# FlowOS: A Programmable Platform for Commodity Hardware Middleboxes

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

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

## Categories and Subject Descriptors

Computer Networks [**Network Components**]: Middle boxes / network appliances

## General Terms

Design

## Keywords

Middlebox, Flow Processing, Stream Processing, Internet, Architecture

## 1. MOTIVATION

Middleboxes such as NATs, proxies, firewalls, IDS/IPSes, 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 [5, 12, 13, 16, 9]. 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 [13, 6]. 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 [13] defines a flow in terms of physical port, VLAN ID, MAC header fields, TCP/IP addresses, and IP protocols. Adam et. al. [3] 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.


*CFI* '13, June 5 – 7, 2013, Beijing, China

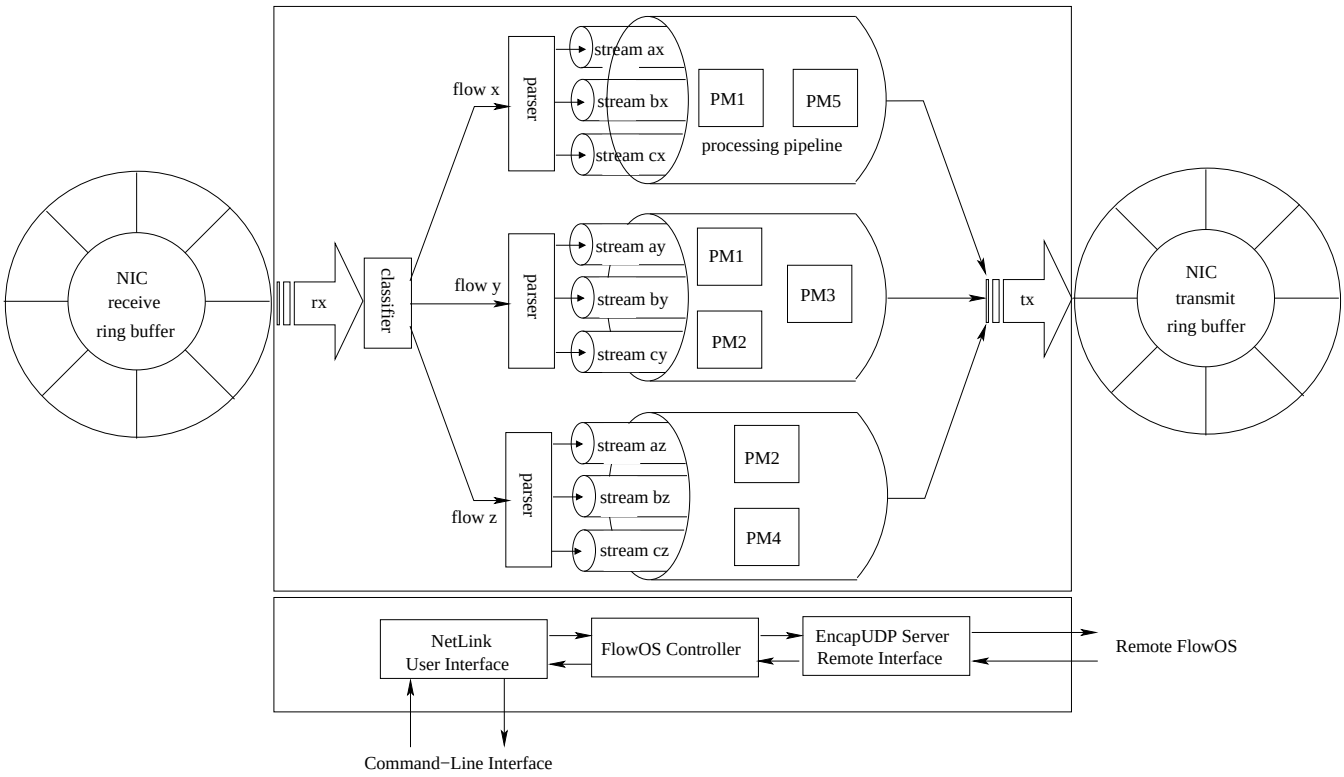


**Figure 1: FlowOS architecture.**

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 is organized as follows. Section 2 describes FlowOS architecture. Section 3 explains the flow processing module programming framework with examples. In Section 4, we present some preliminary results of FlowOS performance and we discuss related works in Section 5. Finally, we conclude the paper in Section 6 by pointing some future work.

## 2. FLOWOS ARCHITECTURE

FlowOS is a Linux kernel module-based Internet flow processing platform. Figure 1 depicts the main functional entities of FlowOS. Following sections explain the functionalities of each of the components of FlowOS.

### 2.1 Streams and Flows

As discussed earlier, a flow is a sequence of IP packets that satisfy certain criteria. In FlowOS, we are interested to data bytes of these IP packets relevant to a specific protocol; for instance, we are interested to TCP payloads of consecutive

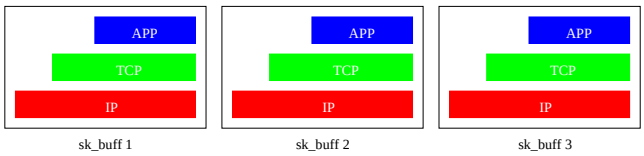


**Figure 2: An abstract view of a flow with three virtual streams.**

IP packets for a TCP application. In other words, an application is neither interested to IP packets nor to TCP segments but the application data. FlowOS extracts the start and end payload pointers for different protocols of interest in the socket buffers of IP packets of a flow, and arrange them in a doubly linked-list. The linked-list of payload buffers of a specific protocol is treated as a virtual byte *stream*. Figure 2 shows how virtual streams of a flow are constructed out of socket buffers. The IP stream is the list of IP packets that is buffers delimited by the pointer to the network header and the pointer to the end of IP payload in socket buffers. Similarly, the TCP stream is the list of buffers delimited by the pointer to the transport header and the pointer to the end of TCP payload in the socket buffer. Finally, the application stream is the list of buffers delimited by the start and end pointers of application payload in socket buffers. Note that streams are related to one another, for instance, a TCP stream is a sub-stream of an IP stream. FlowOS assumes that a higher level (sub-stream) stream is completely contained within a lower level (super-stream) stream.

A special pointer, called *stream pointer*, is used to access data of a stream. In FlowOS, a flow is represented by a queue

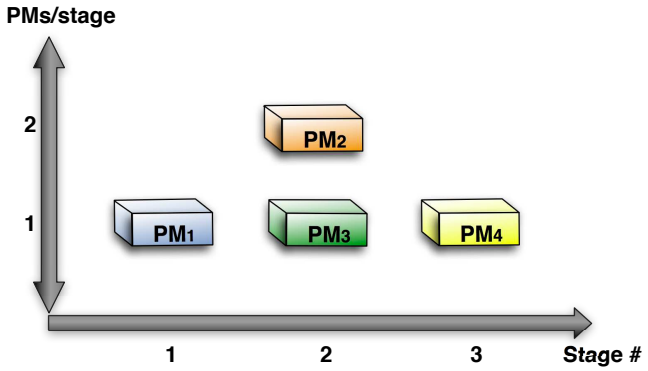


**Figure 3: Pipeline stages, where $pm_2$ and $pm_3$ are at stage 2 processing in parallel while $pm_1$ and $pm_4$ are processing at stage 1 and stage 3 respectively.**

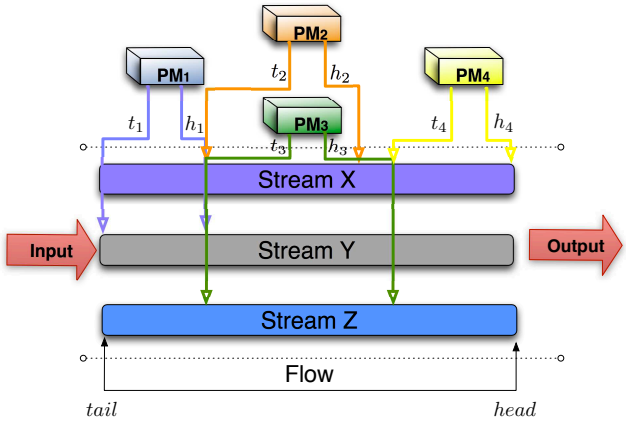


**Figure 4: A pipeline of four PMs processing a flow.**

of socket buffers along with a flow ID that defines the flow, a human readable name, the sequence of protocols of interest, and a processing pipeline of flow processing modules.

## 2.2 Packet Capturing

FlowOS captures packets from the network interface associated to some flow before it goes to Linux network protocol stack by registering an RX handler for the network interface. Ideally, OpenFlow switches are used to classify packets for a flow. However, FlowOS comes with a complimentary software packet classifier to classify packets when multiple flows share the same network interface. RX handler invokes the packet classifier to match the packet against the flows in the system. If a packet does not belong to any of the flows in the system, RX handler passes the packet to the kernel. Otherwise, RX handler inserts the packet to the matching flow and returns RX_HANDLER_CONSUMED to tell the kernel that packet is taken away.

When RX handler sends a packet to a flow, FlowOS has to extract the start and end pointers for different streams of the flow. FlowOS comes with a set of protocol parsers and are called in a sequence as needed by the flow to extract these pointers. For instance, a flow that is defined to process spams needs IP, TCP, and SMTP protocol parsers in sequence to build IP, TCP, and SMTP streams.

## 2.3 Flow Processing

FlowOS itself does not process flows but allows separate flow processing modules to process a flow. A flow processing module (PM) is a middlebox functionality that process a flow such as NATs, proxies, IDSes, etc. In FlowOS, a PM is a kernel thread that works on a specific stream of a flow (Section 3 explains how to write a FlowOS PM).

In FlowOS, a flow can be processed by one or more flow processing modules either sequentially or in parallel or in a combination of them. In order to process a flow, one has to define a *processing pipeline* for the flow. A processing pipeline can be seen as a sequence of processing stages, where at each stage one or more PMs can process concurrently the flow (Figure 3).

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 maintains two stream pointers – head and tail – that delimit the amount of data available for a PM to process. FlowOS injects data into a processing pipeline by moving tail pointers of the first-stage PMs. Once a PM has finished processing some data, it releases that data to the next-stage PM by moving it's head and next-stage PM's tail pointers.

Note that PMs in different sequential stages work on different blocks of data of the flow, but PMs at a particular stage work on the same data block concurrently. Therefore, head and tail pointers management is crucial for safety and performance of FlowOS. Consider the processing pipeline of Figure 4, where $\text{pm}_1$ is placed at stage 1 before $\text{pm}_2$ and $\text{pm}_3$ at stage 2, which implies that $\text{pm}_2$ and $\text{pm}_3$ should not have access to data beyond the head of $\text{pm}_1$. The following invariants are to be preserved by head and tail pointers to ensure that all the PMs at stage $i$ have finished processing the data bytes before releasing them to PMs at stage $i+1$.

1. A PM's head and tail must be within the system head and tail.
2. The head of a PM cannot go beyond its tail.
3. The tail of a PM cannot go beyond the head of the previous PM.

Since PMs run independently and may process the same portion of bytes at the same stage with different speeds, it is difficult to determine when to release data bytes to the next stage and by which PM. In order to simplify the synchronization problem, FlowOS uses a **min-heap** of head pointers of PMs at each 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. Note that FlowOS PM developers do not need to worry about the synchronization issues and simply calls FlowOS API to release data as if it is the only PM processing the data. Each PM can be either read-only or read and write.

## 2.4 Packet Releasing

Once all the PMs of a processing pipeline have finished processing a flow, FlowOS forwards the traffic toward the destination. Note that FlowOS constructs virtual streams out socket buffers in a flow and PMs manipulate data in a stream and are not aware of underlying packet structures.

FlowOS has to resolve the inconsistency between packet headers and the payload before sending them out to the network. FlowOS TX handler is responsible for reconciling packet headers by taking all the changes made by PMs into account if necessary. It runs as an independent thread, which receives FlowOS packets from all the flows in the system, reconciles protocol headers where necessary, and determines the output network interface by performing route lookups. It then enques IP packets to appropriate NIC's output queue for transmission.

## 2.5 FlowOS Controller

FlowOS controller, an independent kernel thread, is responsible for managing flows, processing pipelines, and flow processing modules. It manages flows, processing pipelines, and PMs by executing user commands received either from the local NetLink socket user interface or from the remote UDP socket interface. FlowOS controller defines two different types of messages: FlowOS control messages and PM control messages. Upon receipt of a message, the controller performs necessary sanity checks to ensure that it is a valid FlowOS message and then invokes the appropriate command handler function if it is a FlowOS control message. Otherwise, the controller dispatches the message to the appropriate PM.

FlowOS comes with a simple command-line user interface (CLI) program, which provides a set of user-friendly commands to interact with the FlowOS controller.

The FlowOS CLI creates a raw NetLink socket of type NETLINK_FLOWOS (17) to connect to FlowOS. It implements FlowOS control commands and is able to load shared library for executing PM control commands. Note that every FlowOS command takes an optional –**host** argument, which is used to specify the IP address of FlowOS host, to allow users to manage FlowOS remotely.

## 2.6 Inter-platform Communication

Recall that an Internet flow is a one-way stream of IP packets, however most of the Internet applications need two-way interactions at least for control information and these peer flows are inter-dependent. The peer flows could be processed in different flow processing platforms. Therefore, a flow processing system often needs to communicate with another flow processing system possibly at a remote host. FlowOS uses a kernel space encapsulated UDP server to communicate with remote flow processing platforms[1].

When FlowOS receives a message from a remote FlowOS via encapsulated UDP socket, it performs some basic sanity checks (e.g., the socket is initialized for UDP encapsulation and has valid encapsulation magic, pulls out UDP encapsulation header, and the message contains a valid FlowOS command), and then dispatches to FlowOS controller. Note that when a user issues a FlowOS command with remote host address, FlowOS controller constructs an encapsulated UDP message from it and forwards to the remote host. When a PM needs to send PM control message to its peer which is running at a remote host or to send response to the client at remote host, it constructs and sends encapsulated UDP message to the remote host.

# 3. FLOWOS PM STRUCTURE

[1]Note that security issues are not discussed in this paper and can be considered as future work.

Since FlowOS PMs are kernel modules and run as independent kernel threads, they have to define *module_init()*, *module_exit()*, and a thread function. In addition to these functions, a FlowOS PM has to define the following set of functions, where *modname* is the name of the PM and is used to distinguish the same function from different PMs:

1. *uint32_t modname_protocol()* to return the protocol this PM processes. This is used by the FlowOS to construct the appropriate virtual stream for the PM.
2. *uint8_t modname_type()* to return the type of the PM. A FlowOS PM can either be **PMODULE_RONLY** or **PMODULE_RDWR**.
3. *uint8_t modname_msgcount()* to return the number of messages defined by the PM. Note that a PM often needs to handle configuration or control messages.
4. *int modname_XXX_handler(const struct flowos_pm *, const struct flowos_pmhdr *)* to handle the PM configuration or control message **XXX**.
5. *int modname_register_msghandlers(flowos_pmmsg_ handler *)* to register PM message handlers with FlowOS.

The module thread function *int modname_process(struct flowos_pm *pm)* is the main processing function of a flow processing module. FlowOS passes a pointer to the main data structure **struct flowos_pm** of a FlowOS PM to this function when the PM is attached to the flow. The *pm→head* and *pm→tail* members of this structure delimit the section of the flow currently available for this PM to process. The generic pointer *pm→info* field is to point the PM specific data structure and is used to share data between PM thread and its message handlers. This must be initialized before the thread loop. Then, all the processing is done between the following macros:

```
BEGIN_PROCESS(pm, head, tail);

/* process data */

END_PROCESS;
```

**BEGIN_PROCESS** and **END_PROCESS** macros are defined in *flowos.h* header file, which implement the infinite loop of the thread, handle PM control messages including the thread kill signal, and make the PM sleep when there is no data to process. The parameters *head* and *tail* are stream pointers, which receive copies of *pm→head* and *pm→tail* respectively. One could use the following macro to iterate through all the packets between *head* and *tail* pointers.

```
for_each_packet(head, tail, packet),
```

where *packet* is a pointer to FlowOS packet and points the next packet to be processed. One could also use the follwoing *while* loop to process data between *head* and *tail* pointers.

```
streamp_set(ptr, head); /* set ptr = head */
while(! streamp_is_equal(ptr, tail)){
  /* process data */
  /* to move ptr by 1 byte ahead */
  streamp_inc(ptr);
  /* or to move ptr n bytes ahead */
  streamp_move_next(ptr, n);
}
```

where *ptr* is a stream pointer. FlowOS provides a set of stream manipulating functions similar to C string manipulating functions to simplify writing flow processing modules.

Once a PM is done with a segment of data (one or more bytes), it should use either *flowos_release_data(pm, ptr)* to release all the data bytes up to stream pointer *ptr* or *flowos_release_packet(pm, packet* to release all packets up to the packet pointed to by *packet*.

## 4. PERFORMANCE

In order to test the performance of FlowOS, we have tested it with simple flow processing modules on UCL HEN platform. We have used a linear topology of three PCs (*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. We have used Linux kernel traffic generator (pktgen) to measure the throughput.

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 obtain a throughput of 734Mbps for 64-byte packets and 983Mbps for 1500-byte packets.

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-only PM (that does not modify the content of the flow) and run the same test. We observe that FlowOS with a single read-only PM does not cause any delay to the traffic. In order to see if FlowOS affects the throughput for a read/write PM (that modifies the content of the flow), we define a processing pipeline with a simple network address translator (NAT) and perform the same test. Again, we observe that FlowOS with a single read/write PM does not have noticeable overhead on the traffic. Figure 5 depicts the throughput of FlowOS with a single PM, where the plain red line represents the reference throughput of Linux kernel, the red line with ticks represents the read-only PM and the green line with crosses represent the read/write PM.

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 that 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. Note that concurrent PMs use a **min-heap** of their head pointers to determine which PM is to release data. Surprisingly, FlowOS works without causing any performance overhead in this scenario as well as shown in Figure 6.

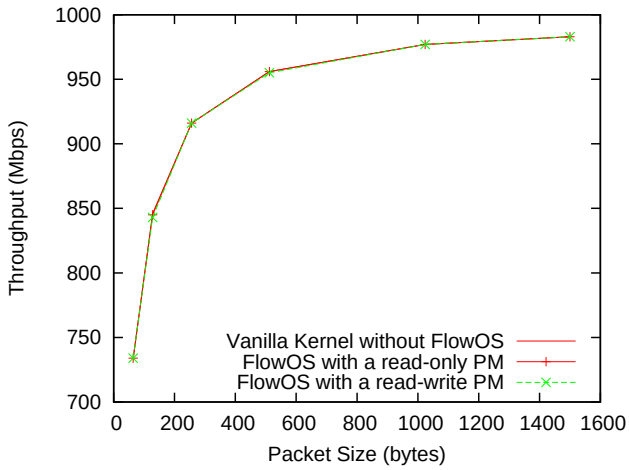


**Figure 5: Throughput of FlowOS with one read-only or read/write processing module.**

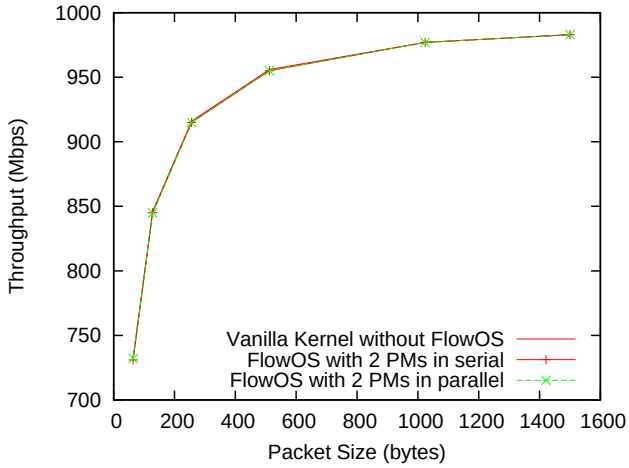


**Figure 6: Throughput of FlowOS with two PMs in sequential and in concurrent settings.**

## 5. RELATED WORKS

Mohamed et. al. [4] proposed a software middlebox platform called ClickOS, which combines the Click [11] moduler 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 [14] hypervisor and can easily be migrated to other hosts.

In [17] authors proposed a software-centric middlebox platform for general-purpose hardware platforms. The main objectives of their proposal include decoupling middlebox application from middlebox hardware, the consolidation of multiple middlebox applications on a shared hardware platform, and providing common APIs for logically centralized middlebox management. They discussed the requirements, challenges, and advantages of such a system without giving actual implementation.

Adam Greenhalgh et. al. [3] proposed a flow processing framework called Flowstream on commodity hardware. Flowstream uses an OpenFlow switch to route flows to commodity servers called module hosts that run middlebox software on virtual machines called processing modules. Authors proposed to use Click [11] modular router software on virtual machines to implement processing modules. A flow can be routed through a sequence of module hosts to be pro-

cessed by a number of processing modules. The switch and module hosts are managed by a platform controller.

Dilip Joseph et. al. [10] presented a simple middlebox model based on RFC 3234 [8], which consists of zones, input preconditions, state databases, processing rules, auxiliary traffic, and the interest and state fields deduced from the processing rules. They showed that it can easily represent many real-world middleboxes and has practical applications.

Zheng et. al. [7] 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 [12] is similar to Maestro that provides a global view of the entire network including the switch-level topology; the locations of users, hosts, middleboxes, and other network elements; 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. [16] 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 trade-offs as dictated by the application requirements. Control applications written within Onix operate on a global view of the network like NOX and use basic state distribution primitives provided by the platform.

# 6. DISCUSSION

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 present 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 stream instead of IP packets such as application gateways.

FlowOS provides a elegant programming model for writing flow processing software. Flows are shared among 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 and runs at line rate. We are developing some application level processing modules to carry out extensive testing.

As future work, we are thinking of 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.

# 7. REFERENCES


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

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

[3] Greenhalgh, A., et.al. Flow processing and the rise of commodity network hardware. *ACM Computer Communication Review 39*, 2 (Nov 2009).

[4] Ahmed, M., et.al. Enabling dynamic network processing with clickos. *ACM SIGCOMM Computer Communication Review 42*, 4 (Oct 2012), 293 – 294.

[5] Greenberg, A, et.al. A Clean Slate 4D Approach to Network Control and Management. *SIGCOMM Computer and Communication Review 35*, 5 (2005), 41–54.

[6] Pfaff, B., et.al. Extending networking into the virtualization layer. In *ACM Workshop on Hot Topics in Networks (HotNets-VIII)* (New York City, NY, Oct 2009).

[7] Cai, Z., et.al. The preliminary design and implementation of the maestro network control platform. Tech. Rep., Rice University, Oct 2008.

[8] Carpenter, B., and Brim, S. Middleboxes: Taxonomy and issues. RFC 3234, Feb 2002.

[9] Lu, G., et.al. Serverswitch: a programmable and high performance platform for data center networks. In *USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, 2011).

[10] Joseph, D., and Stocia, I. Modeling middleboxes. *IEEE Network 22*, 5 (2008), 2 – 25.

[11] Morris, R., et.al. The click moduler router. *ACM Operating Systems Review 34*, 5 (Dec 1999), 217 – 231.

[12] Gude, N., et.al. NOX: Towards an Operating System for Networks. *ACM Computer Communication Review 38*, 3 (Jul 2008).

[13] McKeown, N., et.al. OpenFlow: Enabling Innovation in Campus Networks. *ACM Computer Communication Review 38*, 2 (Mar 2008), 69 – 74.

[14] Barham, P., et.al. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review 37*, 5 (Oct 2003), 164 – 177.

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

[16] Koponen, T., et.al. Onix: A distributed control platform for large-scale production networks. In *Usenix Symposium on OSDI* (Vancouver, BC, Oct 2010).

[17] Sekar, V., et.al. The middlebox manifesto: Enabling innovation in middlebox deployment. In *ACM Workshop on HotNets* (Cambridge, MA, Nov 2011).