Extended Berkeley Packet Filter (eBPF) for Monitoring and Fast Data Path

Team3
Harsh Khetawat, Nida Syed, Niklesh Phabiani, Sudhendu Kumar
{hkhetaw, nsyed, nphabia, skumar23} @ncsu.edu

1. Motivation

There are several network-plugin providers for creating the hypervisor network to connect VMs and containers within the hypervisor or outside the hypervisor. These network plugin providers use different network devices (ovs-bridges, linux-bridges, veth-pairs, tunnel) and provide various services (NAT, DHCP, Tunnelling) for a tenant network system. There is a need for studying the time resource consumed by various network plugins for a given use-case. With the possibility of pipelining various network plugins to get a desired network infrastructure, the study becomes more relevant because of the options and choices that a customer can use.

2. Project objective

a. Understand the capabilities and advantages of eBPF based filtering
b. Use eBPF to evaluate performance of network implementation of different Kubernetes CNI-plugins
    i. We'll start with the base case of measuring the time spent by packet in OVS/linux bridges in different modes [Bridge (L2), Routed (L3), NAT, Tunnel]
c. Implement Fast Data Path/Firewall using eBPF

3. Related work and Literature Review

BPF or the Berkeley Packet Filter was designed for capturing network packets and filtering them based on specific rules[1]. Filters are nothing but programs that are run on a virtual machine. BPF is implemented as user-space programs that are run inside the kernel. However, the original BPF design did not take into account the advances of modern processors, such as the new instructions in multiprocessor systems.
The extended BPF (eBPF) was designed to overcome these limitations of BPF wherein the eBPF virtual machine resembles modern processors, which allows eBPF instructions to be

mapped to the instruction set of the hardware for enhanced performance[2]. This allows just-in-time(JIT) compilation. Additionally, an instruction makes it possible to call in-kernel functions without much cost. Also, the eBPF virtual machine is exposed directly to userspace. eBPF programs can be attached to tracepoints, kprobes, and perf events, which allows for tracing and analysis of network packets and the datapaths they take. There are several engineers who have used eBPF for packet tracing and analysis:

a. **Offloading OVS Flow Processing using eBPF:** William Tu from VMWare has used eBPF to offload OVS [3]. The original OVS data path consists of receiving packets from the bridge/device hook with the OVS kernel datapath module performing the parsing, lookup with the server database, and required actions for each packet that arrives. Tu proposes replacing the entire OVS kernel datapath with eBPF, wherein ovs-vswitchd controls and manages the eBPF datapath. eBPF maps act as communication channels between the ovs-vswitchd and the eBPF datapath. In order to achieve this, Tu proceeds with parsing the header from P4, which is the OVS specification to an eBPF compiler specification from bcc. The flow table lookup involves the perf ring buffer which carries the packet and its metadata to the ovs-vswitchd which does flow translation, and program flow entry into the eBPF map. This allows actions such as flooding, mirror and push vlan, and tunnel, and the performance evaluation in terms of the number of packets sent is better than when compared to the OVS datapath.

b. **XDP to implement load balancers, firewalls and other networking services (used in Cilium and other projects):** When it comes to high performance processing, XDP (eXpress Data Path) and TC (Traffic Control) are two hooks that are available. XDP provides a mechanism to run eBPF programs at the lowest level of the Linux networking stack, directly upon the receipt of the packet and immediately out of driver receive queues[5]. XDP filters packets as soon as they are received using an eBPF program. This returns an action, which can be used to define the context, etc.
   While the kernel is in charge of moving packets quickly, BPF dictates the logic which decides action with respect to reading and/or writing the packet. BPF provides the programming for XDP to allow users to access the low level networking data path to implement virtual switches, load balancers, firewalls, etc. XDP programs are also portable across different XDP platforms which includes not only Linux kernel but potentially NIC offloads, switches, DPDK, and other OSes.[7]
   Another technology initiated by Cisco that relies on BPF and XDP is Cilium. Cilium is an open source software that aims to provide network connectivity and load balancing functionality between workloads like containers[13]. At the core of Cilium is eBPF which provides security and networking capabilities by the dynamic insertion of eBPF bytecode into the Linux kernel at various hooks or tracepoints. Some of the capabilities include core data path filtering, mangling, monitoring, and redirection. Cilium also uses XDP which enables special eBPF programs to run from the network driver with direct access to the packet's DMA buffer. This allows programs to be attached at the earliest possible point in the software stack allowing for a programmable, high-performance packet processing in

the Linux kernel networking data path. Cilium also supports distributed load balancing for traffic between application containers and to external services. The load balancing is implemented using BPF through efficient hash tables allowing for scale, and supports direct server return if the load balancing operation is not performed on the source host.
Chen and Lu proposed a CETH(Common Ethernet Driver Framework) for XDP(eXpress Data Path) to overcome the kernel network performance for virtualization [9]. The simplified CETH for XDP led to efficient memory and buffer management, customizable metadata structure, and compatibility with the kernel IP stack.

c. **OVN:** Another great work related to eBPF is the IOVisor project. IOVisor is a community driven open source project that opened up new ways to innovate, develop and share I/O and networking functions. It allows creation of IOModules that can be used to build networking, security and tracing applications. It even provides a programmable data plane and development tools to simplify the creation and sharing of dynamic IOModules.
Risso, Bertrone, and Bernal proposed combining IOVisor with the OVN(Open Virtual Network) architecture[4]. IOModule is an eBPF program that performs a specific task such as bridging, routing, NATing, etc. IO Modules can be combined to create complex services as in a service chain. The proposal was to keep OVN control plane and remove the OvS data plane, wherein this new IOVisor-OVN architecture reads information from the existing databases such as ovs-localDB(s), maps changes such as service requests into IOModules, and exploits Hover (REST API front-end) to inject, configure and bind IOModules to network interfaces[4].

d. **Creation of complex network services with eBPF:**
Sebastiano Miano et al.[8] exploited eBPF to create complex network functions, by considering the limitation of the size of the eBPF program, which is restricted to 4096 assembly instructions to guarantee that any program will terminate within a bounded amount of time. This restriction may be limiting when creating network functions that perform complex actions in the data plane. A learning for them because of this limitation was to partition the network function into multiple eBPF programs and jumping from one to another through tail calls. This technique enables the creations of network services as a collection of loosely coupled modules, each one implementing a different function. Also, network functions may have to put the current frame on hold while waiting for some other event to happen. An example is a router holding the packet while its ARP request is getting served. eBPF does not have the capability of holding the packet, thereby not allowing it to take ownership of the packet. A remedy suggested for this is the usage of a slow path module, executed in the userspace, that receives packets from the eBPF program and reacts consequently with respect to the processing defined by the developer. The necessity of such a slow path module is also highlighted in [9], wherein the authors use the OvS userspace module to process packets that do not match a flow in the OvS kernel eBPF data path.

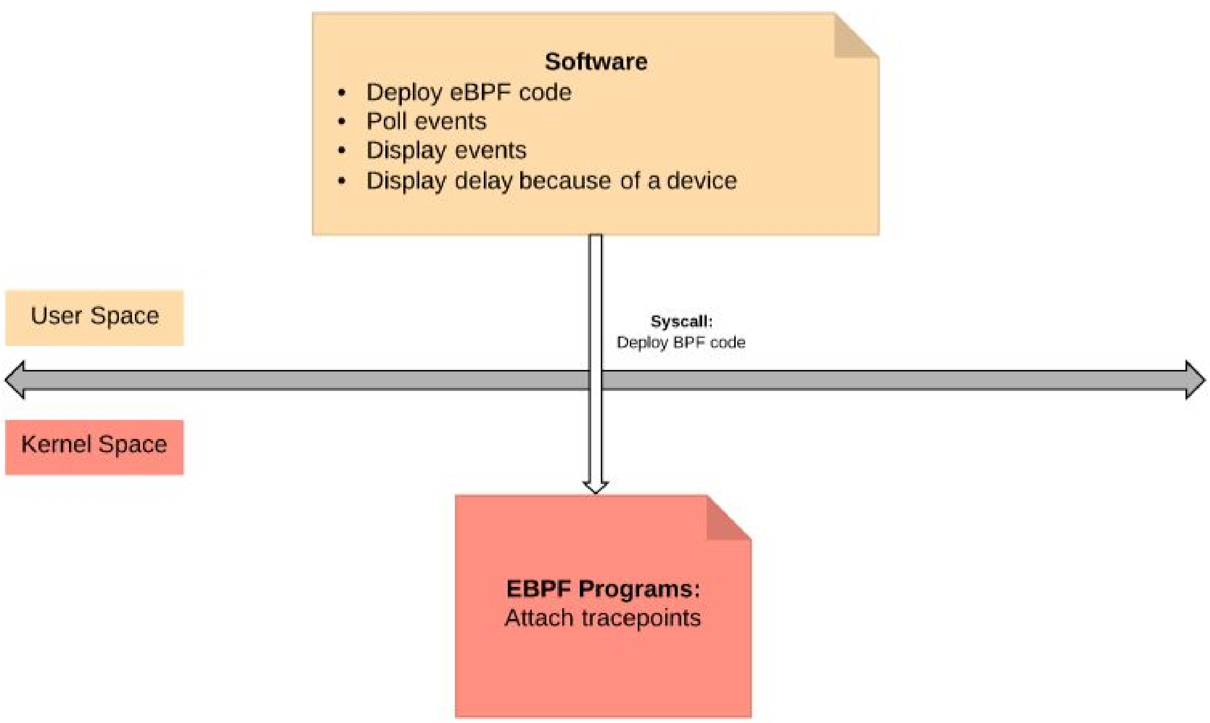
e. **Flow sampling, monitoring:** Yunsong Lu implemented monitoring the whole virtual network stack from socket to virtual switch to physical NIC by making use of existing hooks which can be extended with eBPF and kprobes attached[10]. By the use of Canal View (a container networking framework from Huawei), which is the topology-based virtual networking monitoring system, monitoring of Application-to-Application network traffic was done. This was implemented by making use of NFV(Network Functions Virtualization), Cloud Native, IOVisor, and E2. IOVisor was used as the MDPC (Micro Data Plane Container) with the applications developed and deployed with a Docker-like mechanism. This resulted in a faster network data plane in the kernel.
Now, when it comes to the usage of eBPF in Kubernetes, [16] explains how existing products (Cilium, Weave Scope) leverage eBPF to work with Kubernetes. In specific, it describes what interactions with eBPF are interesting in the context of container deployment.

4. Architecture, Functional Specs and Data Structures

The scope of this project is in two folds:

1. A1: Creating a performance measuring tool
2. A2: Creating a fast path tool in a tenant environment

A1: The tool consists of two softwares. User space software and kernel space EBPF module.



The high level functions of user space software and kernel module is self explanatory from the figure above. The events that the userspace software captures are:

```
# Routing information
("ifname",  ct.c_char * IFNAMSIZ),     #Interface Name
("netns",   ct.c_ulonglong),           #Network Namespace
("func_name", ct.c_char * 100),        #Kernel Function Name
("ts", ct.c_ulonglong),                #Time stamp

# Packet type (IPv4 or IPv6) and address
("ip_version",  ct.c_ulonglong),       #IP version
("icmptype",    ct.c_ulonglong),       #ICMP Type
("icmpid",      ct.c_ulonglong),       #ICMP ID
("icmpseq",     ct.c_ulonglong),       #ICMP SEQ
("saddr",       ct.c_ulonglong * 2),   #Source Address
("daddr",       ct.c_ulonglong * 2),   #Destination Address
```

The userspace code determines the delay incurred by a device by calculating the time differences in the order of nanoseconds.

The BPF Kernel module registers events on the following kernel function:
"Netif_rx"
"Net_dev_queue"
"Napi_gro_receive_entry"
"Netif_receive_skb_entry"
"Net_dev_start_xmit"
"Net_dev_xmit"

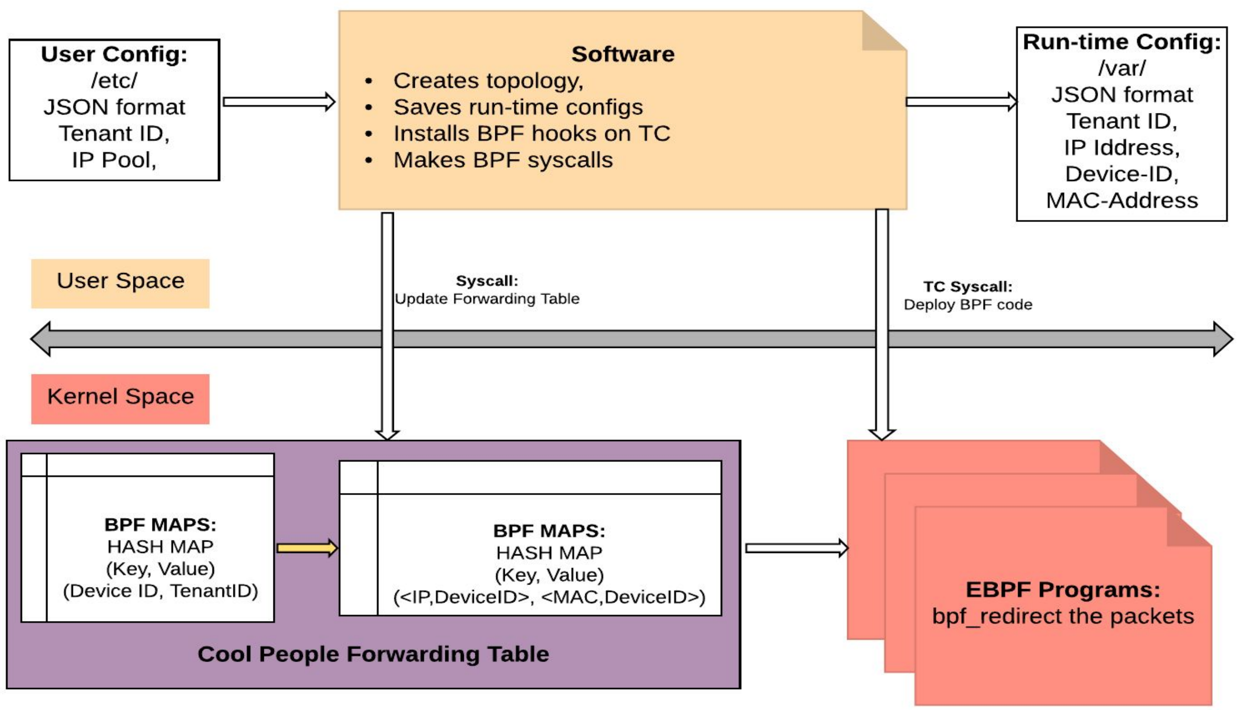
and saves the events in following structure:

```
/* Routing information */
char ifname[IFNAMSIZ];  // To store interface name
u64 netns;              // To store namespace ID
char func_name[100];    // To store networking kernel function name
u64 ts;                 // To store timestamp

/* Packet type (IPv4 or IPv6) and address */
u64 ip_version;         // familiy (IPv4 or IPv6)
u64 icmptype;           // To store ICMP packet type
u64 icmpid;             // In practice, this is the PID of the ping process
u64 icmpseq;            // Sequence number
u64 saddr[2];           // Source address. IPv4: store in saddr[0]
u64 daddr[2];           // Dest   address. IPv4: store in daddr[0]
```

A2: The tool consists of two softwares. One that runs in user space, the other in kernel space. The functions of user space and kernel space programs are self-explanatory. User Config is saved in /etc/configuration.json

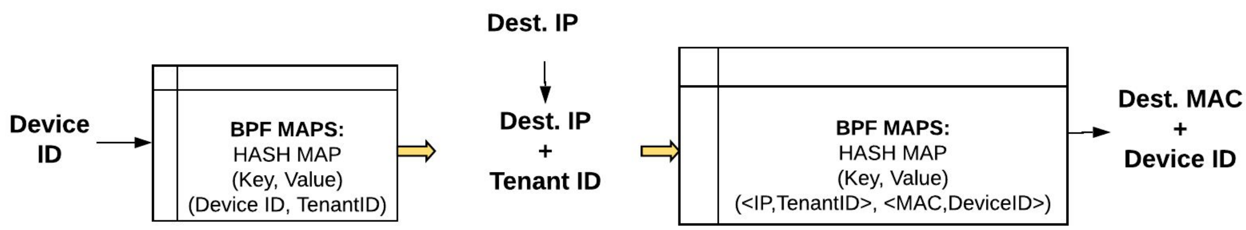
```
{
	"tenant1": [
			{"name": "C1", "IP Subnet": "10.0.10.0/24"},
			{"name": "C2", "IP Subnet": "10.0.11.0/24"}
		   ]
}
```



The user space software creates run-time config in /var/runtimeConfiguration.json

```
{
	"tenant1": [
		{"Device ID": "veth-br40-C1", "IP Address": "10.0.40.2\n", "name": "Container1",
	"MAC Address": "5e:52:1f:cb:5d:78\n"},
		{"Device ID": "veth-br140-C2", "IP Address": "10.0.140.3\n", "name":
	"Container2", "MAC Address": "b2:7c:3a:88:f7:6f\n"}
		]
}
```

Once the run-time configuration is made user-space software populates the forwarding tables by making BPF syscalls. The forwarding table and the operation looks like below.



Data Structure of Auxiliary FT:
The EBPF kernel code creates two Aux FT in /sys/fs/bpf/tc/globals/ directory.
Table 1: egress_ifindex: Keeps mapping of [Dest. IP + Tenant ID] and [MAC + Device ID]
Table 2: deviceid_tenant: Keeps mapping of the device and Tenant ID.

The structure is shown below.

```
struct map_key{
        __u32 destination_ip;
        __u32 tenant_id;
};
struct map_entry{
        __u32 device_id;
        __u8 dst_mac[6];
};
struct bpf_elf_map SEC("maps") egress_ifindex = {
        .type = BPF_MAP_TYPE_HASH,
        .size_key = sizeof(struct map_key),
        .size_value = sizeof(struct map_entry),
        .pinning = PIN_GLOBAL_NS,
        .max_elem = 256,
};
struct bpf_elf_map SEC("maps") deviceid_tenant = {
        .type = BPF_MAP_TYPE_HASH,
        .size_key = sizeof(__u32),
        .size_value = sizeof(__u32),
        .pinning = PIN_GLOBAL_NS,
        .max_elem = 256,
};
```

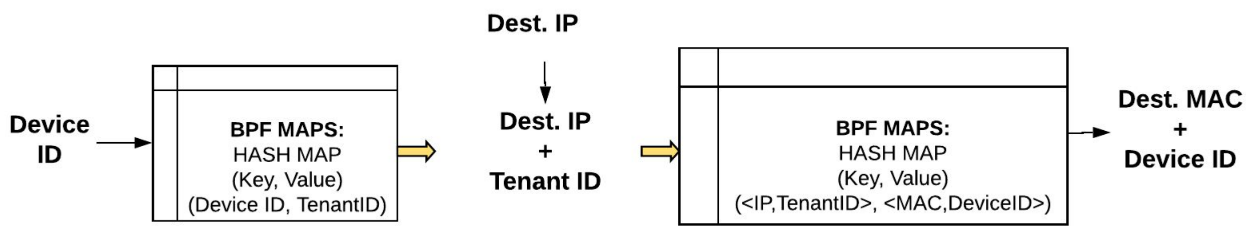
The user space program is responsible for updating these Aux FT:
The user space program accesses the FT created by kernel space program

```
static const char *mapfile = "/sys/fs/bpf/tc/globals/egress_ifindex";
static const char *devtenfile = "/sys/fs/bpf/tc/globals/deviceid_tenant";
        int fd_egress = bpf_obj_get(mapfile);
        int fd_tenant = bpf_obj_get(devtenfile);
```

And updates the FT by making BPF syscalls from user space by reading the values from run time configuration file.

```
        ret = bpf_map_update_elem(fd_tenant, &device_id, &tenant_id, 0);
        ret = bpf_map_update_elem(fd_egress, &key, &value, 0);
```

Data path operation:



When the packet comes in the ingress TC of the source container, the EBPF kernel code looks up the table to find the tenant ID. Interface ID is present in the sk_buff data structure.

```
        __u32 iface = skb->ifindex;
        __u32 *tenantId = bpf_map_lookup_elem(&deviceid_tenant, &iface);
```

On getting the tenant ID, another lookup is made into find the destination device and corrosponding MAC address of the destination container.

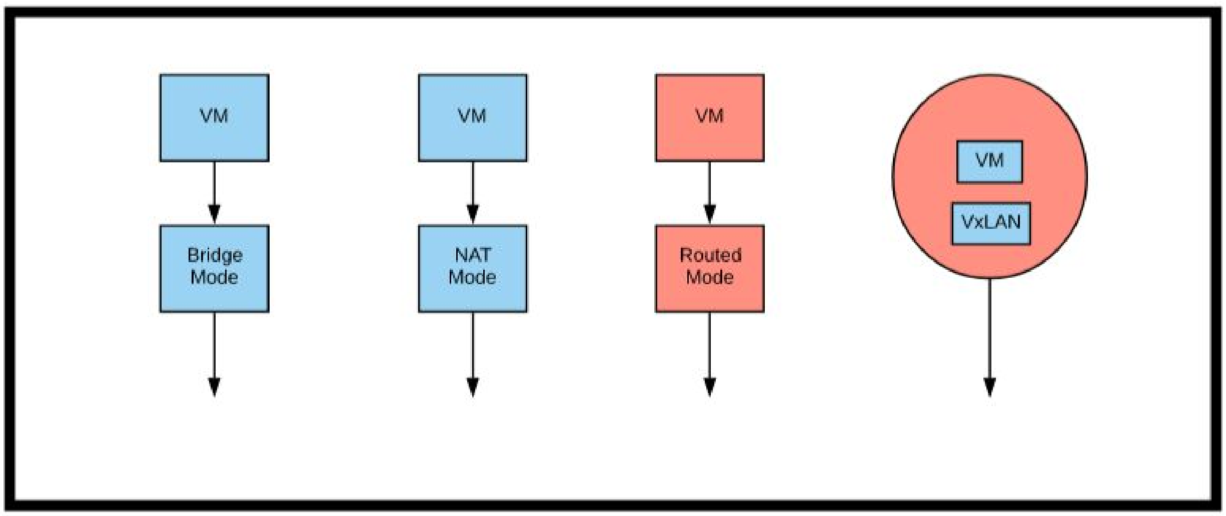
```
struct map_key {
        __u32 destination_ip;
        __u32 tenant_id;
};
struct map_entry {
        __u32 device_id;
        __u8 dst_mac[6];
};
struct map_entry *found_entry = bpf_map_lookup_elem(&egress_ifindex, &key);
```

Once the entry is found, the headers are changed and the packet is redirected to destination device ID.

```
bpf_skb_store_bytes(skb, offsetof(struct ethhdr, h_dest), dst_mac, ETH_ALEN,
BPF_F_RECOMPUTE_CSUM);
bpf_redirect(found_entry->device_id, 0);
```

5. Results

a) Monitoring



Base Scenario: Evaluating the time spent by packet in each device

We have considered four topologies (Bridge mode, NAT mode, Routed mode and VXLAN tunnel mode). We have taken 10 batches of readings with each batch comprising of 100 iterations, and the average is calculated for each batch. This is done for each of the aforementioned topologies.

Input: Ping from VM to 8.8.8.8. For VxLAN device, Ping was done to the VM in another namespace (having the destination TEP).

Results:
The timestamp seen in the snapshots are in nanoseconds.

- **Case I (Bridge Mode):**

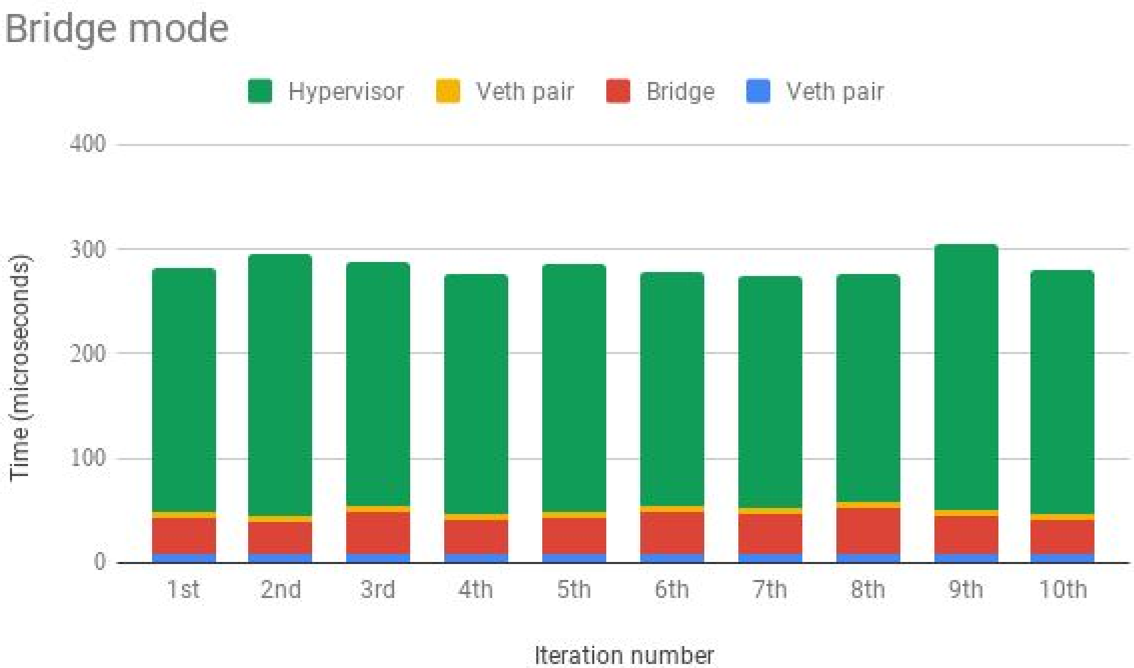
```
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/EBPF$ sudo python tracepkt.py
    NETWORK NS        INTERFACE    TYPE ADDRESSES                  FUNCTION NAME            TIMESTAMP
[  4026532201]       brovsns-ns request 192.168.0.10 -> 8.8.8.8 net_dev_queue   3371013232722767
[  4026532201]       brovsns-ns request 192.168.0.10 -> 8.8.8.8 net_dev_start_xmit      3371013232739754
[  4026531957]       brovsns-br request 192.168.0.10 -> 8.8.8.8 netif_rx        3371013232744839
[  4026531957]       brovsns-br request 192.168.0.10 -> 8.8.8.8 net_dev_xmit    3371013232748073
[  4026531957]           ovs-br request 192.168.0.10 -> 8.8.8.8 netif_rx        3371013233032755
[  4026531957]             ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_queue        3371013233064420
[  4026531957]             ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_start_xmit   3371013233071002
[  4026531957]             ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_xmit         3371013233184599
[  4026531957]             ens3   reply 8.8.8.8 -> 192.168.122.208 napi_gro_receive_entry       3371013241910688
[  4026531957]           ovs-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_queue   3371013241938463
[  4026531957]           ovs-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_start_xmit      3371013241942812
[  4026531957]           ovs-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_xmit    3371013241959964
[  4026531957]       brovsns-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_queue   3371013242065221
[  4026531957]       brovsns-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_start_xmit      3371013242070855
[  4026532201]       brovsns-ns   reply 8.8.8.8 -> 192.168.0.10 netif_rx        3371013242074751
[  4026532201]       brovsns-ns   reply 8.8.8.8 -> 192.168.0.10 net_dev_xmit    3371013242077791
```

brovsns-ns and brovsns-br are the endpoints of the interface between VM and Bridge, and Ovs-br is the bridge.

Interface/ Device	*1st*	*2nd*	*3rd*	*4th*	*5th*	*6th*	*7th*	*8th*	*9th*	*10th*
Veth pair	7.731	7.915	7.881	7.688	8.511	8.118	7.635	8.275	8.470	7.892
Bridge	34.940	31.88	41.34	33.69	34.34	40.85	38.52	44.56	37.31	33.58
Veth pair	5.211	4.988	4.816	4.944	5.172	5.466	5.461	5.120	5.012	4.977
Hypervisor	234.80	250.63	234.34	230.45	238.65	224.28	222.53	218.80	253.41	232.695

All times are in microseconds

Bridge mode



Maximum time is taken in the hypervisor. From our understanding, in addition to NATting, the hypervisor will even have to go through a much longer chain of iptables rules. In all the other cases too, maximum time is taken by the hypervisor.

```
[   4026531957]       brovs-host   reply 8.8.8.8 -> 192.168.0.10 net_dev_queue   6898755419777435
[   4026531957]       brovs-host   reply 8.8.8.8 -> 192.168.0.10 net_dev_start_xmit      6898755419785467
[   4026531957]         brovs-br   reply 8.8.8.8 -> 192.168.0.10 netif_rx        6898755419791224
[   4026531957]         brovs-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_xmit    6898755419795923
[   4026531957]       brovsns-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_queue   6898755419847863
[   4026531957]       brovsns-br   reply 8.8.8.8 -> 192.168.0.10 net_dev_start_xmit      6898755419853525
[   4026532607]       brovsns-ns   reply 8.8.8.8 -> 192.168.0.10 netif_rx        6898755419858039
[   4026532607]       brovsns-ns   reply 8.8.8.8 -> 192.168.0.10 net_dev_xmit    6898755419861876
[   4026532607]       brovsns-ns request 192.168.0.10 -> 8.8.8.8 net_dev_queue   6898756412242516
[   4026532607]       brovsns-ns request 192.168.0.10 -> 8.8.8.8 net_dev_start_xmit      6898756412274828
[   4026531957]       brovsns-br request 192.168.0.10 -> 8.8.8.8 netif_rx        6898756412283188
[   4026531957]       brovsns-br request 192.168.0.10 -> 8.8.8.8 net_dev_xmit    6898756412288398
[   4026531957]         brovs-br request 192.168.0.10 -> 8.8.8.8 net_dev_queue   6898756412308302
[   4026531957]         brovs-br request 192.168.0.10 -> 8.8.8.8 net_dev_start_xmit      6898756412313647
[   4026531957]       brovs-host request 192.168.0.10 -> 8.8.8.8 netif_rx        6898756412318677
[   4026531957]       brovs-host request 192.168.0.10 -> 8.8.8.8 net_dev_xmit    6898756412322697
[   4026531957]             ens3 request 192.168.122.175 -> 8.8.8.8 net_dev_queue        6898756412360661
[   4026531957]             ens3 request 192.168.122.175 -> 8.8.8.8 net_dev_start_xmit   6898756412371897
[   4026531957]             ens3 request 192.168.122.175 -> 8.8.8.8 net_dev_xmit         6898756412564663
[   4026531957]             ens3   reply 8.8.8.8 -> 192.168.122.175 napi_gro_receive_entry       6898756421163838
('veth pair: ', 8360L)
('Device: ', 30459L)
('veth pair: ', 5030L)
('Device: ', 245986L)
('Average with key ', 'brovsns-nsbrovsns-br', 7892L)
('Average with key ', 'brovsns-brbrovs-br', 33579L)
('Average with key ', 'brovs-brbrovs-host', 4977L)
('Average with key ', 'brovs-hostens3', 232695L)
```

Screenshot of the last iteration of a batch

- **<u>Case II (NAT Mode):</u>**

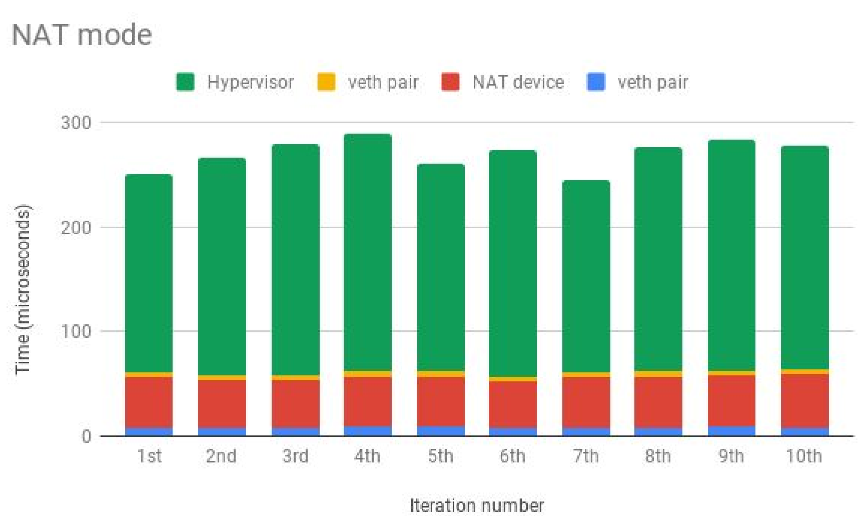
```
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/EBPF$ sudo python tracepkt.py
      NETWORK NS         INTERFACE     TYPE ADDRESSES                    FUNCTION NAME             TIMESTAMP
[  4026532433]       natovsns-ns request 192.168.1.10 -> 8.8.8.8 net_dev_queue   3371369156944237
[  4026532433]       natovsns-ns request 192.168.1.10 -> 8.8.8.8 net_dev_start_xmit       3371369156957590
[  4026531957]      natovsns-nat request 192.168.1.10 -> 8.8.8.8 netif_rx        3371369156962586
[  4026531957]      natovsns-nat request 192.168.1.10 -> 8.8.8.8 net_dev_xmit    3371369156965838
[  4026531957]           ovs-nat request 192.168.1.10 -> 8.8.8.8 netif_rx        3371369157109016
[  4026531957]              ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_queue        3371369157145166
[  4026531957]              ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_start_xmit   3371369157151420
[  4026531957]              ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_xmit         3371369157256810
[  4026531957]              ens3    reply 8.8.8.8 -> 192.168.122.208 napi_gro_receive_entry        3371369165970247
[  4026531957]           ovs-nat    reply 8.8.8.8 -> 192.168.1.10 net_dev_queue   3371369166002202
[  4026531957]           ovs-nat    reply 8.8.8.8 -> 192.168.1.10 net_dev_start_xmit       3371369166007103
[  4026531957]           ovs-nat    reply 8.8.8.8 -> 192.168.1.10 net_dev_xmit    3371369166023908
[  4026531957]      natovsns-nat    reply 8.8.8.8 -> 192.168.1.10 net_dev_queue   3371369166124967
[  4026531957]      natovsns-nat    reply 8.8.8.8 -> 192.168.1.10 net_dev_start_xmit       3371369166130417
[  4026532433]       natovsns-ns    reply 8.8.8.8 -> 192.168.1.10 netif_rx        3371369166133785
[  4026532433]       natovsns-ns    reply 8.8.8.8 -> 192.168.1.10 net_dev_xmit    3371369166136777
```

natovsns-ns and natovsns-br are the endpoints of the interface between VM and Bridge, and Ovs-br is the bridge.

Interface / Device	*1st*	*2nd*	*3rd*	*4th*	*5ht*	*6th*	*7th*	*8th*	*9th*	*10th*
veth pair	8.224	8.273	8.162	8.750	9.041	7.932	8.392	8.516	8.731	8.046
NAT device	48.1	45.61	45.59	48.40	47.80	44.15	48.46	48.38	49.48	51.08
veth pair	4.884	4.679	4.714	4.889	5.143	4.531	4.850	5.012	4.801	5.042
Hypervisor	190.16	208.22	221.63	226.92	198.54	216.84	183.31	215.09	221.49	213.88

All times are in microseconds

NAT mode



- **Case III (Routed Mode):**

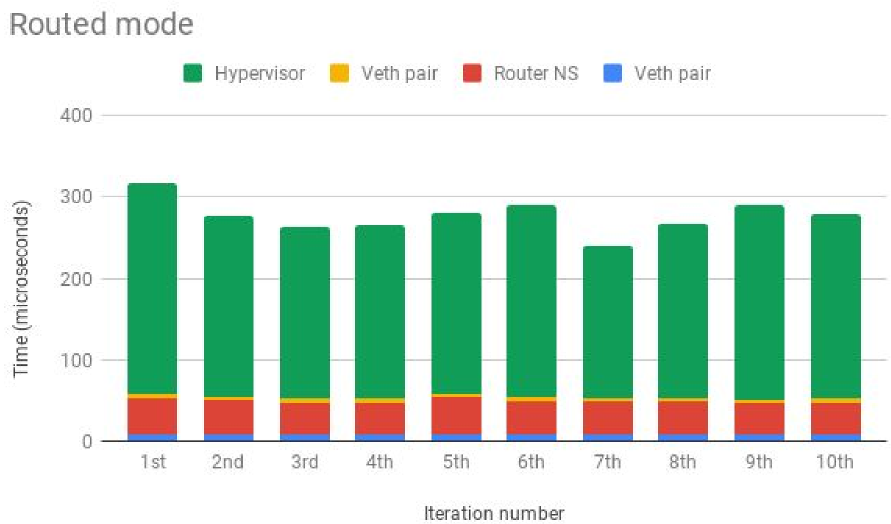
```
ece792@ece792-Standard-PC-i440FX-PIIX-1996:~/EBPF$ sudo python tracepkt.py
    NETWORK NS        INTERFACE    TYPE ADDRESSES                 FUNCTION NAME            TIMESTAMP
[  4026532493]       rovsns-ns1 request 192.168.2.10 -> 8.8.8.8 net_dev_queue   3375143740674908
[  4026532493]       rovsns-ns1 request 192.168.2.10 -> 8.8.8.8 net_dev_start_xmit      3375143740693508
[  4026532625]        rovsns-r1 request 192.168.2.10 -> 8.8.8.8 netif_rx        3375143740698862
[  4026532625]        rovsns-r1 request 192.168.2.10 -> 8.8.8.8 net_dev_xmit    3375143740702057
[  4026532625]           rovs-1 request 192.168.2.10 -> 8.8.8.8 net_dev_queue   3375143740718259
[  4026532625]           rovs-1 request 192.168.2.10 -> 8.8.8.8 net_dev_start_xmit      3375143740721681
[  4026531957]           rovs-2 request 192.168.2.10 -> 8.8.8.8 netif_rx        3375143740724493
[  4026531957]           rovs-2 request 192.168.2.10 -> 8.8.8.8 net_dev_xmit    3375143740727030
[  4026531957]             ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_queue        3375143740745718
[  4026531957]             ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_start_xmit   3375143740751022
[  4026531957]             ens3 request 192.168.122.208 -> 8.8.8.8 net_dev_xmit         3375143740854063
[  4026531957]             ens3   reply 8.8.8.8 -> 192.168.122.208 napi_gro_receive_entry       3375143749578240
[  4026531957]           rovs-2   reply 8.8.8.8 -> 192.168.2.10 net_dev_queue   3375143749606886
[  4026531957]           rovs-2   reply 8.8.8.8 -> 192.168.2.10 net_dev_start_xmit      3375143749611469
[  4026532625]           rovs-1   reply 8.8.8.8 -> 192.168.2.10 netif_rx        3375143749615123
[  4026532625]           rovs-1   reply 8.8.8.8 -> 192.168.2.10 net_dev_xmit    3375143749618117
[  4026532625]        rovsns-r1   reply 8.8.8.8 -> 192.168.2.10 net_dev_queue   3375143749645962
[  4026532625]        rovsns-r1   reply 8.8.8.8 -> 192.168.2.10 net_dev_start_xmit      3375143749649992
[  4026532493]       rovsns-ns1   reply 8.8.8.8 -> 192.168.2.10 netif_rx        3375143749652925
[  4026532493]       rovsns-ns1   reply 8.8.8.8 -> 192.168.2.10 net_dev_xmit    3375143749655796
```

rovsns-ns and rovsns-br are the endpoints of the interface between VM and Bridge, and rovs-1 and rovs-2 is the interface of hanging on the hypervisor.

Interface / Device	*1st*	*2nd*	*3rd*	*4th*	*5th*	*6th*	*7th*	*8th*	*9th*	*10th*
Veth pair	9.241	8.853	7.872	8.734	8.328	8.645	8.061	8.515	8.217	8.085
Router NS	43.73	41.29	39.05	39.07	45.40	40.79	40.42	39.72	38.31	39.15
Veth pair	5.171	5.128	4.865	5.062	5.335	5.127	4.928	4.967	4.750	4.937
Hypervisor	258.93	221.63	211.06	213.11	221.21	235.97	187.22	213.92	238.77	226.35

All times are in microseconds

Routed mode




```
[  4026532955]  v-rovs-routerns request 192.168.2.10 -> 8.8.8.8 net_dev_queue   6901279492330387
[  4026532955]  v-rovs-routerns request 192.168.2.10 -> 8.8.8.8 net_dev_start_xmit      6901279492354280
[  4026533108]  v-routerns-rovs request 192.168.2.10 -> 8.8.8.8 netif_rx        6901279492362720
[  4026533108]  v-routerns-rovs request 192.168.2.10 -> 8.8.8.8 net_dev_xmit    6901279492368553
[  4026533108]  v-routerns-host request 192.168.2.10 -> 8.8.8.8 net_dev_queue   6901279492395497
[  4026533108]  v-routerns-host request 192.168.2.10 -> 8.8.8.8 net_dev_start_xmit      6901279492400907
[  4026531957]  v-host-routerns request 192.168.2.10 -> 8.8.8.8 netif_rx        6901279492405979
[  4026531957]  v-host-routerns request 192.168.2.10 -> 8.8.8.8 net_dev_xmit    6901279492410536
[  4026531957]             ens3 request 192.168.122.175 -> 8.8.8.8 net_dev_queue        6901279492439692
[  4026531957]             ens3 request 192.168.122.175 -> 8.8.8.8 net_dev_start_xmit   6901279492452559
[  4026531957]             ens3 request 192.168.122.175 -> 8.8.8.8 net_dev_xmit         6901279492592795
[  4026531957]             ens3   reply 8.8.8.8 -> 192.168.122.175 napi_gro_receive_entry       6901279501209612
('veth pair: ', 8440L)
('Device: ', 38187L)
('veth pair: ', 5072L)
('Device: ', 186816L)
('Average with key ', 'v-rovs-routernsv-routerns-rovs', 8085L)
('Average with key ', 'v-routerns-rovsv-routerns-host', 39154L)
('Average with key ', 'v-routerns-hostv-host-routerns', 4937L)
('Average with key ', 'v-host-routernsens3', 226353L)
[  4026531957]  v-host-routerns   reply 8.8.8.8 -> 192.168.2.10 net_dev_queue   6901279501268812
[  4026531957]  v-host-routerns   reply 8.8.8.8 -> 192.168.2.10 net_dev_start_xmit      6901279501277232
[  4026533108]  v-routerns-host   reply 8.8.8.8 -> 192.168.2.10 netif_rx        6901279501283592
[  4026533108]  v-routerns-host   reply 8.8.8.8 -> 192.168.2.10 net_dev_xmit    6901279501289381
[  4026533108]  v-routerns-rovs   reply 8.8.8.8 -> 192.168.2.10 net_dev_queue   6901279502652424
[  4026533108]  v-routerns-rovs   reply 8.8.8.8 -> 192.168.2.10 net_dev_start_xmit      6901279502667861
[  4026532955]  v-rovs-routerns   reply 8.8.8.8 -> 192.168.2.10 netif_rx        6901279502674314
[  4026532955]  v-rovs-routerns   reply 8.8.8.8 -> 192.168.2.10 net_dev_xmit    6901279502679297
```

Screenshot of the last iteration in a batch

- **Case IV (Tunnel Mode):**

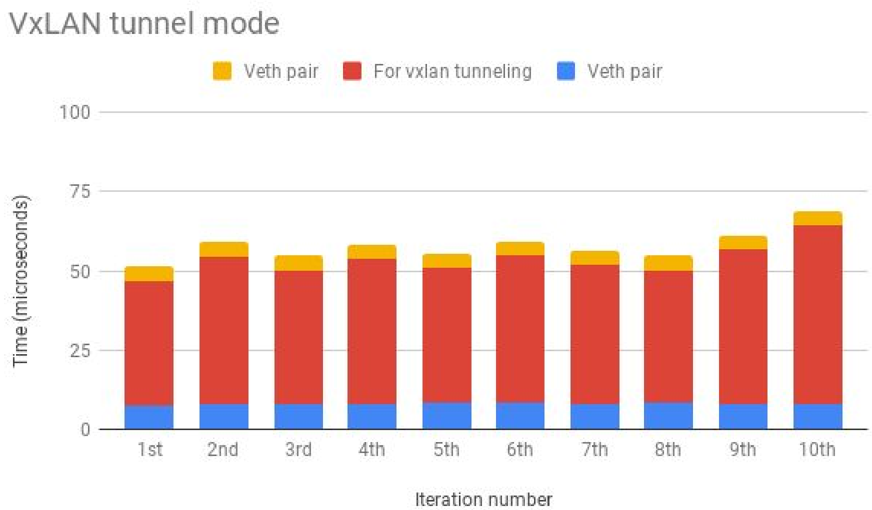
```
   NETWORK NS        INTERFACE    TYPE ADDRESSES                  FUNCTION NAME           TIMESTAMP
[  4026532699]          vxlan0 request 10.0.0.1 -> 10.0.0.2 net_dev_queue      3380054913635303
[  4026532699]          vxlan0 request 10.0.0.1 -> 10.0.0.2 net_dev_start_xmit         3380054913643827
[  4026532699]           veth0  tunnel 192.168.100.1 -> 192.168.200.1 net_dev_queue    3380054913659638
[  4026532699]           veth0  tunnel 192.168.100.1 -> 192.168.200.1 net_dev_start_xmit        3380054913663134
[  4026532553]           veth1  tunnel 192.168.100.1 -> 192.168.200.1 netif_rx         3380054913666962
[  4026532553]           veth1  tunnel 192.168.100.1 -> 192.168.200.1 net_dev_xmit     3380054913669916
[  4026532553]           veth1  tunnel 192.168.100.1 -> 192.168.200.1 net_dev_xmit     3380054913672750
[  4026532553]           veth2  tunnel 192.168.100.1 -> 192.168.200.1 net_dev_queue    3380054913686722
[  4026532553]           veth2  tunnel 192.168.100.1 -> 192.168.200.1 net_dev_start_xmit        3380054913689719
[  4026532761]           veth3  tunnel 192.168.100.1 -> 192.168.200.1 netif_rx         3380054913692907
[  4026532761]           veth3  tunnel 192.168.100.1 -> 192.168.200.1 net_dev_xmit     3380054913695431
[  4026532761]          vxlan0 request 10.0.0.1 -> 10.0.0.2 napi_gro_receive_entry     3380054913894301
[  4026532761]          vxlan0   reply 10.0.0.2 -> 10.0.0.1 net_dev_queue      3380054913919614
[  4026532761]          vxlan0   reply 10.0.0.2 -> 10.0.0.1 net_dev_start_xmit         3380054913923458
[  4026532761]           veth3  tunnel 192.168.200.1 -> 192.168.100.1 net_dev_queue    3380054913931351
[  4026532761]           veth3  tunnel 192.168.200.1 -> 192.168.100.1 net_dev_start_xmit        3380054913934309
[  4026532553]           veth2  tunnel 192.168.200.1 -> 192.168.100.1 netif_rx         3380054913937348
[  4026532553]           veth2  tunnel 192.168.200.1 -> 192.168.100.1 net_dev_xmit     3380054913940004
[  4026532553]           veth2  tunnel 192.168.200.1 -> 192.168.100.1 net_dev_xmit     3380054913942674
[  4026532553]           veth1  tunnel 192.168.200.1 -> 192.168.100.1 net_dev_queue    3380054913951971
[  4026532553]           veth1  tunnel 192.168.200.1 -> 192.168.100.1 net_dev_start_xmit        3380054913954724
[  4026532699]           veth0  tunnel 192.168.200.1 -> 192.168.100.1 netif_rx         3380054913957361
[  4026532699]           veth0  tunnel 192.168.200.1 -> 192.168.100.1 net_dev_xmit     3380054913959748
[  4026532699]          vxlan0   reply 10.0.0.2 -> 10.0.0.1 napi_gro_receive_entry     3380054913969036
```

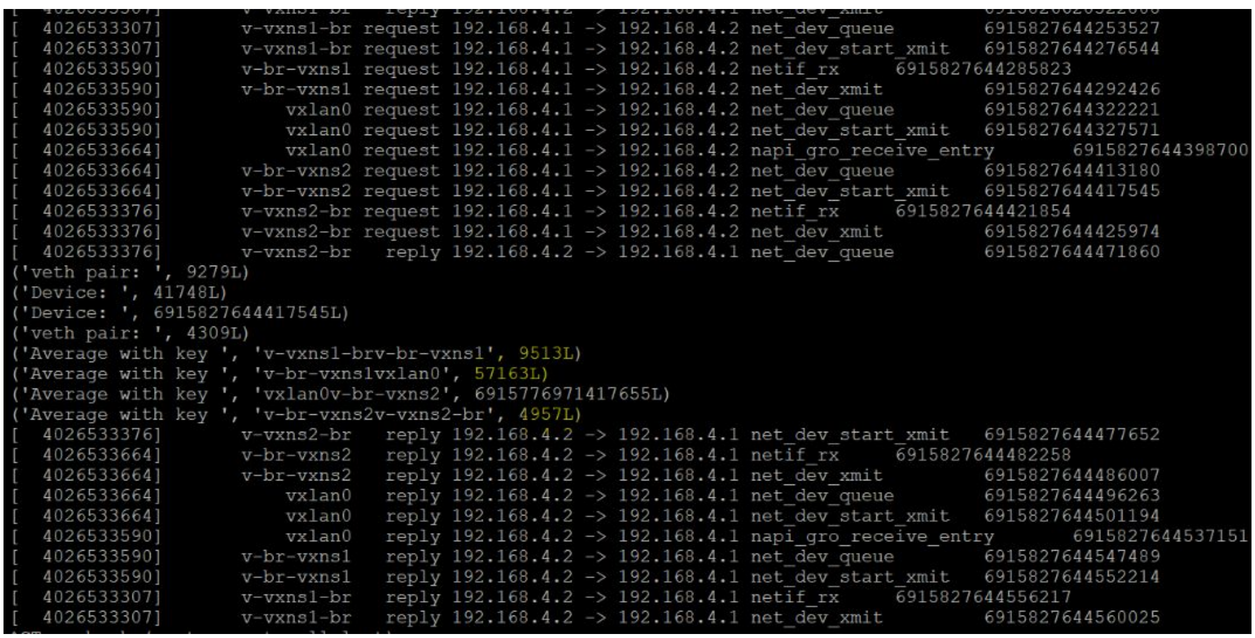
Vxlan0 is the src and destination VxLAN device. Veth0 - veth1 is the link between Src VM namespace and router (which is a namespace). veth2 - veth3 is the link between router and destination VM namespace. 10.0.0.1 and 10.0.0.2 are VMs separated by namespace router. 192.168.100.1 and 192.168.200.1 are the tunnel-endpoint IPs.

Interface / Device	*1st*	*2nd*	*3rd*	*4th*	*5th*	*6th*	*7th*	*8th*	*9th*	*10th*
Veth pair	7.416	8.038	7.815	7.811	8.147	8.239	7.947	8.126	7.973	7.986
For vxlan tunneling	39.47	46.23	42.35	45.94	42.73	46.54	43.99	41.98	48.74	56.29
Veth pair	4.475	5.016	4.564	4.423	4.618	4.559	4.248	4.70	4.577	4.582

All times are in microseconds

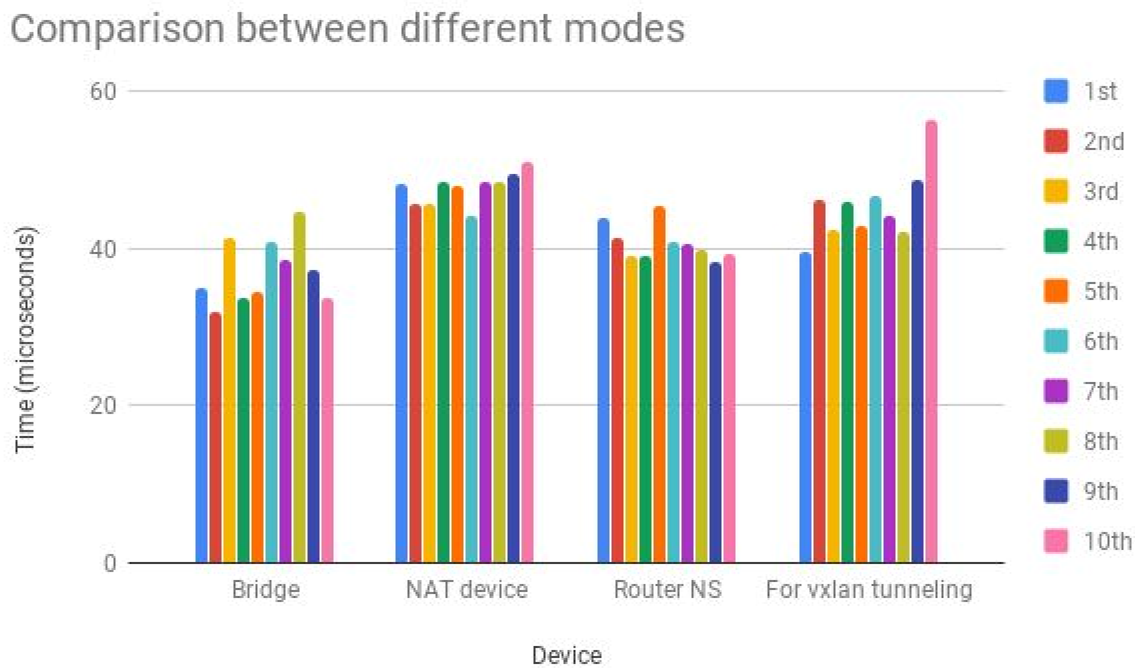
VxLAN tunnel mode





```
[  4026533307]       v-vxns1-br request 192.168.4.1 -> 192.168.4.2 net_dev_queue        6915827644253527
[  4026533307]       v-vxns1-br request 192.168.4.1 -> 192.168.4.2 net_dev_start_xmit   6915827644276544
[  4026533590]       v-br-vxns1 request 192.168.4.1 -> 192.168.4.2 netif_rx     6915827644285823
[  4026533590]       v-br-vxns1 request 192.168.4.1 -> 192.168.4.2 net_dev_xmit         6915827644292426
[  4026533590]           vxlan0 request 192.168.4.1 -> 192.168.4.2 net_dev_queue        6915827644322221
[  4026533590]           vxlan0 request 192.168.4.1 -> 192.168.4.2 net_dev_start_xmit   6915827644327571
[  4026533664]           vxlan0 request 192.168.4.1 -> 192.168.4.2 napi_gro_receive_entry       6915827644398700
[  4026533664]       v-br-vxns2 request 192.168.4.1 -> 192.168.4.2 net_dev_queue        6915827644413180
[  4026533664]       v-br-vxns2 request 192.168.4.1 -> 192.168.4.2 net_dev_start_xmit   6915827644417545
[  4026533376]       v-vxns2-br request 192.168.4.1 -> 192.168.4.2 netif_rx     6915827644421854
[  4026533376]       v-vxns2-br request 192.168.4.1 -> 192.168.4.2 net_dev_xmit         6915827644425974
[  4026533376]       v-vxns2-br   reply 192.168.4.2 -> 192.168.4.1 net_dev_queue        6915827644471860
('veth pair: ', 9279L)
('Device: ', 41748L)
('Device: ', 6915827644417545L)
('veth pair: ', 4309L)
('Average with key ', 'v-vxns1-brv-br-vxns1', 9513L)
('Average with key ', 'v-br-vxns1vxlan0', 57163L)
('Average with key ', 'vxlan0v-br-vxns2', 6915776971417655L)
('Average with key ', 'v-br-vxns2v-vxns2-br', 4957L)
[  4026533376]       v-vxns2-br   reply 192.168.4.2 -> 192.168.4.1 net_dev_start_xmit   6915827644477652
[  4026533664]       v-br-vxns2   reply 192.168.4.2 -> 192.168.4.1 netif_rx     6915827644482258
[  4026533664]       v-br-vxns2   reply 192.168.4.2 -> 192.168.4.1 net_dev_xmit         6915827644486007
[  4026533664]           vxlan0   reply 192.168.4.2 -> 192.168.4.1 net_dev_queue        6915827644496263
[  4026533664]           vxlan0   reply 192.168.4.2 -> 192.168.4.1 net_dev_start_xmit   6915827644501194
[  4026533590]           vxlan0   reply 192.168.4.2 -> 192.168.4.1 napi_gro_receive_entry       6915827644537151
[  4026533590]       v-br-vxns1   reply 192.168.4.2 -> 192.168.4.1 net_dev_queue        6915827644547489
[  4026533590]       v-br-vxns1   reply 192.168.4.2 -> 192.168.4.1 net_dev_start_xmit   6915827644552214
[  4026533307]       v-vxns1-br   reply 192.168.4.2 -> 192.168.4.1 netif_rx     6915827644556217
[  4026533307]       v-vxns1-br   reply 192.168.4.2 -> 192.168.4.1 net_dev_xmit         6915827644560025
```

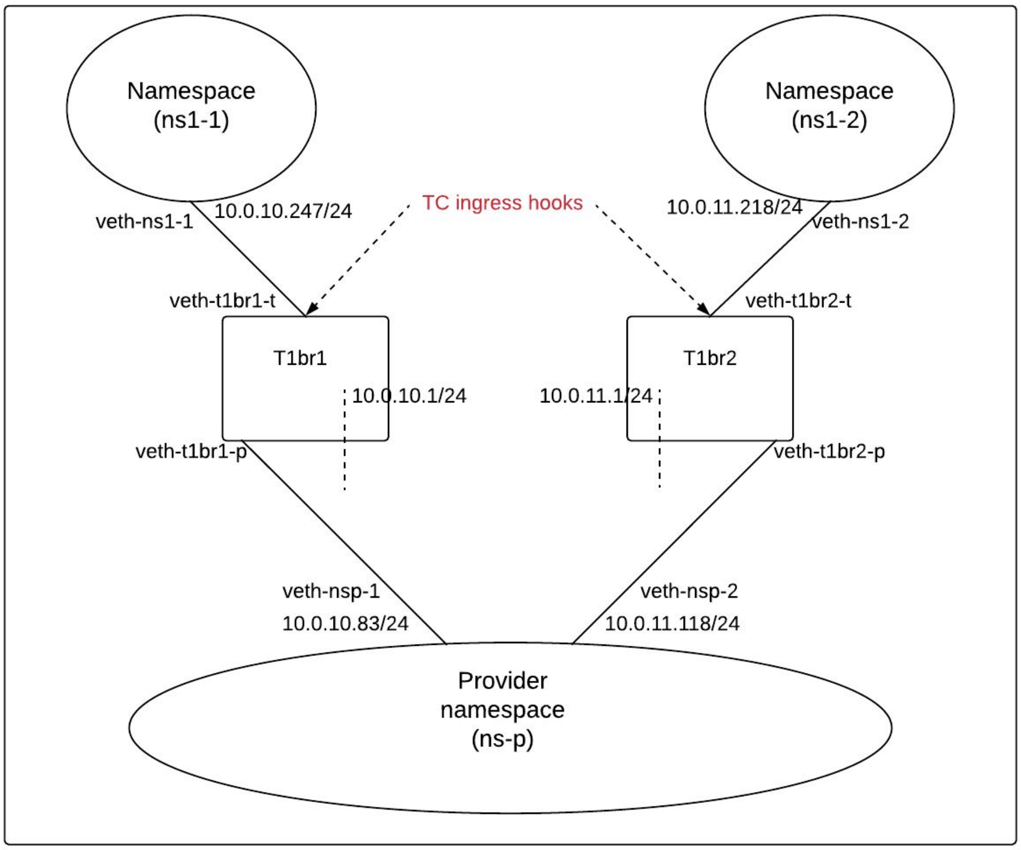
Screenshot of the last iteration of a batch



Comparison between different modes
60
40
20
0
Time (microseconds)
Bridge
NAT device
Router NS
For vxlan tunneling
Device
1st
2nd
3rd
4th
5th
6th
7th
8th
9th
10th

b) Fast Data Path

By implementing fast data path, the packet is made to traverse through the network in a custom way, and this even gives the provision of making the packet traverse faster through the network as compared to the traditional traversal. The topology considered by us for the comparison is



Hypervisor

Interface / Device	*1st*	*2nd*	*3rd*	*4th*	*5th*	*6th*	*7th*	*8th*	*9th*	*10th*
Veth pair	5.594	5.608	5.723	6.558	5.449	5.588	5.639	5.323	5.794	5.866
Bridge	21.52	22.23	46.50	30.37	21.91	24.00	43.46	20.49	21.67	26.85
Veth pair	3.179	3.240	3.320	3.496	3.167	3.20	3.232	3.051	3.284	3.468
Provider NS	26.92	27.45	27.92	30.78	26.72	26.71	27.44	25.60	27.84	29.20
Veth pair	3.019	3.042	3.020	3.831	3.025	2.98	2.946	2.834	3.192	3.209

Bridge	10.43	10.61	11.56	11.66	10.28	14.58	13.33	9.70	11.35	19.79
Veth pair	2.778	2.786	2.768	3.253	2.616	2.699	2.743	2.566	2.83	3.674

All times are in microseconds

Readings for normal data path

Interface / Device	*1st*	*2nd*	*3rd*	*4th*	*5th*	*6th*	*7th*	*8th*	*9th*	*10th*
Veth pair	5.676	5.601	5.893	5.666	5.729	6.061	5.953	5.898	5.797	5.948
Device (Cost of redirection)	**38.28**	**44.22**	**28.48**	**35.03**	**28.62**	**29.36**	**29.31**	**29.1**	**34.9**	**33.32**
Veth pair	2.85	3.121	2.792	3.147	2.756	3.178	2.923	2.977	3.042	3.111

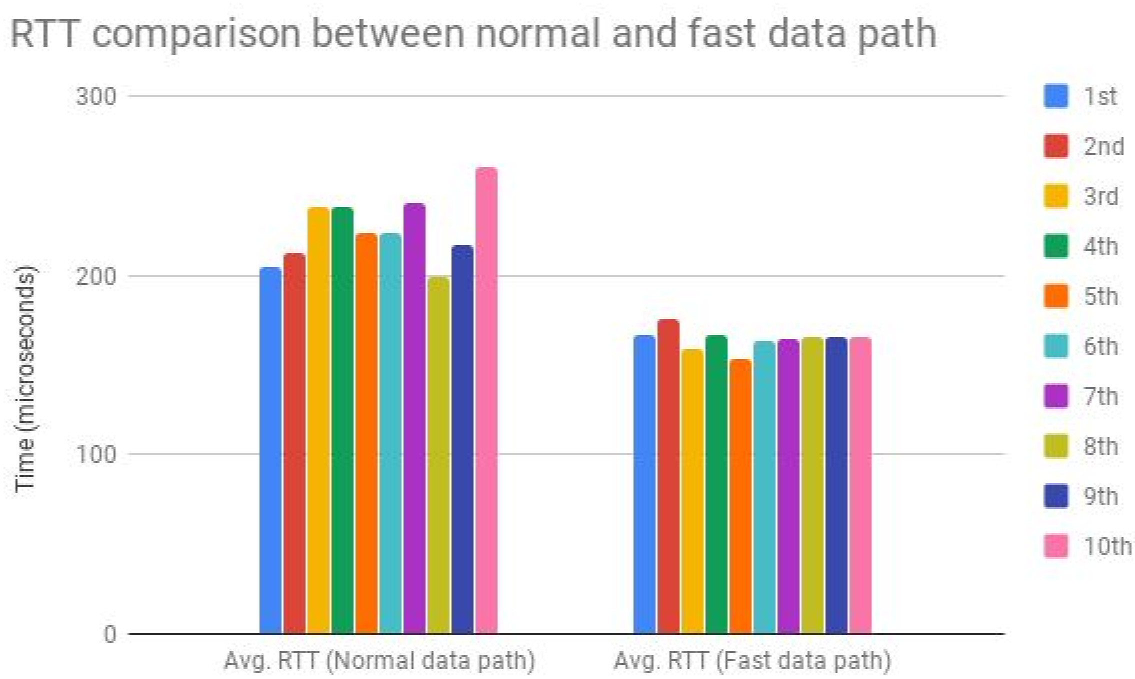
All times are in microseconds

Readings for the Fast Data Path implementation

	1st	*2nd*	*3rd*	*4th*	*5th*	*6th*	*7th*	*8th*	*9th*	*10th*
Avg. RTT (Normal data path)	0.205	0.212	0.238	0.238	0.223	0.224	0.24	0.199	0.217	0.26
Avg. RTT (Fast data path)	0.167	0.176	0.159	0.167	0.153	0.163	0.164	0.166	0.166	0.166

*All times are in **milliseconds***

Average ping time (RTT) comparison between normal and fast data path



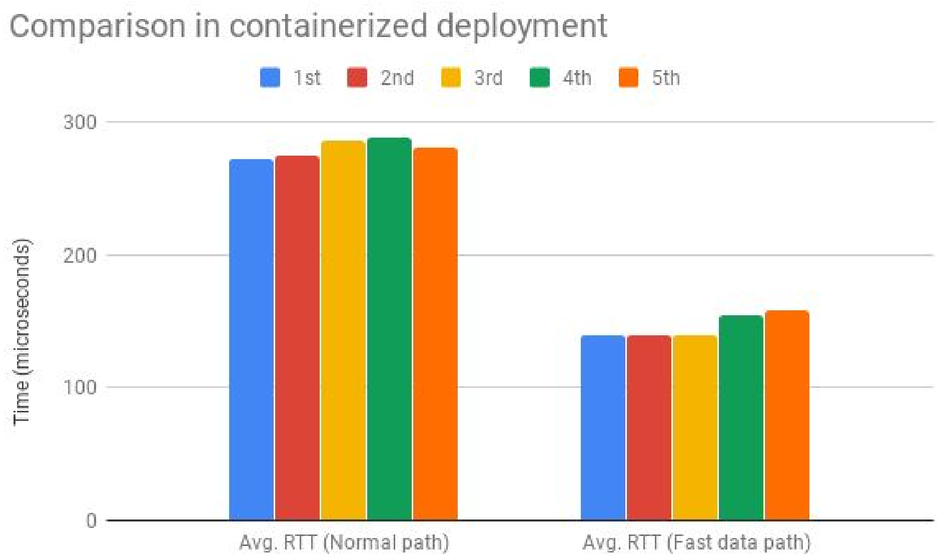
Now, let's consider the fast path with containerized deployment

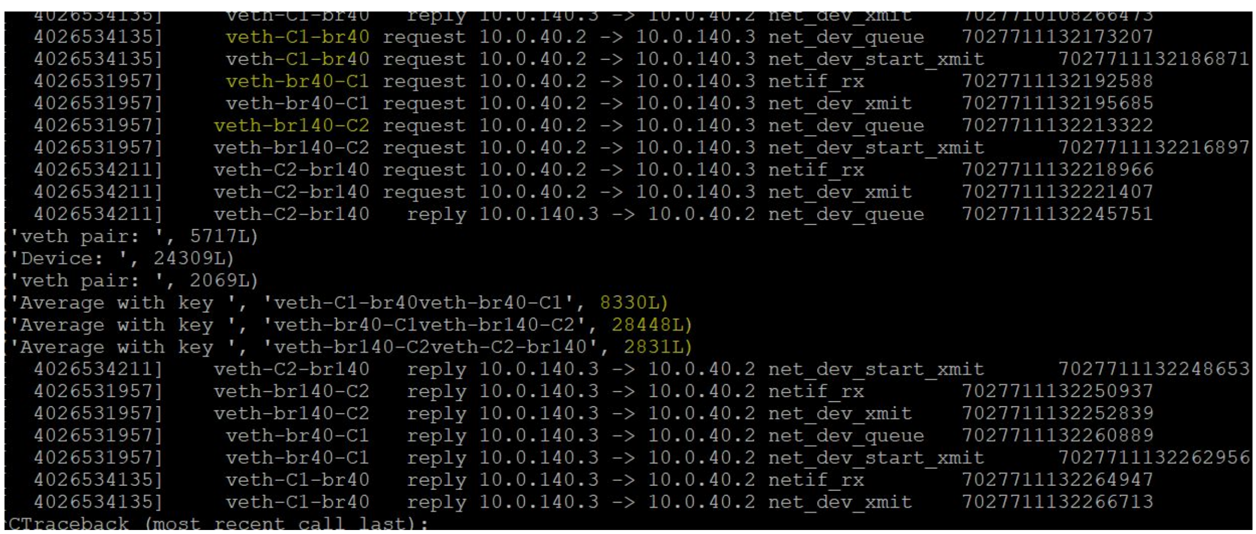
Interface / Device	*1st*	*2nd*	*3rd*	*4th*	*5th*
veth pair	6.116	5.9	6.092	9.351	8.33
Device (Cost of redirection)	25.46	25.18	25.64	27.34	28.45
veth pair	2.548	2.539	2.555	3.08	2.831

All times are in microseconds

	1st	*2nd*	*3rd*	*4th*	*5th*
Avg. RTT (Normal path)	0.273	0.275	0.286	0.289	0.281
Avg. RTT (Fast data path)	0.140	0.139	0.139	0.154	0.158

*All times are in **milliseconds***




```
 4026534135]      veth-C1-br40 request 10.0.40.2 -> 10.0.140.3 net_dev_queue    7027711132173207
 4026534135]      veth-C1-br40 request 10.0.40.2 -> 10.0.140.3 net_dev_start_xmit      7027711132186871
 4026531957]      veth-br40-C1 request 10.0.40.2 -> 10.0.140.3 netif_rx         7027711132192588
 4026531957]      veth-br40-C1 request 10.0.40.2 -> 10.0.140.3 net_dev_xmit     7027711132195685
 4026531957]     veth-br140-C2 request 10.0.40.2 -> 10.0.140.3 net_dev_queue    7027711132213322
 4026531957]     veth-br140-C2 request 10.0.40.2 -> 10.0.140.3 net_dev_start_xmit      7027711132216897
 4026534211]     veth-C2-br140 request 10.0.40.2 -> 10.0.140.3 netif_rx         7027711132218966
 4026534211]     veth-C2-br140 request 10.0.40.2 -> 10.0.140.3 net_dev_xmit     7027711132221407
 4026534211]     veth-C2-br140   reply 10.0.140.3 -> 10.0.40.2 net_dev_queue    7027711132245751
('veth pair: ', 5717L)
('Device: ', 24309L)
('veth pair: ', 2069L)
('Average with key ', 'veth-C1-br40veth-br40-C1', 8330L)
('Average with key ', 'veth-br40-C1veth-br140-C2', 28448L)
('Average with key ', 'veth-br140-C2veth-C2-br140', 2831L)
 4026534211]     veth-C2-br140   reply 10.0.140.3 -> 10.0.40.2 net_dev_start_xmit      7027711132248653
 4026531957]     veth-br140-C2   reply 10.0.140.3 -> 10.0.40.2 netif_rx         7027711132250937
 4026531957]     veth-br140-C2   reply 10.0.140.3 -> 10.0.40.2 net_dev_xmit     7027711132252839
 4026531957]      veth-br40-C1   reply 10.0.140.3 -> 10.0.40.2 net_dev_queue    7027711132260889
 4026531957]      veth-br40-C1   reply 10.0.140.3 -> 10.0.40.2 net_dev_start_xmit      7027711132262956
 4026534135]      veth-C1-br40   reply 10.0.140.3 -> 10.0.40.2 netif_rx         7027711132264947
 4026534135]      veth-C1-br40   reply 10.0.140.3 -> 10.0.40.2 net_dev_xmit     7027711132266713
^CTraceback (most recent call last):
```

6. Summary and Future Scope

As a part of this great and evolving area, we got a chance to explore the management and functionality. On the management side, we monitored the time taken by the packet to traverse on the link or to get processed by a device, taking into consideration different modes. On the other hand, we implemented a simple policy based forwarding mechanism by providing a fast data path functionality in the system. Alongside, we compared the performance gain that we achieve with fast data path with the normal data path. In the future, it would nice to explore the performance in large networks.

References:

[1] A thorough introduction to eBPF
[2] eBPF Based Networking
[3] A presentation by William Tu of VMware on using eBPF to offload OVS: Offloading OVS flow processing using eBPF
[4] A presentation by F. Risso, M. Bertrone, and M. Bernal on IO Visor and OVN: Coupling the flexibility of OVN with the efficiency of IOVisor: Architecture and Demo
[5] A presentation by Jesper Brouer of Red Hat: XDP – Intro and Future Use Cases
[6] High Speed Packet Filtering on Linux by Gilberto Bertin
[7] Leveraging XDP for Programmable, High Performance Data Path in OpenStack
[8] Creation of complex network services with eBPF
Creating Complex Network Services with eBPF: Experience and Lessons Learned
[9] A presentation by Yunsong Lu and Yan Chen of Huawei on using IO Visor for Ethernet driver: CETH for XDP:Common Ethernet Driver Framework for Faster Network IO
[10] A presentation by Yunsong Lu on using eBPF for virtual networking: Evolving Virtual Networking with IO Visor
[11] https://github.com/cilium/cilium
[12] Building an Extensible Open vSwitch Datapath
http://delivery.acm.org/10.1145/3140000/3139657/p72-tu.pdf?ip=152.7.255.201&id=3139657&acc=ACTIVE%20SERVICE&key=6ABC8B4C00F6EE47%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&__acm__=1542139112_97364ee72f7825283ca39de5385c86b2
[13] Why measure performance of CNI plugins:
https://thenewstack.io/kubernetes-and-cni-whats-next-making-it-easier-to-write-networking-plugins/
[14] Feature comparison of various CNIs:
https://chrislovecnm.com/kubernetes/cni/choosing-a-cni-provider/
[15] cgnet:
https://github.com/kinvolk/cgnet
[16] Cilium:
https://kubernetes.io/blog/2017/12/using-ebpf-in-kubernetes/
[17] Calico:
https://www.projectcalico.org/wp-content/uploads/2018/04/ProjectCalico.v3.1.datasheet.pdf
[18] Flannel:
https://github.com/coreos/flannel
https://coreos.com/flannel/docs/latest/flannel-config.html
[19] Weave:
https://www.weave.works/docs/
[20] Understanding eBPF architecture
https://schd.ws/hosted_files/kccnceu18/7f/KubeCon.pdf
[21] eBPF based Container Networking:
https://pdfs.semanticscholar.org/46d8/18dedcf1ed235a6f23ef826b02476897e5a1.pdf?_ga=2.198904748.264721484.1538524519-1426312100.1538524519

[22] eBPF based Real-Time computing in IoT edge network
https://arxiv.org/pdf/1805.02797.pdf
[23] Tracing a packet's journey using Linux tracepoints, perf and eBPF
https://blog.yadutaf.fr/2017/07/28/tracing-a-packet-journey-using-linux-tracepoints-perf-ebpf/
https://blog.yadutaf.fr/2016/03/30/turn-any-syscall-into-event-introducing-ebpf-kernel-probes/
https://jvns.ca/blog/2017/07/05/linux-tracing-systems/