In-kernel offloading of an SDN/OpenFlow Controller

Alexander Shalimov
Applied Research Center for
Computer Networks
Lomonosov Moscow State University
ashalimov@arccn.ru

Pavel Ivashchenko
Applied Research Center for
Computer Networks
pivashchenko@arccn.ru

***Abstract*—This paper presents the novel approach on offloading the most time consuming and frequently used functionality of the SDN/OpenFlow controller to the Linux kernel space. This speeds up network applications in 2.5 times together with possibility of using the all userspace libraries and programming tools.**

I. Introduction

SDN/Openflow is already a mainstream in the area of computer networks [1]. It allows us to automate and to simplify network management and administration: fine-grained flows control, observing the entire network, unified open API to write your own network management applications. All control decisions are done first in a centralized controller and then moves down to overseen network's switches. In other words, the controller is a heart of SDN/OpenFlow network and its characteristics determine the performance of the whole network. The controller throughput means how big and active our network can be in terms of switches, hosts, and flows. The response latency directly affects network's congestion time and end-user QoE. Moreover, as faster controller we have as more reactive network we can introduce: faster reaction on host migration and topology changes, more granular flow control, advanced network application like load balancing techniques, security features, and so on.

The recent SDN/OpenFlow controllers performance evaluations show that the throughput of the controllers are not enough for modern datacenters' networks and large scale networks [2], [3], [4]. There are two complimentary ways to cover this performance gap. The first way is to use multiple instances of a controller collaboratively managing the network and forming a distributed control plane. But this way brings a lot of complexity and overheads on maintaining a consistent network view between all instances. The second way is to improve single controller itself by leveraging ability of contemporary multicore systems and by reducing existing bottlenecks and overheads in data communication path in operating systems. Note these two ways can and should be used together to create high efficient distributed control plane.

In this paper, we presents an extended approach on offloading of frequently used SDN/OpenFlow controller functions down to Linux Kernel to create high performance network applications. The paper is structured as follows. Section 2 describes related works and motivation. Section 3 contains the main idea of the proposed approach on in-kernel offloading of an SDN/OpenFlow controller. Section 4 explains implementations details of our in-kernel offload engine. Section 5 shows the result of performance evaluation of the proposed approach.

II. Background

At present, there are a more than 30 different SDN/OpenFlow controllers created by different vendors/universities/research groups, written in different languages (Python, Java, C/C++, Haskell, Erlang, Ruby), using different runtime multithreading techniques, showing different performance numbers [5]. These controllers are implemented as ordinary applications running in Linux userspace.

From the system point of view, implementation in Linux userspace have several performance drawbacks. Every system call (malloc, free, read and write packet(s) from the socket, etc) leads to context switching between userspace and kernel space that requires additional time. Approximately this time for FreeBSD Linux is 0.1ms and takes 10% time for whole system call [6]. Under the high load this leads to significantly time overhead. Moreover, the userspace programs work in virtual memory that also require additional memory translation and isolation mechanism: hierarchical vs linear address translation.

In our previous work [7], to avoid above mentioned overheads we have implemented the OpenFlow controller as a module inside the Linux kernel space. Our experiment evaluation shows that it has 5 times higher performance than all existing controllers. But, as we understood later in practice, it's very hard to write our own application for Linux kernel space. There are several programming challenges: low-level programming language (object C), limited number of libraries and tools, high risk to corrupt the whole system. Thus, we need to find out a way to simplify network applications programming for the in-kernel controller.

III. Proposed approach

As already mentioned, the Linux kernel allows us to significantly speed up the SDN/OpenFlow controllers and provides abilities to create high performance network management applications. The idea is to use kernel space to accelerate the most time consuming functionality of the controller. We call our approach as in-kernel offloading.

There are several important tasks: determine what functionality should be offloaded, what northbound programming API we should provide for a user, and how to implement this command and data passing interfaces between kernel and user space.

Usually a controller consists of three main layers:

- OpenFlow network layer is responsible for communication with OpenFlow switching devices. It imple-

ments TCP server listening new switches connections and OpenFlow library for parsing incoming OpenFlow messages from TCP streams.

- Service layer contains the most frequently used network functions like link discovery, topology, and routing.
- Application layer represents user-written network applications that might use services and subscribe on events from the network layer (for instance, L2 learning switch, firewall, DDOS).

Figure 1 shows the basic scheme of the proposed idea. Figure 1(a) and Figure 1(d) represent two opposite situations where the all layers of the controller reside fully either in the userspace or in the kernel space, correspondingly. In the userpace, the controller has wide range of applications and libraries but low performance. In kernel space, the controller has fastest performance but limited number of applications.

During offloading procedure the controller is gradually been dipping down to the kernel space. The offload scheme supports two operational modes: pass-through mode and driven mode. Both modes describe which functions run inside the kernel. In the pass-through mode, the in-kernel part receives new OpenFlow messages, parses them, and puts into shared queues (figure 1(b)). In the driven mode, the in-kernel part also runs services inside the kernel (figure 1(c)). In this case, it notifies the userspace applications about changes (e.g., topology) and provides an high level interface to manage the network.

IV. Offload Engine

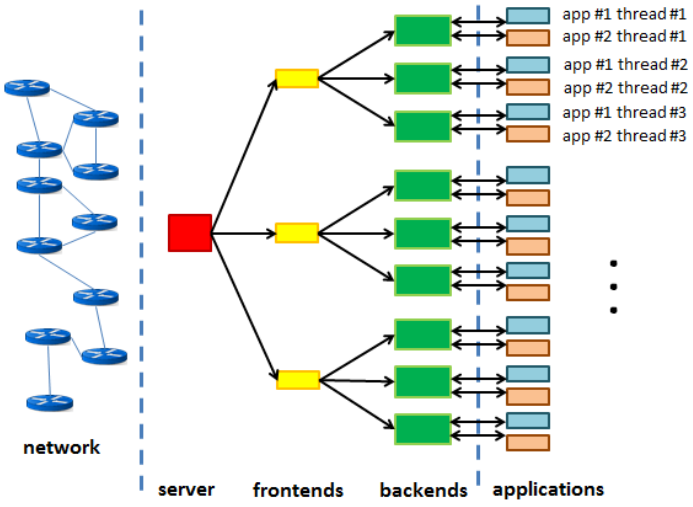

Fig. 2. The SDN/OpenFlow controller offloading architecture.

The figure 2 shows the offload engine architecture. Logically there are three main levels in the offload architecture: in-kernel controller that is responsible for communication with switches, shared data structures that are used to pass information to the userspace, and an userspace network application itself.

The in-kernel has three-tier architecture:

- **Server**. Server thread listens to a socket, accepts new connections from switches and distributes connections between frontend and backend threads.
- **Frontend**. Frontend threads initialize connections and check their correctness: openflow version, hello, features reply. The correctness of headers are checked for every messages in the input buffer until a features reply OpenFlow message will be sent. If all verification is done, connections move to backends.
- **Backend**. Backend threads work with switches and applications. They do the main job on sending and receiving OpenFlow messages. Inside the thread we use poll() to wait for changes in the sockets' descriptors.

Applications running in the userspace communicates with backends through shared data structures. Each backend thread has its own shared data structures. So, to get full speed the userspace application must be multithreaded with the number of threads equal to the number of backends threads in the kernel space, $N\_threads\_app = N\_threads\_kernel$. If $N\_threads\_app < N\_threads\_kernel$ an application will not able to show full power and to process all events coming from the network. If $N\_threads\_app < N\_threads\_kernel$ an application would need to have additional locking mechanism to access to shared backends data structures and thus don't get the full speed either.

Currently multiple applications have to subscribe to different type of OpenFlow messages because we don't store multiple copies of the messages.

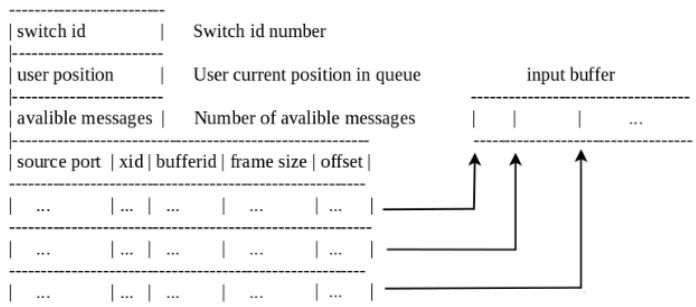

Fig. 3. Packet In queue organization scheme

There are two types of shared data structures in backend threads:

- **Buffer**. All incoming and outgoing raw OpenFlow messages are stored in input and output buffers correspondingly. All buffers are reachable from the userspace through memory mapped regions.
- **Queues**. The data queue is designed to hold pre-parsed OpenFlow messages (see figure 3). For instance, for PacketIn message it holds the following information: source port, xid, bufferid, dpid, ethernet frame (offset and size). An ethernet frame itself resides in input buffer. The control queue is used for communication between kernel and userspace part. In the driven mode, this queue is also used for passing information from the services.

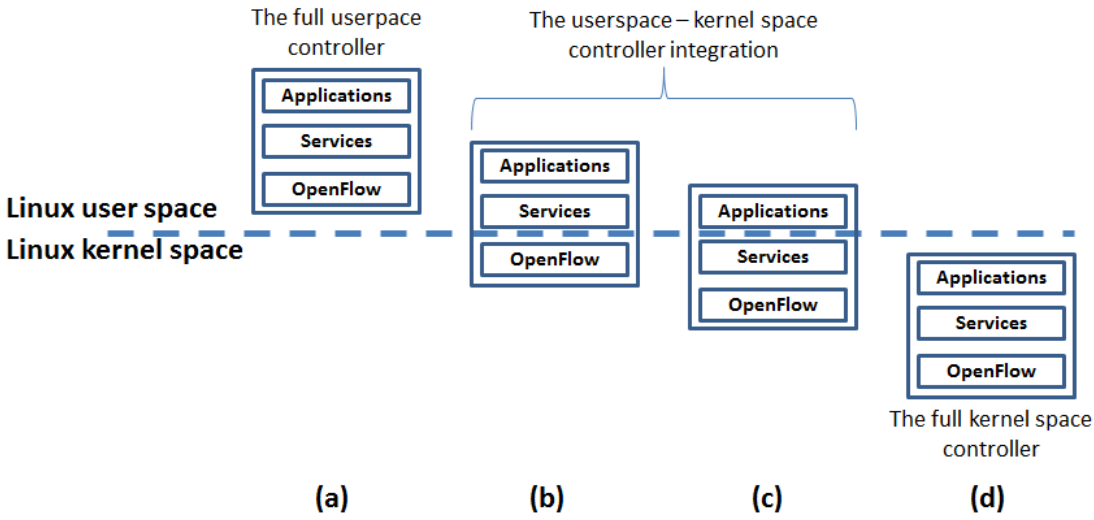

Fig. 1. The basic offload procedure: (a) userspace mode, (b) pass-through mode, (c) driven mode, (d) kernel mode.

From the programming prospective an application's threads open */dev/ctrl* and issue an *ioctl()* to register in controller. Controller queues and OpenFlow packets are in an *mmap()* region with well defined ownership, so that lock free access is possible.

The *poll()* returns the following flags:

- **POLLIN** indicates new events in the data queue (e.g., new packet-in message)and the control queue.
- **POLLRDNORM** means there are events only in the data queue.
- **POLLRDBAND** means there are events only in the control queue.
- **POLLOUT** says all input events have been processed.

A kernel thread reads data from socket and fills buffers, while user application thread reads data from queues and fills output buffer. The kernel thread waits while user application processes all input messages. When the application is done, it calls *write()* function. After that the kernel thread wakes up and finally sends output messages to appropriate switches. Note to speed on application and to decrease network overheads the output message buffer are flushed either by timer in the kernel space or by the application itself. The last option is preferable for fast I/O throughput.

The example below shows the userspace L2 learning switch application that communicated with in-kernel controller through memory API.

```
fds.fd = open("/dev/ctrl", O_RDWR);
fds.events = POLLIN|POLLOUT;
// get memory mapped region size
mem_size = get_memory_size(fds.fd);
// mapping the memory
p = mmap(NULL, size, PROT_READ|PROT_WRITE,
    MAP_SHARED, fds.fd, 0);
// registering the application
app_thread_registration(p->thread_number);
// subscribing to packet-in messages
subscribe_packet_in();
// communicating with in-kernel controller
rx_q = &(p->rx_q);
while (1){
 // reading latest events from the kernel space
 ret = poll(&fds, 1, 2000);
 // nothing to do, wait
 if (ret == 0) continue;
 if (ret > 0){
  // new packet_in messages, process them as
      l2 learning
  if (fds.revents & POLLIN){
   for (; rx_q->avail > 0 ; rx_q->avail--){
    l2(rx_q->id, p->thread_number, rx_q->cur);
       rx_q->cur++;
   }
   continue;
  }
  // the output buffer is full, then send all
      data to switches
  if (fds.revents & POLLOUT){
   write(fds.fd, &p->thread_number,
       sizeof(int));
   continue;
  }
  // kernel space is off
  if (fds.revents & POLLERR)
       error("userpace-kernelspace
           communication failed")
}
```

The driven mode becomes possible when we have implemented pass-through mode and measured that while the userspace thread is 100% loaded, the dedicated kernel thread is only 25% loaded. This observation shows the kernel threads might perform some additional useful functions. This list includes topology discovery, endpoint tracking, dynamic routing, working with some dataplane control protocols like ARP. We add additional type of data and control messages in order tu push changes and information to applications' threads. Applications can send requests through control queues or read push in changes from data queues *[service, type, data]*.

V. Experimental evaluation

Our experimental evaluation consists of two parts. The first part is performance evaluation of pass-through mode of the OpenFlow controller where the goal is to measure I/O overheads on offload engine and kernel/userspace communications based on L2 learning application. The second part is for driven mode based on L3 forwarding application in order to use a topology service.

A. Pass-through mode evaluation

For performance evaluation we use the methodology described in [5]. There we used only one 10Gb channel and on cbench because no one controller was able to process all messages from the channel. In our case, we need two 10Gb channels and two cbench'es. Finally, the test-bed consists of two servers connected with two 10Gb links and two cbench'es generating the packetin messages over these two links.

Figure 4 and Table 1 shows the renewed throughput and latency numbers for the existing controller against the pure in-kernel controller and the in-kernel controller in pass-through mode. The throughput of the pure in-kernel controller is almost 30M flow per second that is 5 times faster than all others. The throughput of the pass-through in-kernel controller is lower with 15M flow per second but it's still 2.5 times faster than others. The latency of the pure in-kernel controller and the pass-through controller are 45us and 50us, respectively.

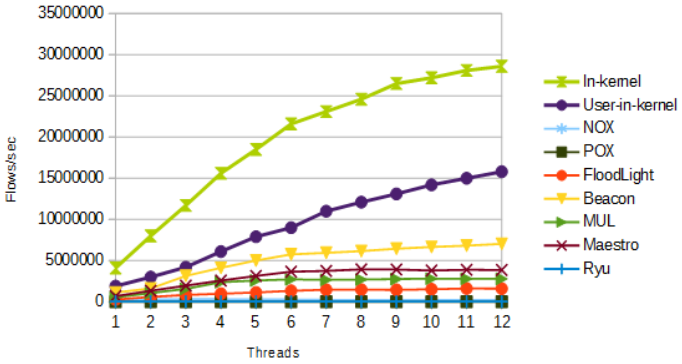

Fig. 4. The average throughput achieved with different number of threads (with 32 switches, $\mathbf{10^5}$ hosts per switch)(Intel(R) Xeon(R) CPU E5645 2.40GHz)

In-Kernel	**45**
Pass-through	**50**
NOX	91
POX	323
Floodlight	75
Beacon	57
MuL	50
Maestro	129
Ryu	105

TABLE I. The minimum response time (10^{-6} secs/ flow)

B. Driven mode evaluation

The controller run L3 forwarding application (calculating the path between two hosts using Dijkstra algorithm) in the userspace and topology discovery services in the kernel space.

We measured the time required for initial topology discovery in the driven mode and the userspace mode. We used mininet to create an OpenFlow network with a tree topology of depth 3 and fanout 3 (i.e 27 hosts, 13 switches, 39 links). It takes 24ms in the userspace mode and 5ms in the driven mode to find out the whole topology. The Beacon controller [8] requires almost 55ms to discover this topology.

We also tried to use the ten physical servers running two instances of Open vSwitch connected with different topologies. The times are slightly less but still different in 4 to 5 times.

Comparing path calculation procedure we measured the 3-4 times difference: 10ms in the userspace mode and 2ms in the driven mode.

VI. Conclusions

Such offloading mechanism accelerates the most time consuming and frequently used parts of the OpenFlow controller using the Linux kernelspace. This allow us to easily create high performance network application. The proposed architecture can be easily extended with other services like verification, link status monitoring, etc. Further work will include the development of new services and simplify API between the kernel space and the userspace.

Our in kernel offloading implementation shows high performance number comparing with existed controllers. The userpace application is still 2.5 times faster with 15M flows per second. Services might be speed on up to 5 times by moving them into the kernel side.

Our approach is the future of previous approaches to inkernel HTTP servers that were only able to return static data to user requests [9].

Acknowledgment

This research is supported by the Skolkovo Foundation Grant N 79, July, 2012 and the Ministry of education and science of the Russian Federation, Unique ID RFMEFI60914X0003.

References

[1] M. Casado, T. Koponen, D. Moon, S. Shenker. *Rethinking Packet Forwarding Hardware*. In Proc. of HotNets, 2008

[2] A. Shalimov, R. Smeliansky, *On Bringing Software Engineering to Computer Networks with Software Defined Networking*, Proceeding of the 7th Spring/Summer Young Researchers' Colloqium on Software Engineering (SYRCoSE 2013), May 30-31, 2013, Kazan, Russia

[3] Advait Dixit, *Towards an Elastic Distributed SDN Controller*, Proceeding of the ACM SIGCOMM HOTSDN 13, Hong Kong.

[4] T. Benson, A. Akella, D. Maltz, *Network traffic characteristics of data centers in the wild*, IMC, 2010

[5] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, R. Smeliansky, *Advanced Study of SDN/OpenFlow controllers*, Proceedings of the CEE-SECR13: Central and Eastern European Software Engineering Conference in Russia, ACM SIGSOFT, October 23-25, 2013, Moscow, Russian Federation

[6] *Netmap*, info.iet.unipi.it/∼luigi/netmap/

[7] P. Ivashchenko, A. Shalimov, R. Smeliansky, *High performance in-kernel SDN/OpenFlow controller*, Proceedings of the 2014 Open Networking Summit Research Track, USENIX, 2014, Santa Clara

[8] David Erickson, *The Beacon OpenFlow Controller*, Proceeding of the ACM SIGCOMM HOTSDN 13, Hong Kong.

[9] kHTTPd - Linux HTTP accelerator, http://www.fenrus.demon.nl/