Porting VPP to FreeBSD: Basic Usage

BY TOM JONES

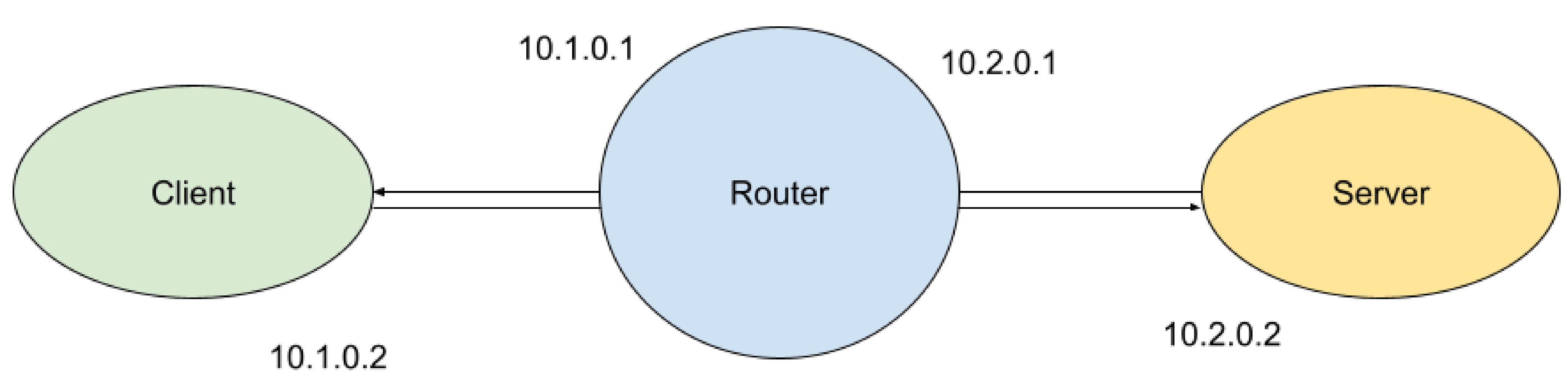
The Vector Packet Process (VPP) is a high-performance framework for processing packets in userspace. Thanks to a project by the FreeBSD Foundation and RGNets, I was sponsored to port VPP to FreeBSD and I am really happy to share some basic usage with readers of the *FreeBSD Journal*.

VPP enables forwarding and routing applications to be written in userspace with a API-controllable interface. High-performance networking is made possible by DPDK on Linux and DPDK and netmap on FreeBSD. These APIs allow direct 0 copy access to data and can be used to make forwarding applications that can significantly exceed the host's forwarding performance.

VPP is a full-network router replacement, and, as such, needs some host configuration to be usable. This article presents some complete examples of how to use VPP on FreeBSD which most users should be able to follow with a virtual machine of their own. VPP on FreeBSD also runs on real hardware.

This introduction to using VPP on FreeBSD gives an example set up showing how to do things on FreeBSD. VPP resources can be difficult to find, the documentation from the project at https://fd.io is high quality.

Lets Build a Router



VPP can be put to lots of purposes, the main one and easiest to configure is as some form of router or bridge. For our example of using VPP as a router, we need to construct a small example network with three nodes — a client, a server and the router.

To show you how VPP can be used on FreeBSD, I'm going to construct an example network with the minimum of overhead. All you need is VPP and a FreeBSD system. I'm also going to install iperf3 so we can generate and observe some traffic going through our router.

From a FreeBSD with a recent ports tree you can get our two required tools with the `pkg` command like so:

```
host # pkg install vpp iperf3
```

To create three nodes for our network, we are going to take advantage of one of FreeBSD's most powerful features, VNET jails. VNET jails give us completely isolated instances of the network stack, they are similar in operation to Linux Network Namespaces. To create a VNET, we need to add the **vnet** option when creating a jail and pass along the interfaces it will use.

Finally we will connect our nodes using **epair** interfaces. These offer the functionality of two ends of an ethernet cable — if you are familiar with **veth** interfaces on linux they offer similar functionality.

We can construct our test network with the following 5 commands:

```
host # ifconfig epair create
epair0a
host # ifconfig epair create
epair1a
# jail -c name=router persist vnet vnet.interface=epair0a vnet.interface=epair1a
# jail -c name=client persist vnet vnet.interface=epair0b
# jail -c name=server persist vnet vnet.interface=epair1b
```

The flags to take note of in these jail commands are **persist** without which the jail will be removed automatically because there are no processes running inside it, **vnet** which makes this jail a vnet jail and **vnet.interface=** which assigns the given interface to the jail.

When an interface is moved to a new vnet, all of its configuration is stripped away \- worth noting in case you configure an interface and then move it to a jail and wonder why nothing is working.

Set up peers

Before turning to VPP, let us set up the client and server sides of the network. Each of these needs to be given an ip address and the interface moved to the up state. We will also need to configure default routes for the client and server jails.

```
host # jexec client
# ifconfig
lo0: flags=8008<LOOPBACK,MULTICAST> metric 0 mtu 16384
        options=680003<RXCSUM,TXCSUM,LINKSTATE,RXCSUM_IPV6,TXCSUM_IPV6>
        groups: lo
        nd6 options=21<PERFORMNUD,AUTO_LINKLOCAL>
epair0b: flags=1008842<BROADCAST,RUNNING,SIMPLEX,MULTICAST,LOWER_UP> metric 0
mtu 1500
        options=8<VLAN_MTU>
        ether 02:90:ed:bd:8b:0b
        groups: epair
        media: Ethernet 10Gbase-T (10Gbase-T <full-duplex>)
        status: active
        nd6 options=29<PERFORMNUD,IFDISABLED,AUTO_LINKLOCAL>
# ifconfig epair0b inet 10.1.0.2/24 up
# route add default 10.1.0.1
add net default: gateway 10.1.0.1
```

```
host # jexec server
# ifconfig epair1b inet 10.2.0.2/24 up
# route add default 10.2.0.1
add net default: gateway 10.2.0.1
```

Our client and server jails now have ip addresses and routes towards the VPP router.

Netmap requirements

For our examples, we are going to use VPP with netmap, a high-performance userspace networking framework that ships as a default component of FreeBSD. Netmap requires a little interface configuration before it can be used — the interface needs to be in the `up` state and have the `promisc` option configured.

```
host # jexec router
# ifconfig epair0a promisc up
# ifconfig epair1a promisc up
```

Now we are able to start using VPP\!

VPP First Commands

VPP is very flexible and offers configuration by a config file, a command line interface, and an API with mature Python bindings. VPP needs a base configuration telling it where to get commands and the names of the files it uses for control if they aren't the default. We can give VPP a minimal configuration file on the command line as part of its arguments. For this example, we tell VPP to drop into interactive mode – offer us a cli, and we tell vpp to only load the plugins we will use (netmap) which is a sensible default.

If we don't disable all plugins, we will either need to set up the machine to use DPDK, or disable that plugin on its own. The syntax to do so is the same as the syntax to enable the netmap plugin.

```
host # vpp "unix { interactive} plugins { plugin default { disable } plugin
netmap_plugin.so { enable } plugin ping_plugin.so { enable } }"

       _______    _        _   _____  ___
    __/ __/ _ \  (_)__    | | / / _ \/ _ \
    _/ _// // / / / _ \   | |/ / ___/ ___/
    /_/ /____(_)_/\___/   |___/_/  /_/

vpp# show int
          Name               Idx        State  MTU (L3/IP4/IP6/MPLS)
Counter      Count
local0                        0          down          0/0/0/0
```

If all is set up, you will see the VPP banner and the default cli prompt (`vpp#`).

The VPP command line interface offers a lot of options for the creation and management of interfaces, groups like bridges, the addition of routes and tools for interrogating the performance of a VPP instance.

The syntax of the interface configuration commands is similar to the linux iproute2 commands – coming from FreeBSD these are a little alien, but they are reasonably clear once you start to get used to them.

Our VPP server hasn't been configured with any host interfaces yet, `show int` only lists the default `local0` interface.

To use our netmap interfaces with vpp, we need to create them first and then we can configure them.

The create command lets us create new interfaces, we use the netmap subcommand and the host interface.

```
vpp# create netmap name epair0a
netmap_create_if:164: mem 0x882800000
netmap-epair0a
vpp# create netmap name epair1a
netmap-epair1a
```

Each netmap interface is created with a prefix of **`netmap-`** . With the interfaces created, we can configure them for use and start using VPP as a router.

```
vpp# set int ip addr netmap-epair0a 10.1.0.1/24
vpp# set int ip addr netmap-epair1a 10.2.0.1/24
vpp# show int addr
local0 (dn):
netmap-epair0a (dn):
  L3 10.1.0.1/24
netmap-epair1a (dn):
  L3 10.2.0.1/24
```

The command **`show int addr`** (the shortened version of **`show interface address`**) confirms our ip address assignment has worked. We can then bring the interfaces up:

```
vpp# set int state netmap-epair0a up
vpp# set int state netmap-epair1a up
vpp# show int
          Name               Idx State  MTU (L3/IP4/IP6/MPLS)
Counter   Count
local0                        0   down         0/0/0/0
netmap-epair0a                1   up           9000/0/0/0
netmap-epair1a                2   up           9000/0/0/0
```

With our interfaces configured, we can test functionality from VPP by using the ping command:

```
vpp# ping 10.1.0.2
116 bytes from 10.1.0.2: icmp_seq=2 ttl=64 time=7.9886 ms
116 bytes from 10.1.0.2: icmp_seq=3 ttl=64 time=10.9956 ms
116 bytes from 10.1.0.2: icmp_seq=4 ttl=64 time=2.6855 ms
116 bytes from 10.1.0.2: icmp_seq=5 ttl=64 time=7.6332 ms

Statistics: 5 sent, 4 received, 20% packet loss
vpp# ping 10.2.0.2
116 bytes from 10.2.0.2: icmp_seq=2 ttl=64 time=5.3665 ms
116 bytes from 10.2.0.2: icmp_seq=3 ttl=64 time=8.6759 ms
116 bytes from 10.2.0.2: icmp_seq=4 ttl=64 time=11.3806 ms
```

```
116 bytes from 10.2.0.2: icmp_seq=5 ttl=64 time=1.5466 ms

Statistics: 5 sent, 4 received, 20% packet loss
```

And if we jump to the client jail, we can verify that VPP is acting as a router:

```
client # ping 10.2.0.2
PING 10.2.0.2 (10.2.0.2): 56 data bytes
64 bytes from 10.2.0.2: icmp_seq=0 ttl=63 time=0.445 ms
64 bytes from 10.2.0.2: icmp_seq=1 ttl=63 time=0.457 ms
64 bytes from 10.2.0.2: icmp_seq=2 ttl=63 time=0.905 ms
^C
--- 10.2.0.2 ping statistics ---
3 packets transmitted, 3 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.445/0.602/0.905/0.214 ms
```

As a final piece of initial set up, we will start up an iperf3 server in the server jail and use the client to do a TCP throughput test.:

```
server # iperf3 -s

client # iperf3 -c 10.2.0.2
Connecting to host 10.2.0.2, port 5201
[  5] local 10.1.0.2 port 63847 connected to 10.2.0.2 port 5201
[ ID]   Interval          Transfer    Bitrate          Retr  Cwnd
[  5]   0.00-1.01   sec   341 MBytes  2.84 Gbits/sec  0     1001 KBytes
[  5]   1.01-2.01   sec   488 MBytes  4.07 Gbits/sec  0     1.02 MBytes
[  5]   2.01-3.01   sec   466 MBytes  3.94 Gbits/sec  144   612 KBytes
[  5]   3.01-4.07   sec   475 MBytes  3.76 Gbits/sec  0     829 KBytes
[  5]   4.07-5.06   sec   452 MBytes  3.81 Gbits/sec  0     911 KBytes
[  5]   5.06-6.03   sec   456 MBytes  3.96 Gbits/sec  0     911 KBytes
[  5]   6.03-7.01   sec   415 MBytes  3.54 Gbits/sec  0     911 KBytes
[  5]   7.01-8.07   sec   239 MBytes  1.89 Gbits/sec  201   259 KBytes
[  5]   8.07-9.07   sec   326 MBytes  2.75 Gbits/sec  0     462 KBytes
[  5]   9.07-10.06  sec   417 MBytes  3.51 Gbits/sec  0     667 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID]   Interval         Transfer     Bitrate          Retr
[  5]   0.00-10.06  sec  3.98 GBytes  3.40 Gbits/sec  345          sender
[  5]   0.00-10.06  sec  3.98 GBytes  3.40 Gbits/sec               receiver

iperf Done.
```

VPP Analysis

Now that we have sent some traffic through VPP, the output of `show int` contains more information:

```
vpp# show int
     Name               Idx    State  MTU (L3/IP4/IP6/MPLS)         Counter          Count
local0                   0     down        0/0/0/0
netmap-epair0a           1      up         9000/0/0/0       rx packets           4006606
                                                            rx bytes          6065742126
                                                            tx packets           2004365
                                                            tx bytes           132304811
                                                            drops                          2
                                                            ip4                  4006605
netmap-epair1a           2      up         9000/0/0/0       rx packets           2004365
                                                            rx bytes           132304811
                                                            tx packets           4006606
                                                            tx bytes          6065742126
                                                            drops                          2
                                                            ip4                  2004364
```

The interface command now gives us a summary of the bytes and packets that have passed across the VPP interfaces. This can be really helpful to debug how traffic is moving around, especially if your packets are going missing.

The V in VPP stands for vector and this has two meanings in the project. VPP aims to use vectorised instructions to accelerate packet processing and it also bundles groups of packets together into vectors to optimize processing. The theory here is to take groups of packets through the processing graph together saving cache thrashing and giving optimal performance.

VPP has a lot of tooling for interrogating what is happening while packets are processed. Deep tuning is beyond this article, but a first tool to look at to understand what is happening in VPP is the **`runtime`** command.

Runtime data is gathered for each vector as it passes through the VPP processing graph, it collects how long it takes to transverse each node and the number of vectors processed.

To use the run time tooling, it is good to have some traffic. Start a long running iperf3 throughput test like so:

```
client # iperf3 -c 10.2.0.2 -t 1000
```

Now in the VPP jail, we can clear the gathered run time statistics so far, wait a little bit and then look at how we are doing:

```
vpp# clear runtime
... wait ~5 seconds ...
vpp# show runtime
Time 5.1, 10 sec internal node vector rate 124.30 loops/sec 108211.07
  vector rates in 4.4385e5, out 4.4385e5, drop 0.0000e0, punt 0.0000e0
          Name                 State         Calls       Vectors     Suspends     Clocks    Vectors/Call
ethernet-input                 active        18478       2265684     0            3.03e1    122.62
fib-walk                       any wait          0       0           3            1.14e4    0.00
ip4-full-reassembly-expire-wal any wait          0       0           102          7.63e3    0.00
ip4-input                      active        18478       2265684     0            3.07e1    122.62
ip4-lookup                     active        18478       2265 684    0            3.22e1    122.62
ip4-rewrite                    active        18478       2265684     0            3.05e1    122.62
ip6-full-reassembly-expire-wal any wait          0       0           102          5.79e3    0.00
ip6-mld-process                any wait          0       0           5            6.12e3    0.00
```

```
ip6-ra-process                  any wait         0        0           5         1.18e4    0.00
netmap-epair0a-output           active        8383        755477      0         1.12e1    90.12
netmap-epair0a-tx               active        8383        755477      0         1.17e3    90.12
netmap-epair1a-output           active       12473        1510207     0         1.04e1    121.08
netmap-epair1a-tx               active       12473        1510207     0         2.11e3    121.08
netmap-input                    interrupt wa 16698        2265684     0         4.75e2    135.69
unix-cli-process-0              active           0        0           13        7.34e4    0.00
unix-epoll-input                polling     478752        0           0         2.98e4    0.00
```

The columns in the **show runtime** output give us a great idea of what is happening in vpp. They tell us which nodes have been active since the run time counters were cleared, their current state, how many times this node was called, how much time it used, and how many vectors were processed per call. Out of the box, the maximum vector size for vpp is 255.

A final debugging task you can perform is to examine the packet processing graph in its entirety with the **show vlib graph** command. This command shows each node and the potential parent and child nodes which could lead to it.

Next Steps

VPP is an incredible piece of software — once the headaches of compatibility were addressed, the core parts of VPP were reasonably straightforward to port. Even with just minimal tuning, VPP is able to reach some impressive performance with netmap on FreeBSD, and it does even better if you configure DPDK. The VPP documentation is slowly getting more information about running on FreeBSD, but the developers really need example use cases of VPP on FreeBSD.

If you start from this example of a simple network, it should be reasonably straight forward to port it onto a large network with faster interfaces.

TOM JONES is a FreeBSD committer interested in keeping the network stack fast.