 byte-long frame sizes. The sender transmits packets at the line-rate so that packets of the same size should arrive with the corresponding constant interval at the receiver.

Figure 7 shows the performance for 1518-byte-long packets; only the first 200 packets are shown in the figure. This time, tcpdump(1) was able to receive all the packets without any packet loss. However, the evolution of the timestamp is very bursty for tcpdump(1) because timestamps are taken by software when multiple packets are processed at a time (* The timestamp of tcpdump is shifted to the right for the delay of the first timestamp so as to not exceed the theoretical value of timestamp.) On the other hand, the timestamp of EtherPIPE evolves linearly, reflecting the theoretical packet arrival interval at 1Gbps line-rate.

These results show that our EtherPIPE implementation can work at 1Gbps line-rate even with 64-byte-long shortest size packets, as well as the accuracy and benefit of hardware-based timestamp.

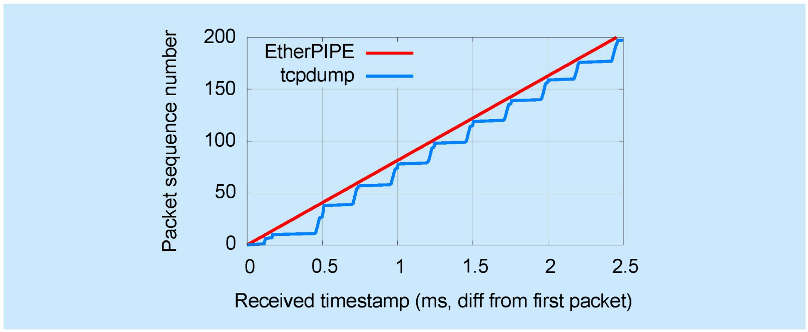

Fig.7 *The result of 200 packets which sending 1518 byte packets by pktgen and received by 'cat /dev/ethpipe/0' and tcpdump*

VIII. Discussion

In this section, we describe the limitations of the current EtherPIPE design and its implementation, and discuss the extensions to EtherPIPE.

8.1 Interface namespace

Our current device naming rules cannot express multiple network cards. For supporting multiple network cards, each EtherPIPE device may be put in subdirectory of each cards such as */dev/ethpipe/slot0/0*.

Because the control plane of packet processing is complex, it would be handled well by the network stack of OS.

We also will develop virtual network devices for OS network stack under */dev/ethpipe/*. Further discussion on the EtherPIPE device namespace is required, however, we do not examine it in detail due to the limitation of space.

8.2 Configuration of Interfaces

EtherPIPE currently focuses on lightweight scripting of packets over the data link layer. One of the limitations of the current EtherPIPE is that it ignores metadata of physical devices or socket options for TCP/IP. Ignoring the configuration functions, EtherPIPE can access packets in a simple way. To handle upper layers, some metadata handling scheme is required in EtherPIPE.

To implement such metadata, we can add other devices that have their own purpose for packet processing. The current EtherPIPE raw interface should be kept for performance. And the EtherPIPE shell interface may need to improve the ASCII format for usability on shell scripting even if it needs to pay a performance penalty. If it needs scalability, a device should be developed to set socket like options or store dynamic parameters in data format.

IX. Related Work

The concept of character-based network interfaces is not new. STREAMS [23] employs a modular architecture for implementing I/O between device drivers including network subsystems. Plan 9 [24] pushes it further to abstract everything including network as a file, and controls the network stack and services through files. Other systems such as x-kernel [25] and Netgraph [26] provide a framework for building a network stack by connecting protocol modules. The main focus of these systems is to provide an abstraction of network interfaces and protocol stack components.

There exist network interface devices that allow the programmer to access Ethernet frames such as DLPI (Data Link Provider Interface) [27], a STREAMS device driver of SunOS, and the TUN/TAP device driver. They are often used to implement a tunneling or bridging function in user space.

The purpose of EtherPIPE is to allow network scripting. To this end, it provides a simple abstraction of Ethernet ports as a character device, and converts Ethernet frames to and from ASCII representation for easy processing by UNIX commands.

X. Conclusion

Shell scripting is a powerful approach to manipulating files, however, it has not supported network processing. Our EtherPIPE allows shell scripting to deal with network devices and network I/O in the same manner as file devices and file I/O. Through the development of the EtherPIPE, we have shown that many packet processing operations can be described by chains of standard UNIX commands with standard input/output and pipe.

EtherPIPE is a low-layer network device yet, its data format is simple and easy to handle in commands and scripting languages. Therefore, EtherPIPE can be used not only for simple network scripting but also for more complex packet processing using scripting languages. We believe that EtherPIPE is suitable for SDN where simple packet manipulations are often required. As a lightweight implementation method for SDN applications, we hope the EtherPIPE opens a new paradigm of network programming.

References

[1] McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, and Turner J. Openflow: enabling innovation in campus networks. SIGCOMM CCR, 38(2):69–74, Mar. 2008.

[2] OpenFlowHub.org. Floodlight. http://floodlight.openflowhub.org/.

[3] Trema developers. Trema - Full-Stack OpenFlow Framework in Ruby and C. http://trema.github.com/trema/.

[4] Atlas A, Halpern J, Hares S, Ward D, and Nadeau T. An architecture for the interface to the routing system. Internet Draft, IETF, August 2013.

[5] Cisco Systems, Inc. Cisco OnePK - Cisco One Platform Kit. http://developer.cisco.com/web/onepk.

[6] Rizzo L and Landi M. netmap: Memory Mapped Access to Network Devices. SIGCOMM CCR, 41(4):422–423, Aug. 2011.

[7] Intel Corporation. Intel DPDK - Data Plane Development Kit. http://dpdk.org/.

[8] Krasnyansky M. Universal TUN/TAP device driver. https://www.kernel.org/doc/Documentation/networking/tuntap.txt.

[9] Catlett C and Foot G. Libnet Homepage. http://libnet.sourceforge.net/.

[10] tcpdump.org. TCPDUMP and LIBPCAP. http://www.tcpdump.org/.

[11] The Wireshark Foundation. Wireshark. http://www.wireshark.org/.

[12] Olsson R. pktgen the linux packet generator. In Proceedings of the Linux Symposium, Ottawa, volume 2, pages 11–24, Jul. 2005.

[13] Biondi P. Scapy. http://www.secdev.org/projects/scapy/.

[14] The Open Networking Foundation. OpenFlow Switch Specification. Technical Report Version 1.3.1 (Wire Protocol 0x04), Sep. 2012.

[15] McCanne St and Jacobson V. The bsd packet filter: a new architecture for user-level packet capture. In Proceedings of USENIX'93 Winter Conference, Jan. 1993.

[16] Hartmeier D. Design and performance of the openbsd stateful packet filter (pf). In Proceedings of USENIX ATC 2002, pages 171–180, Jun. 2002.

[17] Lidl J K, Lidl G D, and Borman R P. Flexible packet filtering: providing a rich toolbox. In Proceedings of BSDC'02, Feb. 2002.

[18] Pfaff B and Pettit J and Koponen T and Amidon K and Casado M and Shenker S. Extending networking into the virtualization layer. In Proceedings of HotNets-VIII, Oct. 2009.

[19] Kantee A. Environmental Independence: BSD Kernel TCP/IP in User Space. In Proceedings of AsiaBSDCon'2009, 2009.

[20] Degioanni L, Risso F, and Varenni G. PCAP Next Generation Dump File Format, Mar. 2004.

[21] Lattice Semiconductor Corporation. LatticeECP3 Versa Development Kit, 2013.

[22] Matsuya T. TAP device driver for EtherPIPE. https://github.com/sora/ethpipe/tree/master/software/tappipe.

[23] Ritchie M D. A stream input-output system. AT&T Bell Laboratories Technical Journal, 63(8):1897–1910, Oct. 1984.

[24] Presotto D and Winterbottom P. The organization of networks in plan 9. Winter 1993 USENIX Conference Proceedings, 1993.

[25] Hutchinson C N and Peterson L L. The x-kernel: An architecture for implementing network protocols. IEEE Transactions on Software Engineering, 17(1):64–76, January 1991.

[26] Elischer J and Cobbs A. FreeBSD man pages - netgraph(4). http://www.freebsd.org/cgi/man.cgi?query=netgraph&sektion=4.

[27] Oracle Corporation. man pages section 7: Device and Network Interfaces dpli(7P). http://docs.oracle.com/cd/E19253-01/816-5177/dlpi-7p/index.html.

Biographies

Yohei Kuga, is a Doctor Course Student in the Graduate School of Media and Governance at Keio University, Tokyo, Japan. His research interests includes Internet measurement and operating system support for networking. Email: sora@sfc.wide.ad.jp

Takeshi Matsuya, is a Doctor Course Student in the Graduate School of Media and Governance at Keio University, Tokyo, Japan. His research interests about high performance networking and network hardware.

Hiroaki Hazeyama, received his Ph.D. degree in Engineering from Nara Institute of Science and Technology (NAIST), Japan, in 2006. He is currently an assistant professor in the Graduate School of Information Science, NAIST. His research interests include network operation, network security, large-scale network testbed and testbed federation.

Kenjiro Cho, received the B.S. degree in electronic engineering from Kobe University, the M.Eng. degree in computer science from Cornell University, and the Ph.D. degree in media and governance from Keio University. He is Research Director with the Internet Initiative Japan, Inc., Tokyo, Japan. He is also an Adjunct Professor with Keio University and Japan Advanced Institute of Science and Technology, Tokyo, Japan, and a board member of theWIDE project. His current research interests include traffic measurement and management and operating system support for networking.

Rodney Van Meter, received a B.S. from the California Institute of Technology in 1986, an M.S. from the University of Southern California in 1991, and a Ph.D. from Keio University in 2006. His research interests include storage systems, networking, post-Moore's Law computer architecture, and quantum computing. He is an Associate Professor of Faculty of Environment and Information Studies at Keio University's Shonan Fujisawa Campus.

Osamu Nakamura, received a B.S. from Keio University in 1982, an M.S. in 1984, and a Ph.D. in engineering in 1993. He became assistant professor in the Faculty of Environment and Information Studies at Keio University's Shonan Fujisawa Campus in 1993, associate professor in 2000, and professor in 2006.