EtherPIPE: an Ethernet character device for network scripting

Yohei Kuga
Keio University

Takeshi Matsuya
Keio University

Hiroaki Hazeyama
NAIST

Kenjiro Cho
IIJ

Osamu Nakamura
Keio University

ABSTRACT

The UNIX command tools are designed to combine simple generic commands to accomplish various complex tasks. Meanwhile, in network programming, we often end up writing many similar functions and packaging functions of all network layers to build an application. In this paper, we propose 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. We developed a prototype of EtherPIPE as a character device driver for a commodity FPGA card. This paper argues for use cases of the EtherPIPE, and discusses enhanced formats of character devices for easier network scripting.

Categories and Subject Descriptors

C.2.3 [**Network Operations**]: [Network management]; D.4.4 [**Communications Management**]: [Network communication]

Keywords

Network I/O; Ethernet; Shell script; Software-Defined Networking; Device driver

1. INTRODUCTION

In this paper, we argue for the abstraction of a network interface and packet streams on an OS. Network processing is much more difficult than file processing on an OS. The question *"Why cannot we process packets by UNIX shell commands as readily and interactively as we can process files?"* is our simple motivation.

One of the reasons why network I/O is usually implemented as a network 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 method on network devices to access raw packets, and behaves as a 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 manner 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 easier by using a virtual network device[1] or one of several libraries to access raw sockets[2]. 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 Software Defined Network(SDN).

As a proof of concept, we developed EtherPIPE as an Ethernet device driver for a commodity FPGA network card on Linux[†]. 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 prototype implementation. We present examples of applications by EtherPIPE network scripting in Section 6 and discuss extensions and limitations

[†]EtherPIPE, `https://github.com/sora/ethpipe`

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
HotSDN'13, August 16, 2013, Hong Kong, China.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2178-5/13/08 ...$15.00.

of EtherPIPE in Section 7. After referring to related work in Section 8, we conclude this paper in Section 9.

2. 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 to program network applications as shell scripts, and try to define a primitive function set. We focus on network applications on data-link layer as a first step. Note that we call all of Ethernet frames, IP packets and / or TCP/UDP datagrams, "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, packet generator, packet capture tools, forwarding and header modification. 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 equipment requires the functions of packet sending and / or packet receiving at the minimum. Usual TCP connections employ both packet sending and packet receiving, in other words, an application uses `read(2)` and `write(2)` on a socket file descriptor for a TCP connection.

Packet capture tools such as tcpdump[3] or WireShark[4] require only the packet receiving function. On the other hand, packet generator applications, such as pktgen[5] or scapy[6], are mainly composed of the packet sending function.

OpenFlow[7] defines *Packet-in* and *Packet-out* functions. 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. Barkley Packet Filter (BPF)[8] OpenBSD Packet Filter (PF)[9] or IPFW[10] 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[11] provide more flexible forwarding control.

Header Modification

Header Modification is a key function to achieve forwarding, routing or encapsulation. 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[12] provides a header modification environment in the user space.

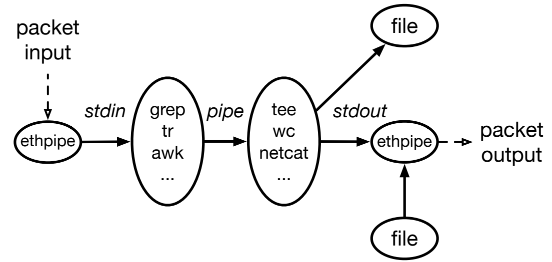

Figure 1: Network scripting

Table 1: 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

3. 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 system call to a device file for packet receiving, and the write 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 on a character device type network I/O 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 one liner 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 manner, a custom network scripting command can be reused in another network script. Through our network scripting, interactive processing against network streams can be realized.

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

Figure 2: EtherPIPE device names

```
[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 ...
```

Figure 3: Shell interface

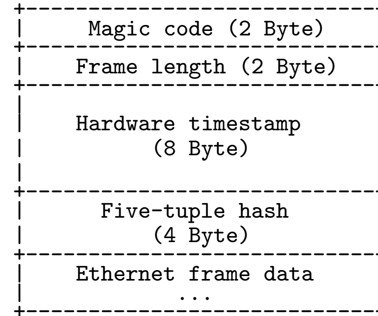

Figure 4: Raw interface

4. DESIGN OF DEVICE FORMATS

The EtherPIPE device tree and its name space are shown in Figure 2. For simplicity, we consider a 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/`.

On the other hand, 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 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 of 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 a high-bandwidth network connections. Figure 4 shows the format of the raw interface. This interface has metadata that indicates a hardware time stamp, a frame length, a five-tuple hash and Ethernet frame data. All EtherPIPE metadata are computed on network hardware to support network processing in the user space.

The hardware time stamp is described in a 64-bit counter value taken when the head of a packet arrives at a network hardware. Wireshark[4] and its capture format as PcapNG[13] support a nanosecond time stamp with NIC hardware counters. We use the same 64-bit time stamp to convert from the EtherPIPE raw format to the PcapNG format easily.

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.

5. PROTOTYPE

We developed a prototype of EtherPIPE as a character device driver for a commodity FPGA network card on Linux, and implemented almost all of the EtherPIPE primitive functions. The current prototype implementation does not perform at high speed because it is just for a proof of concept with a single TX/RX buffer on the FPGA. We will improve the performance in the next prototype. However, the contribution of this paper is just to show a proof of concept.

We support two FPGA cards, the LatticeECP3 versa kit[14] and the NetFPGA-1G[15]. The Lattice ECP3 versa kit has two 1000BASE-T interfaces and one PCI Express interface. NetFPGA-1G has four 1000BASE-T interfaces and one PCI interface. Both FPGA cards have multiple 1000BASE-T Ethernet ports, therefore, they can be used to test the forwarding case by EtherPIPE network scripting.

6. 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
```

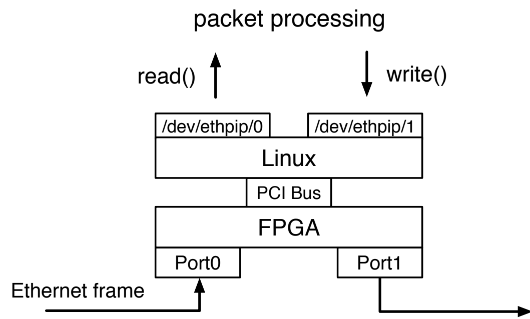

Figure 5: prototype

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
```

Packet monitoring or packet analysis often employs tcpdump[3] or WireShark[4] to store packets in Pcap format or PcapNG format files. EtherPIPE is suited to capture packets or to generate packets from Raw and Shell interface.

Command 1 shows an example to generate (replay) packets from 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 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`, nanosecond accuracy time stamp will be contained in each PcapNG format packet.

6.2 Mac address filtering

Command 5: filtering dstmac
```
$ awk '$1=="001122334455"{print $0}' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 5 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 decap and encap

Command 6: VLAN tagging
```
$ sed -e 's/^\([^ ]* \)\{2\}/&8100 00 01 /' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 7: VLAN untagging
```
$ sed -e 's/8100 00 01 //' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 8: VLAN translation
```
$ sed -e 's/8100 00 02 /8100 00 01 /' \
< /dev/ethpipe/0 > /dev/ethpipe/1
```

Command 3 decapsules of Ethernet header from a packet. 802.1Q VLAN operation can be described by `sed(1)`; VLAN tagging in Command 6, VLAN untagging in Command 7 and VLAN translation in Command 8 respectively.

6.4 Port mirroring and Forwarding

Command 9: Port mirroring
```
$ cat /dev/ethpipe/0 \
| tee /dev/ethpipe/1 > /dev/ethpipe/2
```

Command 10: forwarding
```
$ echo `arp -an | grep "eth1" | cut -f4 -d" "` \
| sed -e 's/://g' -e 's/\ /\\\|/g' > eth1.txt; \
grep -i `cat eth1.txt` < /dev/ethpipe/0 >      \
/dev/ethpipe/1
```

Port mirroring can be composed of chains of `tee(1)` and the shell interfaces of EtherPIPE like Command 9. The example of Command 9 mirrors received packets from Port 0 to Port 1 and Port 2. Adding a set of `tee(1)` and pipe (|), the number of the destination ports can be extended.

As with Command 5, Command 10 forwards packets according to the ARP table entries. As a feature of the UNIX shell programming, Command 10 is described in two lines that are separated by ";". In actual usage, a shell script for ARP entries of each interface will be made.

6.5 Overlay tunneling

Command 11: 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 12: 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 11, L2 over TCP tunnel is achieved by `nc(1)`. `Inc(1)` of the `192.168.0.1` node (the first line) reads packets from Port 0 and encapsulates read packets into a TCP (port 9999). In the second line, `10.0.0.1` node connects to `192.168.0.1` TCP port 9999, decapsules packets, and throws decapsulated packets into port 0 of `10.0.0.1` node.

On the other hand, Command 12 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`.

7. DISCUSSION

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

7.1 Performance

The current prototype hardware does not perform at high speed, and is not suitable for evaluation. Thus, 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

Table 2: Measuring Shell command-based packet proceedings using dummy driver

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

* cat /dev/ethpipe > dump
† grep "ˆ002222222222" /dev/ethpipe > dump
‡ grep "ˆ001111111111|ˆ002222222222 ... ˆ005555555555" /dev/ethpipe > dump
§ 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

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 Intel Core i5 760 2.8 GHz and 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 by 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. 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 to process data byte-by-byte. Many UNIX commands are line-oriented that needs 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 string matching rules used in a command. We will provide examples and guidelines on matching rules for users.

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 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 mention it due to the limitation of space.

7.3 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.

8. RELATED WORK

The concept of character-based network interfaces is not new. STREAMS[16] employs a modular architecture for implementing I/O between device drivers including network subsystems. Plan 9[17] pushes it further to abstract everything including network as a file, and controls networkstack and services throughfiles. Other systems such as x-kernel[18] and Netgraph[19] provide a framework for building a networkstack by connecting protocol modules. The main focus of these systems is to provide abstraction of network interfaces and protocol stack components.

There exist network interface devices that allow to access Ethernet frames such as DLPI (Data Link Provider Interface)[20], a STREAMS device driver of SunOS, and the TUN/TAP device driver[1]. 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.

9. CONCLUSION

Shell scripting is powerful utility for 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 with 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 commands using 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 of SDN applications, we hope the EtherPIPE opens a new paradigm of network programming.

10. REFERENCES

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[15] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. Netfpga: reusable router architecture for experimental research. In *Proceedings of PRESTO '08*, pages 1–7, Aug. 2008.

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

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

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

[19] Julian Elischer and Archie Cobbs. FreeBSD man pages - netgraph(4). `http://www.freebsd.org/cgi/man.cgi?query=netgraph&sektion=4`.

[20] 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`.