FlowOS: A Flow-based Platform for Middleboxes

Mehdi Bezahaf
School of Computing and Communications
Lancaster University, UK
m.bezahaf@lancaster.ac.uk

Abdul Alim
School of Computing and Communications
Lancaster University, UK
a.alim@lancaster.ac.uk

Laurent Mathy
Dept. of Electrical Engineering and Computer Science
University of Liege, Belgium
laurent.mathy@ulg.ac.be

ABSTRACT

Middleboxes are heavily used in the Internet to process the network traffic for a specific purpose. As there is no open standards, these proprietary boxes are expensive and difficult to upgrade. In this paper, we present a programmable platform for middleboxes called FlowOS to run on commodity hardware. It provides an elegant programming model for writing flow processing software, which hides the complexities of low-level packet processing, process synchronisation, and inter-process communication. We show that FlowOS itself does not add any significant overhead to flows by presenting some preliminary test results.

1. MOTIVATION

Middleboxes such as NATs, proxies, firewalls, IDS, WAN optimizers, load balancers, and application gateways etc. are integral part of today's Internet and play important role in providing high levels of service for many applications. A recent study [15] shows that the number of different middleboxes in an enterprise network often exceeds the number of routers. The odd thing about these middleboxes is that they do not have any standard and cannot interact with each other. Usually they come as vendor specific hardware boxes and requires special training for installation and maintenance. Often it is necessary to deploy new hardware to add new features to existing middlebox functionalities. The network operating cost increases significantly due to the lack of compatibility and upgradeability of middleboxes.

Recent trends in networking is to define a network management framework commonly known as *software defined networks (SDN)* that provides a programmatic interface upon which developers can write network management applications as necessary [6, 8, 11, 9, 10]. SDN decouples the control plane from the data plane so that the control plane can work independently from that of the data plane. Recall that SDN often offloads the control functions from switches and runs them as a software service at some centralized server, which allows control functions to be moved around [11, 13].

HotMiddlebox'13, December 9, 2013, Santa Barbara, CA, USA.

ACM 978-1-4503-2574-5/13/12 ...$15.00.
http://dx.doi.org/10.1145/2535828.2535836.

Note that SDN provides a platform for network management functions and does not offer services for middlebox functionalities, which are closely related to data plane.

Despite the heterogeneity of middleboxes, one common thing among them is that they all work on either specific or aggregate traffic flows. A network flow can be defined in many ways, but generally speaking, it is a sequence of network packets travelling from one point to another one and match certain characteristics. Note that a flow is unidirectional data stream that travels from a source to a destination, where a source or a destination could be an application, a physical port, or an aggregate of them. OpenFlow [11] defines a flow in terms of physical port, VLAN ID, MAC header fields, TCP/IP addresses, and IP protocols. Adam et. al. [7] show that middlebox functionalities that process traffic flows can be implemented as software modules and run on inexpensive commodity hardware namely x86 PCs with PCI Express network interfaces. They also claim that these processing modules can be run on virtual machines (VM) to provide isolation and mobility in terms of VM migration.

It is harder to write software for middlebox functions as there is no suitable high-level APIs for middle functions apart from libpcap [2]. Besides, middlebox functionalities require very high performance and are preferred to be run in the kernel space to avoid copying packets back and forth between kernel space and user space. The pcap library is a low-level interface for capturing IP packets. Programmers have to handle low-level packet processing in order to write any flow processing software. Suppose, a programmer wants to write a spam filter middlebox module, he is interested to SMTP header and email body and does not care about TCP/IP headers. But with pcap library, he has to process every TCP/IP packets to retrieve email message and then drop IP headers that belong to an email.

In this paper, we present the design and implementation of an Internet flow processing platform called FlowOS that provides a development environment for flow *processing modules* (PMs). FlowOS is a Linux kernel module-based system that captures IP packets for a flow (defined using OpenFlow primitives) and constructs one or more virtual byte *streams* to be processed by flow processing modules. Flow processing modules are also kernel modules that run as independent kernel threads and process data of specific streams. FlowOS provides a set of high-level APIs to write flow processing modules like writing a socket application. Programmers see a flow as a specific application byte stream and do not need to handle low-level packet processing. The rest of the paper

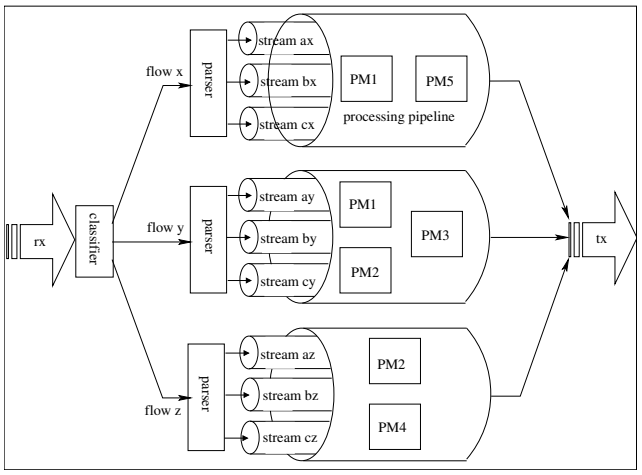

Figure 1: FlowOS architecture.

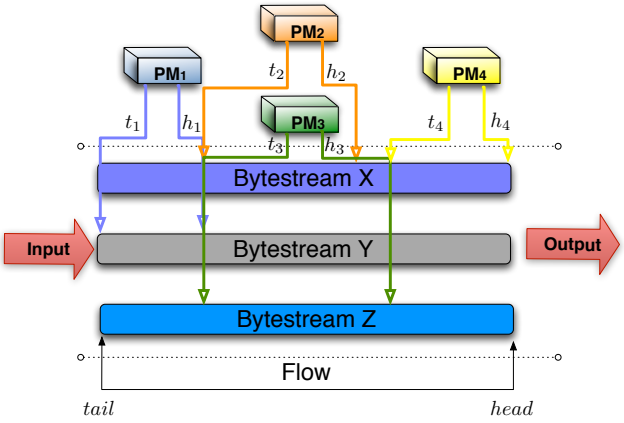

Figure 3: A pipeline of four PMs processing a flow.

is organized as follows. Section 2 describes FlowOS architecture. Section 3 describes how TCP sessions will stay alive using FlowOS, even after modifying the TCP stream. In Section 4, we present some preliminary results of FlowOS performance. In Section 6, we discuss how FlowOS reacts to high speed network interfaces. Finally, we conclude the paper in Section 7 by pointing some future work.

2. FLOWOS PROGRAMMING MODEL

FlowOS is implemented as a Linux kernel module-based Internet flow processing platform. Figure 1 depicts the main functional entities of FlowOS, that we describe in the following, along with the programming model provided.

2.1 Flows and Streams

As discussed earlier, a flow is a sequence of IP packets that satisfy certain criteria. It is the role of the classifier (see fig. 1) to group packets into flows.

One of the major goals of FlowOS is to expose in a clean way the data bytes, contained in these IP packets, relevant to specific protocol layers; for instance, one might be interested in the TCP payload of consecutive IP packets for a TCP application. To shield the programmer from the intricacies of IP packet structure or TCP segments, the FlowOS parsers extract, in the socket buffers of IP packets of a flow, start and end pointers for different protocols of interest, and arrange the identified corresponding packet segments in a doubly linked-list. Each linked-list thus consists of nodes that contain the start and end pointers, into a corresponding socket buffer (e.g. IP packet), for the bytes of a specific protocol. The set of bytes identified by each linked-list constitutes a virtual byte *stream*. FlowOS exposes streams via *stream pointers* that behave just as though all the bytes of a stream were contiguous in memory.

The IP stream corresponds to all the bytes making up the IP packets of the flow (packet headers and payloads). Similarly, the TCP stream corresponds to IP packet payload bytes only. Finally, the application stream is made up of the TCP payload bytes.

Note that streams encapsulated into one another: for instance, a TCP stream is a sub-stream of an IP stream. Our stream model assumes that a higher level (sub-stream) stream is completely contained within a lower level (super-stream) stream. In fact, if an IP packet does contain one byte of a stream, all the subsequent bytes of the packet do also belong to that stream – in other words, the end pointer that identifies the last byte of a given stream in an IP packet, necessarily points to the last byte of this packet, if that packet does contain any of this stream's bytes.

2.2 Flow Processing

Beyond the extraction of streams from flows, FlowOS itself does not process flows but allows separate flow processing modules to process a flow. A flow processing module (PM) can be thought of as a middlebox functionality that works on a specific stream of a flow. A PM runs as a kernel thread.

FlowOS supports the concept of processing pipeline, that is, a sequence of processing stages, where at each stage one or more PMs can process the flow concurrently.

Once a processing pipeline is configured for a flow, PMs can process data in the flow. Since a flow is shared by all the PMs on a processing pipeline, each PM has two stream pointers – head and tail – that delimit a "window" of available data for the PM to process at any one time. FlowOS injects data into a processing pipeline by moving tail pointers of the first-stage PMs. A PM releases data to the next stage PM by moving that PM's tail pointer, while a PM indicates it has finished with some data by simply moving its own head pointer past that data.

Note that by ensuring that the next stage PM's tail pointer is always equal to its own head pointer, a PM can guarantee that all the downstream PMs in the pip-eline are accessing blocks of data that do not overlap with its own data window.

On the other hand, when several PMs work in parallel at a same pipeline stage, their tail pointers must be "shared", that is data gets released to all these PMs simultaneously. Since these parallel PMs run independently and may process the same portion of bytes at the same stage with different speeds, particular attention must be paid to determine when to release data bytes to the next stage. In order to simplify the synchronization problem, FlowOS imposes that the tail pointer of the next stage is always equal to the head pointer of the "slowest" previous parallel PM, that is, equal to the "trailing head" pointer of the previous parallel stage. To efficiently support this policy, FlowOS uses a **min-heap** of

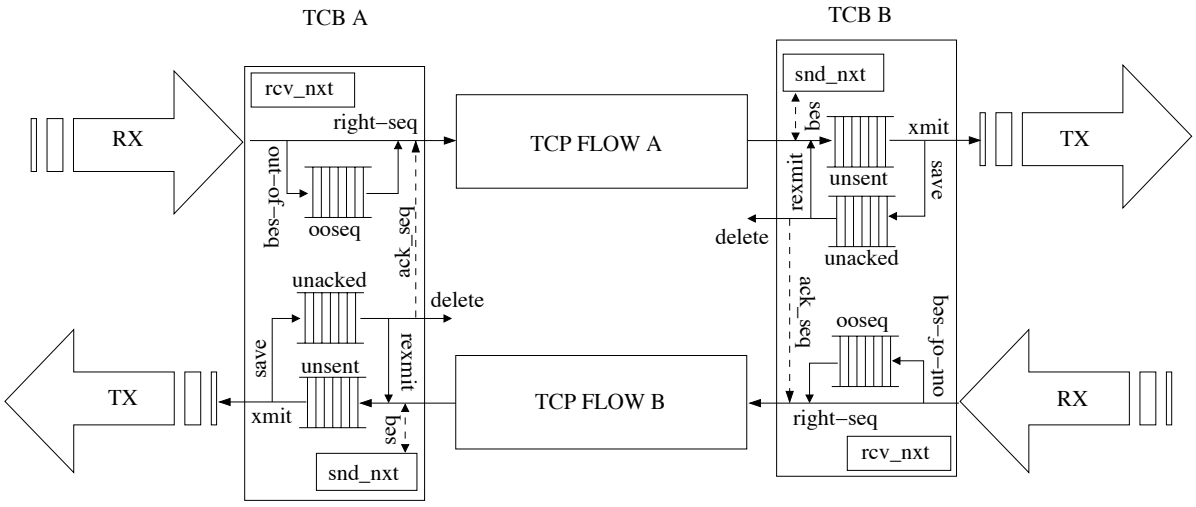

Figure 2: FlowOS TCP architecture.

head pointers of PMs at each parallel stage. When a PM has finished processing a block of data and tries to release data by moving its head, the system checks the PM's head pointer with the heap top. If the PM's head is at the top of the heap, the block of data is actually passed to the next stage PMs (by updating its tail pointer). Note that FlowOS PM developers do not need to worry about the synchronization issues and simply call FlowOS API to release data as if it is the only PM processing the data.

Finally, note that thanks to our definition of streams of a flow and the PM synchronization mechanism via head and tail stream pointers, it is easy to support the operations of pipelines whose PMs operate on different streams, making FlowOs a particularly modular and flexible platform.

2.3 Packet Forwarding

The trailing head stream pointer of a pipeline delineates treated from "in-process" data: all the data that occurs before this trailing head pointer can be forwarded through towards the destination.

As FlowOS constructs virtual streams out of socket buffers and PMs manipulate stream data directly in these buffers, a packet can be forwarded once the trailing head pointer gone past it.

The FlowOS TX handler is responsible for this forwarding, reconciling packet headers by taking all the changes made by PMs into account if necessary. It runs as an independent kernel thread, which receives FlowOS packets from all the flows in the system,

3. TCP IN THE MIDDLE

Depending on the functionality implemented by the PMs, these may require that FlowOS reconstructs TCP flows before injecting these flows into the processing pipeline. One simple solution would be to terminate TCP connections at the middlebox and use two different TCP sessions, one between the source and the middlebox and the other between the middlebox and the destination. However, such an approach presents several drawbacks: explicit client configuration may be required; high copy overhead has the data would have to be copied from one TCP connection to another; difficulties with migrating middleboxes for load sharing/balancing; etc.

Instead, FlowOS implements minimal logic to ensure correct TCP behaviour while not terminating TCP connections on the middlebox. Figure 2 illustrates the FlowOS TCP flow handling process.

When FlowOS receives the first TCP SYN packet for a flow, it creates two TCP connection control blocks (TCB), one behaves much like a TCP server and the other behaves much like a TCP client, and initialises them with appropriate TCP information from the packet. Subsequently, when FlowOS receives the TCP SYNACK packet for some flow, it looks for the peer flow (peer flows are two opposite flows of a TCP connection) and if the peer flow is found, FlowOS populates TCBs related to this TCP connection with TCP information from the SYNACK packet and forwards the packet. Once FlowOS receives the TCP ACK packet for a TCP connection and forwards it, TCBs related to this TCP connection are fully initialised and control TCP flows much like two separate TCP connections. These two TCBs separate the TCP streams between the source and FlowOS and between FlowOS and the destination.

During flow reconstruction, if a TCP flow receives an out of sequence segment, the TCB (input) associated to this flow puts the received segment into its OOSEQ (Out of Order SEQuence) queue for reordering. Note that the OOSEQ queue maintains segments in increasing order of TCP sequence numbers and adjusts segments if one overlaps with another by trimming overlapping bytes. Upon receipt of an in-order data segment, the TCB checks the OOSEQ queue to see if any previously out-of-sequence segments become in-order, and passes all these segments to the stream, updating the next expected sequence number accordingly.

The logic must also take into account that part of the TCP stream is inside the processing pipeline (and thus has not be sent towards the destination), and react appropriately to external TCP events: such as silently dropping retransmissions of data that is still in the pipel-ine, for instance. Another example is sending an acknowledgement for retransmitted segments that have already been acked by the destination.

Clearly, for this logic to work correctly, the two peer flows of a TCP connection must be kept in tight synchronization. This is easy to do if the two peer flows reside on the same FlowOS box, but will require inter-box communications if this is not the case.

Two fundamental operations provided by FlowOS is the deletion and/or the addition of bytes in a stream. FlowOS must therefore manage a constant mapping between the sequence numbers in the original flow, and the corresponding sequence numbers in the modified flow, so as to hide flow modifications from both ends of the connection and avoid breaking their TCP instances. Furthermore, as the output TCP stream may be different from the input TCP stream, FlowOS must buffer modified parts of the TCP stream long enough to ensure correct retransmission operations: for instance, if a modified segment has been lost on the way to the destination, a retransmission, by the source, of the original data must result in a retransmission, by FlowOs, of the modified data towards the destination.

4. PERFORMANCE

4.1 Sequential/Serial PMs

For our first performance test of FlowOS, we deployed, on the UCL HEN platform, three PCs in a linear topology (*source, middlebox*, and *sink*) and a switch, where the PCs are Dell PowerEdge 1850 servers with a 3.0GHz single core Xeon processor, 2GB RAM, and Intel PRO 1000 MT Gigabit network interfaces and they are connected by means of a Force10 E1200 switch. PCs run Debian Linux with kernel 3.1.1.

First, we measure the throughput for vanilla Linux kernel without FlowOS on the *middlebox* for different sized packets starting from small 64 bytes increasing up to maximum 1500 bytes.

We then run FlowOS on the *middlebox* and define a flow that captures all IP packets coming from the *source*. We define a processing pipeline with a single read/write PM (network address translator (NAT)) and run the same test. We observe that FlowOS with a single read/write PM does not have noticeable overhead on the traffic.

Next we want to see how FlowOS performs when multiple PMs are put on a processing pipeline. For this, first we create a pipeline of two PMs that work serially (the second PM can access data only after the first PM has released it), so there is no contention. We put our IP checksum module at the first position and NAT at the second stage and run the same test again. We observe that FlowOS with two PMs in sequence works at the same rate as with one PM. We then change the processing pipeline to make two PMs process data concurrently, where the PM that works slowly releases data to the system for forwarding. FlowOS works without causing any performance overhead in this scenario as well as shown in Figure 4.

4.2 TCP sessions

Middleboxes sometimes examine the content of application traffic and filter out unwanted data from a flow (e.g., dynamic ad blocker, WAN optimizer). In this section, we examine the performance of such processing module (*viz., adblocker*[1]*, which in FlowOS is less than 20 lines of code).* We have used QEMU based virtual network to perform this test, which gives us a relative performance data. In this experiment we have used the same linear topology used previously. The sink runs iperf server and the source sends traffic to the sink via the middlebox.

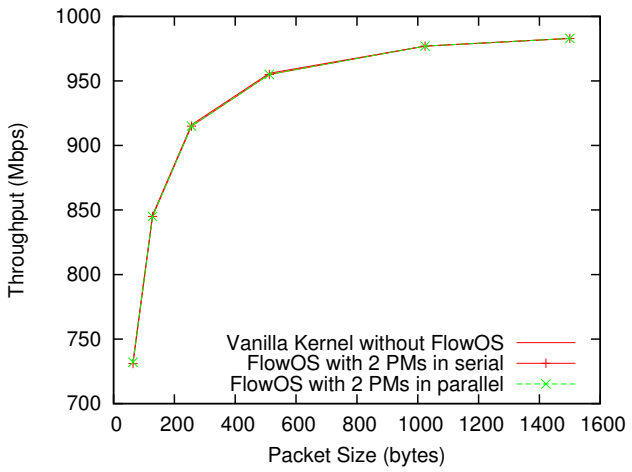

Figure 4: Throughput of FlowOS with two PMs in sequential and in concurrent settings.

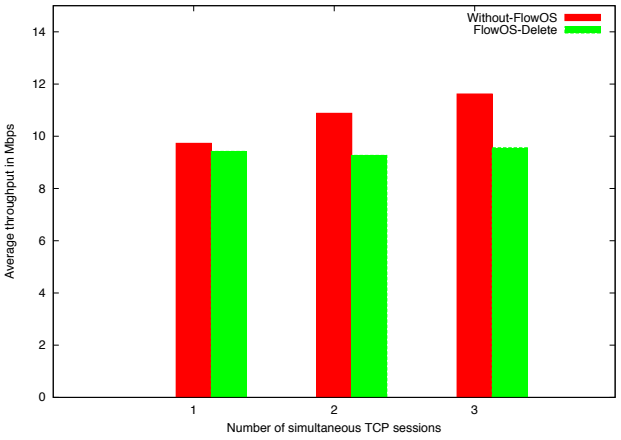

Figure 5: Relative throughputs of FlowOS adblocker processing module that deletes data from the stream w.r.t. vanilla kernel forwarding.

The adblocker looks for javascript tags to identify google ads and clips the content of the flow to eliminate the ad. Note that it changes the length of TCP segments which in turn changes the socket buffer carrying the segment. FlowOS reflects these changes and reconciles IP packets before sending them out. FlowOS also adjusts the TCP sequence number and respective acknowledgements. Figure 5 shows the relative performance of adblocker with respect to vanilla Linux kernel forwarding.

The throughput falls as the number of simultaneous flows increased when adblocker is added to each of these flows.

[1]This PM has been developed at the CHANGE Bootcamp at UCL, Belgium

Note that adblocker removes some part of a TCP segment which requires data to be moved in order to fix the IP packet for retransmission. A single clipping requires all subsequent packets to be fixed by recomputing sequence and acknowledgement numbers and hence TCP checksum recomputation. Therefore, it is important for middleboxes that add or remove data to streams to have enough computing power for additional processing. In our experiments, we have used virtual machines which are not very powerful.

5. RELATED WORKS

Mohamed et. al. [3] proposed a software middlebox platform called ClickOS, which combines the Click [12] modular router and MiniOS [1] kernel together to implement middlebox functions using Click components. The tiny footprint of MiniOS makes ClickOS a very lightweight DomU host under Xen [4] hypervisor and can easily be migrated to other hosts. Because of its reliance on Click, ClickOS is a packet-processing platform, and does not exhibit the ease of flow processing programming that FlowOS does.

In [14] authors proposed a software-centric middlebox platform for general-purpose hardware platforms. They discussed the requirements, challenges, and advantages of such a system without giving actual implementation.

Zheng et. al. [5] proposed a clean-slate system for network control and management that provides services to the network control applications such as communication between applications, scheduling of application executions, feedback management, concurrency management, and network state transition management.

NOX [8] provides a global view of the entire network including the switch-level topology and the services being offered by means of a set of "base" applications. At the application programming model level, NOX allows applications to register for notification of specified network events and processes these events by defining event handlers.

Teemu Koponen et. al. [9] proposed an extension of NOX called Onix that provides flexible distribution primitives allowing application designers to implement control applications without re-inventing distribution mechanisms, and while retaining the flexibility to make performance/scalability tradeoffs as dictated by the application requirements.

Our FlowOS work complements these efforts, by providing a flow-oriented programmable data path for middleboxes.

6. DISCUSSION

The process of creating a packet for transmission on the network involves assembling multiple pieces of data. Packet data must often be copied in from user space, and the headers used by various levels of the network stack must be added as well. This assembly can require a fair amount of data copying. One of the main features of 10G network interfaces is the fact that they perform scatter/gather I/O, which means that the packet does not need to be assembled into a single chunk by the kernel, and much of that copying will be avoided. Scatter/gather I/O also enables "zero-copy" transmission of network data directly from user-space buffers. Each chunk of data is called a fragment and can be saved in different memory page then the other fragments of the same packet. The network interface take care of assembling all the fragments in a single packet before transmitting them by looping through all fragments and mapping

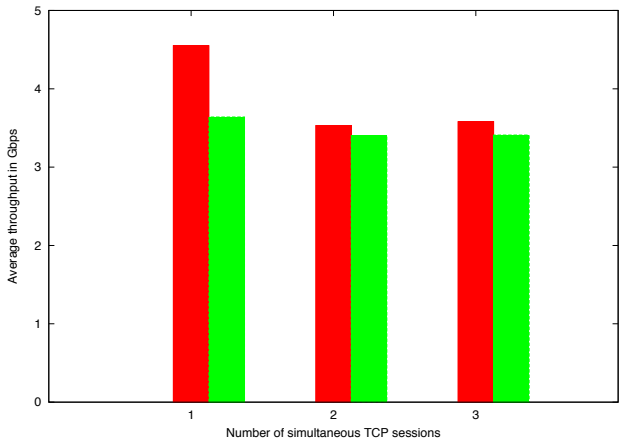

Figure 6: FlowOS on high speed data.

each one of them for a DMA transfer. Our current version of FlowOS, where one skb is considered as a continuity of bytes in the same memory page, does not take into account this skb implementation.

In order to further our tests, we ran three virtual machines (using QEMU) on a Dell server PowerEdge T620 with a 2.2GHz four cores Xeon processor, 4GB RAM, and Intel Ethernet X540 DP 10GBASE-T network interface. The three virtual machines were connected in a linear topology (*source, middlebox*, and *sink*).

Given that our current version of FlowOS does not take into account this Scatter/gather I/O feature, we use the kernel function *skb_linearize()* to linearise all bytes contained in the same skb (including all the fragments). Figure 6 shows clearly how much the kernel function *skb_linearize()* affects the performance by slowing down the throughput for vanilla Linux kernel without FlowOS to 3.50Gbps in average. Using FlowOS, the performance slightly decreases, which is normal due to the processing performed.

7. CONCLUSION

Internet flow processing is ubiquitous and operators use specialised middleboxes for processing Internet flows. These closed proprietary middleboxes are expensive and it is difficult to add new functionalities to them. Moreover, they complicate the network management as they do not comply with any standards. Recently, researchers are talking about programmable platforms for middleboxes. However, most of them consider flows are sequence of IP packets and middleboxes are network layer entity. In this paper, we presented a programmable platform for commodity hardware middleboxes called FlowOS, which extracts streams from a flow for middlebox to process. Of course, one can write a middlebox software that process IP packets such as NATs or IP firewalls using FlowOS, but FlowOS provides socket like interface for writing middlebox software that process application byte streams instead of IP packets.

FlowOS provides an elegant programming model for writing flow processing software. Flows are shared amo-ng multiple PMs but FlowOS hides the complexities of process synchronisation even if they process the data concurrently. It also hides the complexities of inter-PM communications by providing an integrated inter-platform communication

model. A PM communicates with another PM transparently of their location: they could be on the same machine or run on different machines.

We have performed some basic tests to evaluate its performance, which shows that FlowOS itself does not add any significant overhead to network flows compared with the line rate, and it takes completely into account TCP sessions. We are developing some application level processing modules to carry out extensive testing.

As future work, we will integrate memory isolation for different flows and PMs so that the malfunction of one PM or flow does not affect the others. We are also planning to incorporate PM and flow migration so that operators can move flows and/or PMs on demand to respond to network requirements.

8. REFERENCES

[1] Mini-os-devnotes. `http://wiki.xen.org/wiki/Mini-OS-DevNotes`.

[2] Tcpdump and libpcap. `http://www.tcpdump.org`.

[3] Ahmed, M., Huici, F., and Jahanpanah, A. Enabling dynamic network processing with clickos. *ACM SIGCOMM Computer Communication Review 42*, 4 (October 2012), 293 – 294.

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (October 2003), 164 – 177.

[5] Cai, Z., Dinu, F., Zheng, J., Cox, A. L., and Ng, T. S. E. The preliminary design and implementation of the maestro network control platform. Tech. rep., Rice University, October 2008.

[6] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM Computer and Communication Review 35*, 5 (2005), 41–54.

[7] Greenhalgh, A., Huici, F., Hoerdt, M., Papadimitriou, P., Handley, M., and Mathy, L. Flow processing and the rise of commodity network hardware. *ACM Computer Communication Review 39*, 2 (November 2009).

[8] Gude, N., Koponen, T., Pfaff, J. P. B., Casado, M., McKeown, N., and Shenker, S. NOX: Towards an Operating System for Networks. *ACM Computer Communication Review 38*, 3 (July 2008).

[9] Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., and Hama, T. Onix: A distributed control platform for large-scale production networks. In *Usenix Symposium on OSDI* (Vancouver, BC, Canada, October 2010).

[10] Lu, G., Guo, C., Li, Y., Zhou, Z., Yuan, T., Wu, H., Xiong, Y., Gao, R., and Zhang, Y. Serverswitch: a programmable and high performance platform for data center networks. In *USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA, 2011).

[11] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communication Review 38*, 2 (March 2008), 69 – 74.

[12] Morris, R., Kohler, E., Jannotti, J., and Kaashoek, M. F. The click moduler router. *ACM Operating Systems Review 34*, 5 (December 1999), 217 – 231.

[13] Pfaff, B., Pettit, J., Koponen, T., Amidon, K., Casado, M., and Shenker, S. Extending networking into the virtualization layer. In *ACM Workshop on Hot Topics in Networks (HotNets-VIII)* (New York City, NY, October 2009).

[14] Sekar, V., Ratnasamy, S., Reiter, M. K., Egi, N., and Shi, G. The middlebox manifesto: Enabling innovation in middlebox deployment. In *ACM Workshop on HotNets* (Cambridge, MA, USA, November 2011).

[15] Sherry, J., and Ratnasamy, S. A survey of enterprise midlebox deployments. Tech. rep., University of California at Berkeley, February 2012. Technical Report No. UCB/EECS-2012-24.