Improving the performance of Open vSwitch

Marta Carbone, Gaetano Catalli, Luigi Rizzo
Dipartimento di Ingegneria dell'Informazione
Università di Pisa, Italy

May 30, 2011

Abstract

Open vSwitch is a software implementation of a virtual switch, designed to be fully configurable and compatible with the most used protocols. Among other features, the program includes a user space forwarding engine, which can be used to build flexible packet processing systems.

In the process of porting Open vSwitch to FreeBSD, we measured its forwarding performance and found disappointingly low figures, which existed also in the original Linux implementation. As a consequence, we analysed and revised the architecture of some key parts of the code obtaining a speedup of a factor of 10, up to 690 Kpps.

The main contribution of this paper is to illustrate the architecture of the system, its performance bottlenecks, and present how we revised it to achieve huge performance improvements. As a second contribution, we extend the program adding a BPF-compatible driver, enabling operation on BSD systems. This driver is of particular importance because it opens the way to a recently developed network API called netmap, which promises further huge performance improvement.

1 Introduction

Open vSwitch [2] is a software implementation of a modular packet switch, which can be used both as a software switch and as a controller for dedicated switching hardware. Among other features, Open vSwitch contains two implementation of forwarding engines – one running in user space, and one talking to a kernel space forwarding engine. While the original Open vSwitch code runs on Linux, it is designed to be portable to other architectures.

As part of another project related to fast packet processing, we decided to port the code to FreeBSD. This mostly required to write a BPF-compatible driver to send and receive traffic, and this was a first step to adapt the code to our high performance drivers. In the process, we measured the forwarding performance of the system and found surprisingly low values. This motivated an in-depth analysis of the code to find the reasons of such behaviour.

Our search quickly identified a poor design of the main eventloop of the system, and brought us to restructure the code in a way that is minimally intrusive yet provided a ten-fold improvement in the forwarding performance.

In the rest of the paper we will present the architecture of the Open vSwitch code, discuss in detail its performance issues, and present the solution that let us improve its performance.

2 Architecture

In Open vSwitch, a *virtual switch* moves layer-2 packets between two or more network ports. In the software, it is represented by an object called *datapath*, which contains a list of ports used by the virtual switch, and a *flow table* that associates a specific action to each flow. Each packet flowing through the switch is compared against the entries in the flow table, and in case of a match the corresponding action

is executed.

The software architecture of Open vSwitch puts a clear separation between the *controller*, in charge of managing ports and computing the content of the flow tables, and the *datapath*, which does the actual packet switching. The two components can run on different systems, and, as a consequence, the datapath can be implemented with various technology, from software to hardware.

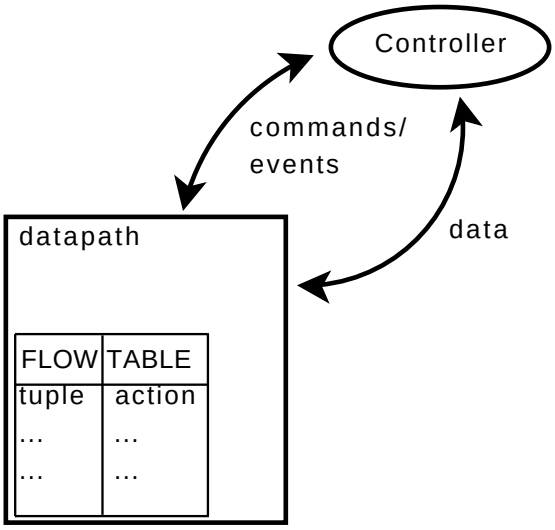

Figure 1: A virtual switch architecture.

The software architecture of OpenvSwitch is represented in Figure 2, which includes two options for the datapath. One, on the left side, indicates a pure user space implementation. On the right side, a kernel implementation.

The original Open vSwitch code is written for Linux in portable C code, though there are obvious system dependencies when it comes to access specific kernel modules, or even APIs to send and receive packets.

The upper layers of the user space code compile clearly on FreeBSD and are:

- *the ofproto library*, implementing the core of the openflow protocol [3];

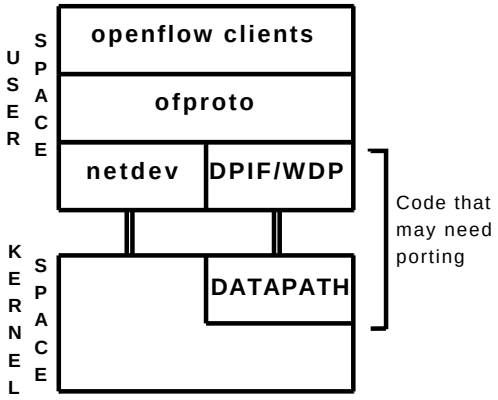

Figure 2: A virtual switch architecture.

- clients using the *ofproto* library (ovs-vswitchd, ovs-openflowd).

The lower layers handle the communication with the network devices provided by the kernel, so they are more platform dependent.

The user space implementation is based on a device abstraction, called *netdev*, that implements the API functions to communicates with the network devices provided by the kernel. The kernel space implementation implements the flow table directly into the kernel, allowing a faster processing of traffic compared to the userspace version.

3 Porting Open vSwitch to FreeBSD

The porting work requires the implementation of a suitable software layer that talks to native FreeBSD APIs (in our case `libpcap` and the routing sockets).

In this work we ignored the kernel module, because of its tight integration with the Linux kernel, and because we have plans for better options [1]. As a consequence, we focused our attention on porting the `ovs-openflowd` client, and on the creation of a new *netdev* object, `netdev-bsd`, that uses the FreeBSD network APIs.

Configuration	Configured ports 2	4	6	8
FreeBSD usersp.+ BPF	65	56	51	47
Linux userspace	50	-	-	-
Linux Kernel	300*	-	-	-
FreeBSD static waiters	74	73	68	67
FreeBSD 2 threads + 1pkt per poll()	380	330	300	270
FreeBSD 2 threads + 50pkt per poll()	690*	690	684	680

Figure 3: ovs-openflowd throughput in Kpps with different configurations.

A detailed description of this port will be given in the full version of the paper.

4 Performance

Once the port has been completed, we measured its forwarding performance to figure out the operating limits of the system.

To test the OpenvSwitch performance we run a set of tests using different software configurations. Testing sessions involve real PCs used as sender/receiver, and packets generators (`netsend` and `pg`) to send minimum-sized packets at different rates. The results are in the first row of Figure 3, and are definitely disappointing, peaking at 65 Kpps and rapidly decreasing as the number of ports grows.

Thinking of bugs in our implementation of the network module, we then compared the values with those on Linux, and we found equally poor performance – only 50 Kpps with userspace forwarding, about 300 Kpps with the kernel forwarding module. As a reference, native FreeBSD bridging on the same hardware runs above 700 Kpps.

The difference between the Linux and the FreeBSD performace was unexpected. Since the FreeBSD port uses the `pcap` library, which involves an additional copy of the packet, we expect this value to be slower if compared with the corresponding Linux result. The explanation of this depends on the interface used to grab packets from the kernel. The Linux implementation gets the packets from the kernel one by one, while in FreeBSD the `libpcap` buffer gets more that one packet for each syscall. Note that in this preliminary version, the `ovs-openflowd` program still process one packet at time.

4.1 Code scrutiny

To explain and correct the poor performance, we started to investigate the Open vSwitch architecture.

The `ovs-openflowd` main loop executes the following steps:

- create a list of objects waiting for packets (*waiters* list);
- call the `poll()` function on the file descriptors associated to the entries in the waiters list;
- surprisingly, destroy the waiters list;
- invoke all the callbacks associated to all objects in the system, irrespective of whether or not the corresponding file descriptor reported as ready.

The inefficiency of this process is obvious – a lot of useful information is thrown away and rebuilt at every iteration. This is especially undesirable as the poll() wakes up at every packet to forward.

Another source of inefficiency is the fact that the waiters list contains at least one element for each configured port in the (software) switch, meaning that performance degrades rapidly with the number of ports.

An initial, simple performance improvement consists in preserving the list of waiters across iterations. This change alone (see the line "static waiters" in the Figure) brings the 2-port performance to 74 Kpps, and degradation with the number of ports is much slower. Still, this is far from being satisfactory.

Short of a major restructuring of the program, our solution to achieved decent throughput is to remove the handling of high-traffic file descriptor from the main loop. In detail, we created a separate thread and event loop which only deals with BPF file descriptors, and invoke only the callbacks for which descriptors are ready. In this system, some packets need to be processed by the main event loop (they are the

initial packets of a flow, which cause flowtable entries to be created). We address this problem using a pipe that connects the two event loops: when one of these packets is received, it is stored into a queue shared by the two threads, and the event is signaled to the pipe so that the main event loop is woken up and can process queued packets (a small subset of the overall traffic).

Despite the extra signalling and locking costs involved, we managed to reach a throughput of ∼690Kpps, which is close to the limit of our traffic generator.

References

[1] NetMapOpen: memory mapping of network devices. http://info.iet.unipi.it/∼luigi/netmap.

[2] Open vSwitch: an Open Virtual Switch. http://openvswitch.org.

[3] OpenFlow. http://www.openflow.org.