
Vector Packet Processor Documentation

Release 0.1

John DeNisco

Jul 06, 2018

Contents

1	Overview	3
1.1	What is VPP?	3
1.2	Features	6
1.3	Performance	11
1.4	Architectures and Operating Systems	13
2	Getting Started Guides	15
2.1	Users	15
2.2	Developers	39
3	Use Cases	61
3.1	FD.io VPP with Virtual Machines	61
3.2	Using VPP as a Home Gateway	69
3.3	vSwitch/vRouter	72
4	Troubleshooting	75
4.1	CPU Load/Usage	75
5	User Guides	79
5.1	Progressive VPP Tutorial	79
5.2	API User Guides	103
6	Reference	105
6.1	Command Line Reference	105
6.2	VPP with Containers	116

This is beta VPP Documentation it is not meant to be complete or accurate yet!!!!

FD.io Vector Packet Processing (VPP) is a fast, scalable and multi-platform network stack.

FD.io VPP is, at it's core, a scalable layer 2-4 network stack. It supports integration into both Open Stack and Kubernetes environments. It supports network management features including configuration, counters and sampling. It supports extending with plugins, tracing and debugging. It supports use cases such as vSwitch, vRouter, Gateways, Firewalls and Load Balancers, to name but a few. Finally it is useful both a software development kit or an appliance out of the box.

1.1 What is VPP?

FD.io's Vector Packet Processing (VPP) technology is a *Fast, Scalable and Deterministic, Packet Processing* stack that runs on commodity CPUs. It provides out-of-the-box production quality switch/router functionality and much, much more. FD.io VPP is at the same time, an *Extensible and Modular Design* and *Developer Friendly* framework, capable of boot-strapping the development of packet-processing applications. The benefits of FD.io VPP are its high performance, proven technology, its modularity and flexibility, integrations and rich feature set.

For more detailed information, see the following sections:

1.1.1 Packet Processing

- Layer 2 - 4 Network Stack
 - Fast lookup tables for routes, bridge entries
 - Arbitrary n-tuple classifiers
 - Control Plane, Traffic Management and Overlays
- Linux and FreeBSD support
 - Wide support for standard Operating System Interfaces such as AF_Packet, Tun/Tap & Netmap.
- Wide network and cryptographic hardware support with DPDK.
- Container and Virtualization support
 - Para-virtualized interfaces; Vhost and Virtio
 - Network Adapters over PCI passthrough
 - Native container interfaces; MemIF
- Universal Data Plane: one code base, for many use cases
 - Discrete appliances; such as Routers and Switches.

- Cloud Infrastructure and Virtual Network Functions
- Cloud Native Infrastructure
- The same binary package for all use cases.
- Out of the box production quality, with thanks to CSIT.

For more information, please see *Features* for the complete list.

1.1.2 Fast, Scalable and Deterministic

- Continuous integration and system testing
 - Including continuous & extensive, latency and throughput testing
- Layer 2 Cross Connect (L2XC), typically achieve 15+ Mpps per core.
- Tested to achieve **zero** packet drops and ~15µs latency.
- Performance scales linearly with core/thread count
- Supporting millions of concurrent lookup tables entries

Please see *Performance* for more information.

1.1.3 Developer Friendly

- Extensive runtime counters; throughput, instructions per cycle, errors, events etc.
- Integrated pipeline tracing facilities
- Multi-language API bindings
- Integrated command line for debugging
- Fault-tolerant and upgradable
 - Runs as a standard user-space process for fault tolerance, software crashes seldom require more than a process restart.
 - Improved fault-tolerance and upgradability when compared to running similar packet processing in the kernel, software updates never require system reboots.
 - Development experience is easier compared to similar kernel code
 - Hardware isolation and protection (iommu)
- Built for security
 - Extensive white-box testing
 - Image segment base address randomization
 - Shared-memory segment base address randomization
 - Stack bounds checking
 - Static analysis with Coverity

1.1.4 Extensible and Modular Design

- Pluggable, easy to understand & extend
- Mature graph node architecture
- Full control to reorganize the pipeline
- Fast, plugins are equal citizens

Modular, Flexible, and Extensible

The FD.io VPP packet processing pipeline is decomposed into a ‘packet processing graph’. This modular approach means that anyone can ‘plugin’ new graph nodes. This makes VPP easily extensible and means that plugins can be customized for specific purposes. VPP is also configurable through its Low-Level API.

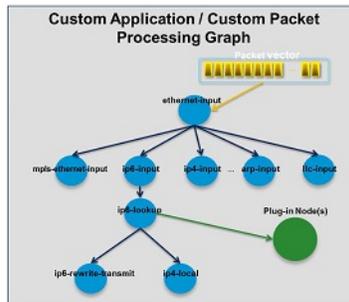


Fig. 1: Extensible and modular graph node architecture.

At runtime, the FD.io VPP platform assembles a vector of packets from RX rings, typically up to 256 packets in a single vector. The packet processing graph is then applied, node by node (including plugins) to the entire packet vector. The received packets typically traverse the packet processing graph nodes in the vector, when the network processing represented by each graph node is applied to each packet in turn. Graph nodes are small and modular, and loosely coupled. This makes it easy to introduce new graph nodes and rewire existing graph nodes.

Plugins are [shared libraries](#) and are loaded at runtime by VPP. VPP find plugins by searching the plugin path for libraries, and then dynamically loads each one in turn on startup. A plugin can introduce new graph nodes or rearrange the packet processing graph. You can build a plugin completely independently of the FD.io VPP source tree, which means you can treat it as an independent component.

What is vector packet processing?

As the name implies, FD.io VPP uses vector packet processing, as opposed to scalar packet processing. A scalar packet processing stack simply processes one packet at a time: an interrupt handling function takes a single packet from a device rx ring, and processes it by traversing a set of functions: A calls B calls C . . . return return return, then return from interrupt. For each packet, one of three things happens: the path punts, drops, or rewrites and forwards the packet.

Scalar packet processing is simple, but problematic in these ways:

- When the path length exceeds the size of the I-cache, thrashing occurs. Each packet incurs an identical set of I-cache misses The only solution: bigger caches.
- Deep call stack adds load-store-unit pressure since stack-locals fall out of the L1 D-cache
- Aside from prefetching packet data - probably not in time - one can't address dependent read latency on table walks in a meaningful way

In contrast, vector packet processing constructs vectors of packets by scraping up to 256 packets at a time from device rx rings, and processes them using a directed graph of node. The graph scheduler invokes one node dispatch function at a time, restricting stack depth to a few stack frames.

This scheme fixes the I-cache thrashing problem.

Graph node dispatch functions iterate across up to 256 vector elements. Processing the first packet in a vector warms up the I-cache. The remaining packets all hit in the I-cache, reducing I-cache miss stalls by up to two orders of magnitude.

Given a vector of packets, one can pipeline and prefetch to cover dependent read latency on table data needed to process packets.

Vector packet processing techniques lead to a **stable** graph dispatch circuit time equilibrium. For a given offered load, imagine that the dispatch circuit time - and hence the vector size - converge to certain values. Say that an operating system event such as a clock-tick interrupt introduces a delay into the main dispatch loop.

The next rx vector will be larger. Larger vectors are processed more efficiently: I-cache warmup costs are amortized over a larger number of packets.

Rapidly, the rx vector size and the dispatch circuit time return to the previous equilibrium values. Given a relatively stable offered load, it's an important advantage for the vector size to remain stable in the face of exogenous events.

1.2 Features

<i>SDN & Cloud Integrations</i>		<i>Control Plane</i>	<i>Plugins</i>
<i>Tunnels</i>	<i>Layer 4</i>	<i>Traffic Management</i>	
	<i>Layer 3</i>		
	<i>Layer 2</i>		
<i>Devices</i>			

1.2.1 Devices

Hardware

- DPDK
 - Network Interfaces
 - Cryptographic Devices
- Open Data Plane
- Intel Ethernet Adaptive Virtual Function

Operating System

- Netmap
- af_packet
- Tap V2 (FastTap)

Virtualization:

- SSVM
- Vhost / VirtIO

Containers

- Vhost-user
- MemIF

1.2.2 SDN & Cloud Integrations

1.2.3 Traffic Management

IP Layer Input Checks

- Source Reverse Path Forwarding
- Time To Live expiration
- IP header checksum
- Layer 2 Length < IP Length

Classifiers

- Multiple million Classifiers - Arbitrary N-tuple

Policers

- Colour Aware & Token Bucket
- Rounding Closest/Up/Down
- Limits in PPS/KBPS
- Types:
 - Single Rate Two Colour
 - Single Rate Three Colour
 - Dual Rate Three Colour
- Action Triggers
 - Conform
 - Exceed
 - Violate
- Actions Type
 - Drop
 - Transmit
 - Mark-and-transmit

Switched Port Analyzer (SPAN) * mirror traffic to another switch port

ACLs

- Stateful
- Stateless

COP

MAC/IP Pairing

(security feature).

1.2.4 Layer 2

MAC Layer

- Ethernet

Discovery

- Cisco Discovery Protocol
- Link Layer Discovery Protocol (LLDP)

Link Layer Control Protocol

- Bit Index Explicit Replication – Link Layer Multi-cast forwarding.
- Link Layer Control (LLC) - multiplex protocols over the MAC layer.
- Spatial Reuse Protocol (SRP)
- High-Level Data Link Control (HDLC)
- Logical link control (LLC)
- Link Agg Control Protocol (Active/Active, Active/Passive) – 18.04

Virtual Private Networks

- MPLS
 - MPLS-o-Ethernet – Deep label stacks supported
- Virtual Private LAN Service (VPLS)
- VLAN
- Q-in-Q
- Tag-rewrite (VTR) - push/pop/Translate (1:1,1:2, 2:1,2:2)
- Ethernet flow point Filtering
- Layer 2 Cross Connect

Bridging

- Bridge Domains
- MAC Learning (50k addresses)
- Split-horizon group support
- Flooding

ARP

- Proxy
- Termination
- Bidirectional Forwarding Detection

Integrated Routing and Bridging (IRB)

- Flexibility to both route and switch between groups of ports.
- Bridged Virtual Interface (BVI) Support, allows traffic switched traffic to be routed.

1.2.5 Layer 3

IP Layer

- ICMP
- IPv4
- IPv6
- IPSEC
- Link Local Addressing

MultiCast

- Multicast FiB
- IGMP

Virtual Routing and forwarding (VRF)

- VRF scaling, thousands of tables.
- Controlled cross-VRF lookups

Multi-path

- Equal Cost Multi Path (ECMP)
- Unequal Cost Multi Path (UCMP)

IPv4

- ARP
- ARP Proxy
- ARP Snooping

IPv6

- Neighbour discovery (ND)
- ND Proxy
- Router Advertisement
- Segment Routing
- Distributed Virtual Routing Resolution

Forwarding Information Base (FIB)

- Hierarchical FIB
- Memory efficient
- Multi-million entry scalable
- Lockless/concurrent updates
- Recursive lookups
- Next hop failure detection
- Shared FIB adjacencies
- Multicast support
- MPLS support

1.2.6 Layer 4

1.2.7 Tunnels

Layer 2

- L2TP
- PPP
- VLAN

Layer 3

- Mapping of Address and Port with Encapsulation (MAP-E)
- Lightweight IPv4 over IPv6
 - An Extension to the Dual-Stack Lite Architecture

- GENEVE
- VXLAN

Segment Routing

- IPv6
- MPLS

Generic Routing Encapsulation (GRE) * GRE over IPSEC * GRE over IP * MPLS * NSH

1.2.8 Control Plane

- DHCP client/proxy
- DHCPv6 Proxy

1.2.9 Plugins

- iOAM

1.3 Performance

1.3.1 Overview

One of the benefits of FD.io VPP, is high performance on relatively low-power computing, this performance is based on the following features:

- A high-performance user-space network stack designed for commodity hardware.
 - L2, L3 and L4 features and encapsulations.
- Optimized packet interfaces supporting a multitude of use cases.
 - An integrated vhost-user backend for high speed VM-to-VM connectivity.
 - An integrated memif container backend for high speed Container-to-Container connectivity.
 - An integrated vhost based interface to punt packets to the Linux Kernel.
- The same optimized code-paths run execute on the host, and inside VMs and Linux containers.
- Leverages best-of-breed open source driver technology: [DPDK](#).
- Tested at scale; linear core scaling, tested with millions of flows and mac addresses.

These features have been designed to take full advantage of common micro-processor optimization techniques, such as:

- Reducing cache and TLS misses by processing packets in vectors.
- Realizing IPC gains with vector instructions such as: SSE, AVX and NEON.
- Eliminating mode switching, context switches and blocking, to always be doing useful work.
- Cache-lined aligned buffers for cache and memory efficiency.

1.3.2 Packet Throughput Graphs

These are some of the packet throughput graphs for FD.io VPP 18.04 from the [CSIT 18.04 benchmarking report](#).

L2 Ethernet Switching Throughput Tests

VPP NDR 64B packet throughput in 1 Core, 1 Thread setup, is presented in the graph below.

NDR Performance Tests

This is a VPP NDR 64B packet throughput in 1 Core, 1 Thread setup, live graph of the NDR (No Drop Rate) L2 Performance Tests.

IPv4 Routed-Forwarding Performance Tests

VPP NDR 64B packet throughput in 1t1c setup (1thread, 1core) is presented in the graph below.

IPv6 Routed-Forwarding Performance Tests

VPP NDR 78B packet throughput in 1t1c setup (1 thread, 1 core) is presented in the graph below.

1.3.3 Trending Throughput Graphs

These are some of the trending packet throughput graphs from the [CSIT trending dashboard](#). **Please note that,** performance in the trending graphs will change on a nightly basis in line with the software development cycle.

L2 Ethernet Switching Performance Tests

This is a live graph of the 1 Core, 1 Thread, L2 Ethernet Switching Performance Tests Test on the x520 NIC.

IPv4 Routed-Forwarding Performance Tests

This is a live graph of the IPv4 Routed Forwarding Switching Performance Tests.

IPv6 Routed-Forwarding Performance Tests

VPP NDR 78B packet throughput in 1t1c setup (1 thread, 1 core) is presented in the trending graph below.

1.3.4 For More information on CSIT

These are FD.io Continuous System Integration and Testing (CSIT)'s documentation links.

- [CSIT Code Documentation](#)
- [CSIT Test Overview](#)
- [VPP Performance Dashboard](#)

1.4 Architectures and Operating Systems

1.4.1 Architectures

- – The FD.io VPP platform supports:
 - * x86/64
 - * ARM

1.4.2 Operating Systems and Packaging

FD.io VPP supports package installation on the following recent LTS operating systems releases:

- – Operating Systems:
 - * Debian
 - * Ubuntu
 - * CentOS
 - * OpenSUSE

2.1 Users

2.1.1 Installing VPP Binaries from Packages

If you are simply using vpp, it can be convenient to simply install the packages. This guide will describe how pull and install the VPP packages.

Package Descriptions

The following is a brief description of the packages to be installed with VPP.

Packages

vpp

Vector Packet Processing executables

- vpp - the vector packet engine
- vpp_api_test - vector packet engine API test tool
- vpp_json_test - vector packet engine JSON test tool

vpp-lib

Vector Packet Processing runtime libraries. This package contains the VPP shared libraries, including:

- vppinfra - Foundation library supporting vectors, hashes, bitmaps, pools, and string formatting.
- svm - vm library

- vlib - vector processing library
- vlib-api - binary API library
- vnet - network stack library

vpp-plugins

Vector Packet Processing plugin modules

- acl
- dpdk
- flowprobe
- gtpu
- ixge
- kubeproxy
- l2e
- lb
- memif
- nat
- pppoe
- sixrd
- stn

vpp-dbg

Vector Packet Processing debug symbols

vpp-dev

Vector Packet Processing development support. This package contains development support files for the VPP libraries

vpp-api-java

JAVA binding for the VPP Binary API.

vpp-api-python

Python binding for the VPP Binary API.

vpp-api-lua

Lua binding for the VPP Binary API.

Installing on Ubuntu

The following are instructions on how to install VPP on Ubuntu.

Ubuntu 16.04 - Setup the fd.io Repository

From the following choose one of the releases to install.

Update the OS

It is probably a good idea to update and upgrade the OS before starting

```
apt-get update
```

Point to the Repository

Create a file “`/etc/apt/sources.list.d/99fd.io.list`” with the contents that point to the version needed. The contents needed are shown below.

VPP latest Release

Create the file `/etc/apt/sources.list.d/99fd.io.list` with contents:

```
deb [trusted=yes] https://nexus.fd.io/content/repositories/fd.io.ubuntu.xenial.main/ .  
↪/
```

VPP stable/1804 Branch

Create the file `/etc/apt/sources.list.d/99fd.io.list` with contents:

```
deb [trusted=yes] https://nexus.fd.io/content/repositories/fd.io.stable.1804.ubuntu.  
↪xenial.main/ ./
```

VPP master Branch

Create the file `/etc/apt/sources.list.d/99fd.io.list` with contents:

```
deb [trusted=yes] https://nexus.fd.io/content/repositories/fd.io.master.ubuntu.xenial.  
↪main/ ./
```

Install the Mandatory Packages

```
sudo apt-get update  
sudo apt-get install vpp vpp-lib vpp-plugin
```

Install the Optional Packages

```
sudo apt-get install vpp-dbg vpp-dev vpp-api-java vpp-api-python vpp-api-lua
```

Uninstall the Packages

```
sudo apt-get remove --purge vpp*
```

Installing on Centos

The following are instructions on how to install VPP on Centos.

Setup the fd.io Repository (Centos 7.3)

From the following choose one of the releases to install.

Update the OS

It is probably a good idea to update and upgrade the OS before starting

```
yum update
```

Point to the Repository

Create a file “`/etc/yum.repos.d/fdio-release.repo`” with the contents that point to the version needed. The contents needed are shown below.

VPP latest Release

Create the file “`/etc/yum.repos.d/fdio-release.repo`”.

```
[fdio-release]
name=fd.io release branch latest merge
baseurl=https://nexus.fd.io/content/repositories/fd.io.centos7/
enabled=1
gpgcheck=0
```

VPP stable/1804 Branch

Create the file “`/etc/yum.repos.d/fdio-release.repo`”.

```
[fdio-stable-1804]
name=fd.io stable/1804 branch latest merge
baseurl=https://nexus.fd.io/content/repositories/fd.io.stable.1804.centos7/
enabled=1
gpgcheck=0
```

VPP master Branch

Create the file “/etc/yum.repos.d/fdio-release.repo”.

```
[fdio-master]
name=fd.io master branch latest merge
baseurl=https://nexus.fd.io/content/repositories/fd.io.master.centos7/
enabled=1
gpgcheck=0
```

Install VPP RPMs

```
sudo yum install vpp
```

Install the optional RPMs

```
sudo yum install vpp-plugins vpp-devel vpp-api-python vpp-api-lua vpp-api-java
```

Uninstall the VPP RPMs

```
sudo yum autoremove vpp*
```

Installing on openSUSE

The following are instructions on how to install VPP on openSUSE.

Installing

To install VPP on openSUSE first pick the following release and execute the appropriate commands.

openSUSE Tumbleweed (rolling release)

```
sudo zypper install vpp vpp-plugins
```

openSUSE Leap 42.3

```
sudo zypper addrepo --name network https://download.opensuse.org/repositories/network/
↪openSUSE_Leap_42.3/network.repo
sudo zypper install vpp vpp-plugins
```

Uninstall

```
sudo zypper remove -u vpp vpp-plugins
```

openSUSE Tumbleweed (rolling release)

```
sudo zypper remove -u vpp vpp-plugins
```

openSUSE Leap 42.3

```
sudo zypper remove -u vpp vpp-plugins
sudo zypper removerepo network
```

For More Information

For more information on VPP with openSUSE, please look at the following post.

- <https://www.suse.com/communities/blog/vector-packet-processing-vpp-opensuse/>

2.1.2 Writing VPP Documentation

Building VPP Documents

Overview

These instructions show how the VPP documentation sources are built.

FD.io VPP Documentation uses *reStructuredText* (rst) files, which are used by *Sphinx*. We will also cover how to view your build on Read the Docs in *Using Read the Docs*.

To build your files, you can either *Create a Virtual Environment using virtualenv*, which installs all the required applications for you, or you can *Install Sphinx manually*.

Create a Virtual Environment using virtualenv

For more information on how to use the Python virtual environment check out *Installing packages using pip and virtualenv*.

Get the Documents

For example start with a clone of the vpp-docs.

```
$ git clone https://github.com/fdioDocs/vpp-docs
$ cd vpp-docs/docs
```

Install the virtual environment

In your `vpp-docs` directory, run:

```
$ python -m pip install --user virtualenv
$ python -m virtualenv env
$ source env/bin/activate
$ pip install -r etc/requirements.txt
```

Which installs all the required applications into it's own, isolated, virtual environment, so as to not interfere with other builds that may use different versions of software.

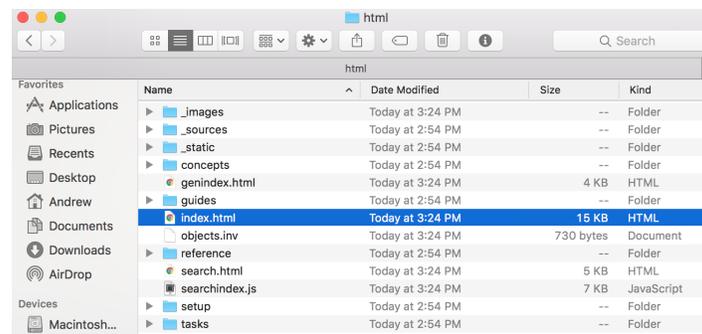
Build the html files

Be sure you are in your `vpp-docs/docs` directory, since that is where Sphinx will look for your `conf.py` file, and build the `.rst` files into an `index.html` file:

```
$ make html
```

View the results

If there are no errors during the build process, you should now have an `index.html` file in your `vpp-docs/docs/_build/html` directory, which you can then view in your browser.



Whenever you make changes to your `.rst` files that you want to see, repeat this build process.

Note: To exit from the virtual environment execute:

```
$ deactivate
```

Install Sphinx manually

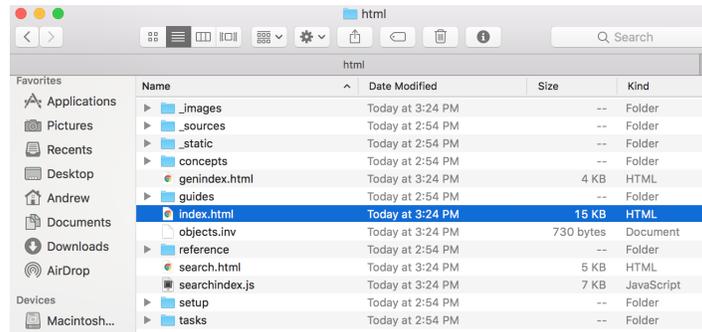
Skip this step if you created a `virtualenv` in the previous step. If you dont want to create a `virtualenv`, you should install Sphinx [here](#), and follow their [getting started guide](#).

Building these files will generate an `index.html` file, which you can then view in your browser to verify and see your file changes.

To *build* your files, make sure you're in your **vpp-docs/docs** directory, where your **conf.py** file is located, and run:

```
$ make html
```

If there are no errors during the build process, you should now have an **index.html** file in your **vpp-docs/docs/_build/html** directory, which you can then view in your browser.



Whenever you make changes to your **.rst** files that you want to see, repeat this build process.

Using Read the Docs

Read the Docs is a website that “simplifies software documentation by automating building, versioning, and hosting of your docs for you”. Essentially, it accesses your Github repo to generate the **index.html** file, and then displays it on its own *Read the Docs* webpage so others can view your documentation.

Create an account on *Read the Docs* if you haven't already.

Go to your [dashboard](#), and click on “Import a Project”.

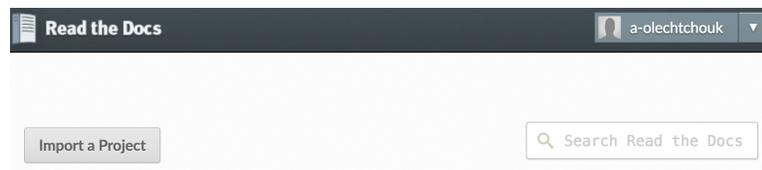
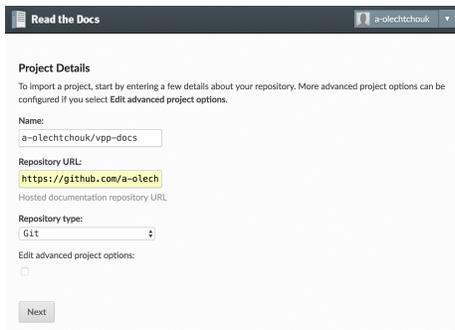


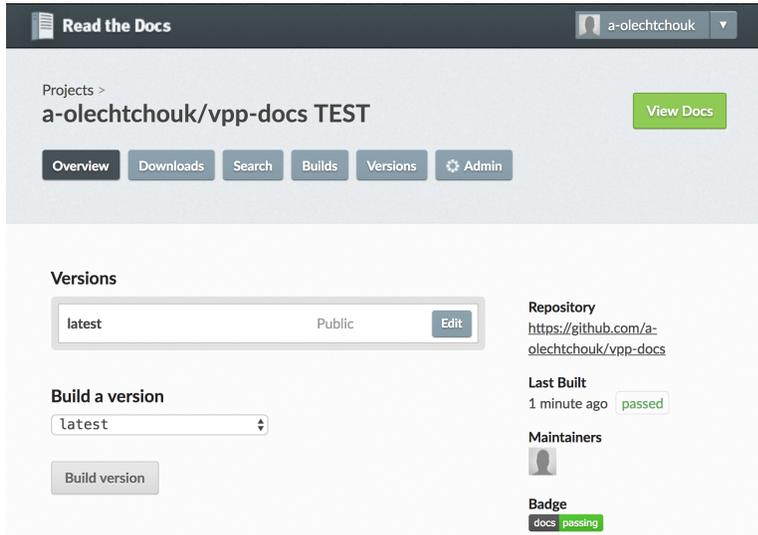
Fig. 1: This will bring you to a page where you can choose to import a repo from your Github account (only if you've linked your Github account to your Read the Docs account), or to import a repo manually. In this example, we'll do it manually. Click “Import Manually”.

This will bring you to a page that asks for your repo details. Set “Name” to your forked repo name, or whatever you want. Set “Repository URL” to the URL of your forked repo (<https://github.com/YOURUSERNAME/vpp-docs>). “Repository type” should already be selected to “Git”. Then click “Next”.



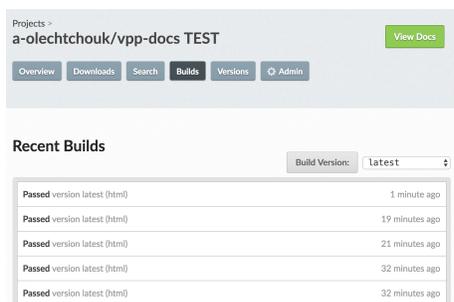
This will bring you to a project page of your repo on Read the Docs. You can confirm it's the correct repo by checking on the right side of the page the Repository URL.

Then click on “Build Version”.



Which takes you to another page showing your recent builds.

Then click on “Build Version:”. This should “Trigger” a build. After about a minute or so you can refresh the page and see that your build “Passed”.



Now on your builds page from the previous image, you can click “View Docs” at the top-right, which will take you a *readthedocs.io* page of your generated build!

Pushing your changes to the VPP Docs Repository

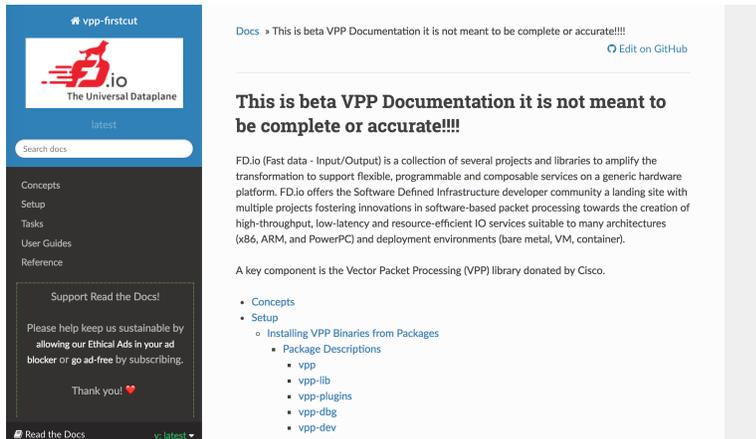
Overview

This section will cover how to fork your own branch of the [fdioDocs/vpp-docs](#) repository, clone that repo locally to your computer, make changes to it, and how to issue a pull request when you want your changes to be reflected on the main repo.

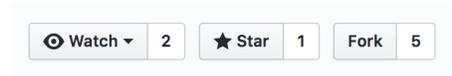
Forking your own branch

In your browser, navigate to the repo you want to branch off of. In this case, the [fdioDocs/vpp-docs](#) repo. At the top right of the page you should

see this:



Click on “Fork”, and then a pop-up should appear where you should then click your Github username. Once this is done, it should automatically take you to the Github page where your new branch is located, just like in the image below.



Now your **own branch** can be **cloned** to your computer using the URL (<https://github.com/YOURUSERNAME/vpp-docs>) of the Github page where your branch is located.

Creating a local repository

Now that you have your own branch of the main repository on Github, you can store it locally on your computer. In your shell, navigate to the directory where you want to store your branch/repo. Then execute:

```
$ git clone https://github.com/YOURUSERNAME/vpp-docs
```

This will create a directory on your computer named **vpp-docs**, the name of the repo.

Now that your branch is on your computer, you can modify and build files however you wish.

If you are not on the master branch, move to it.

```
$ git checkout master
```

Keeping your files in sync with the main repo

The following talks about remote branches, but keep in mind that there are currently *two* branches, your local “master” branch (on your computer), and your remote “origin or origin/master” branch (the one you created using “Fork” on the Github website).

You can view your *remote* repositories with:

```
$ git remote -v
```

At this point, you may only see the remote branch that you cloned from.

```
Macintosh:docs Andrew$ git remote -v
origin  https://github.com/a-olechtchouk/vpp-docs (fetch)
origin  https://github.com/a-olechtchouk/vpp-docs (push)
```

Now you want to create a new remote repository of the main vpp-docs repo (naming it upstream).

```
$ git remote add upstream https://github.com/fdioDocs/vpp-docs
```

You can verify that you have added a remote repo using the previous **git remote -v** command.

```
$ git remote -v
origin  https://github.com/a-olechtchouk/vpp-docs (fetch)
origin  https://github.com/a-olechtchouk/vpp-docs (push)
upstream https://github.com/fdioDocs/vpp-docs (fetch)
upstream https://github.com/fdioDocs/vpp-docs (push)
```

If there have been any changes to files in the main repo (hopefully not the same files you were working on!), you want to make sure your local branch is in sync with them.

To do so, fetch any changes that the main repo has made, and then merge them into your local master branch using:

```
$ git fetch upstream
$ git merge upstream/master
```

Note: This is optional, so don't do these commands if you just want one local branch!!!

You may want to have multiple branches, where each branch has its own different features, allowing you to have multiple pull requests out at a time. To create a new local branch:

```
$ git checkout -b cleanup-01
$ git branch
* cleanup-01
  master
  overview
```

Now you can redo the previous steps **for** "Keeping your files in sync with the main repo" **for** your newly created local branch, and **then** depending on which branch you want **to** send out a pull request **for**, proceed below.

Pushing to your branch

Now that your files are in sync, you want to add modified files, commit, and push them from *your local branch* to your *personal remote branch* (not the main fdioDocs repo).

To check the status of your files, run:

```
$ git status
```

In the output example below, I deleted `gettingsources.rst`, made changes to `index.rst` and `pushingapatch.rst`, and have created a new file called `buildingrst.rst`.

```
Macintosh:docs Andrew$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

   deleted:    tasks/writingdocs/gettingsources.rst

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

   modified:   tasks/writingdocs/index.rst
   modified:   tasks/writingdocs/pushingapatch.rst

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   tasks/writingdocs/buildingrst.rst
```

To add files (use `git add -A` to add all modified files):

```
$ git add FILENAME1 FILENAME2
```

Commit and push using:

```
$ git commit -m 'A descriptive commit message for two files.'
```

Push your changes for the branch where your changes were made

```
$ git push origin <branch name>
```

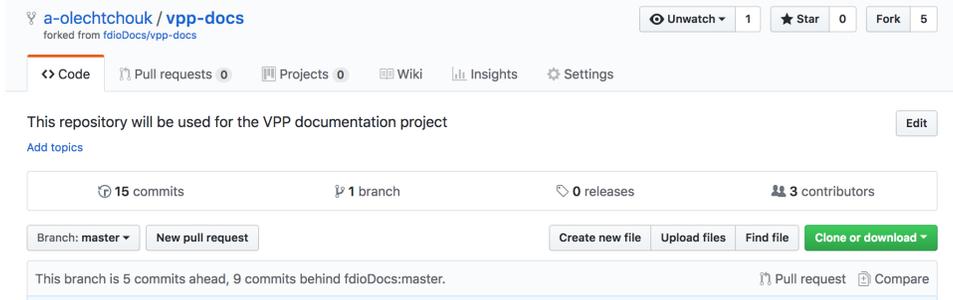
Here, your personal remote branch is “origin” and your local branch is “master”.

Note: Using `git commit` after adding your files saves a “Snapshot” of them, so it’s very hard to lose your work if you *commit often*.

Initiating a pull request (Code review)

Once you’ve pushed your changes to your remote branch, go to your remote branch on Github (<https://github.com/YOURUSERNAME/vpp-docs>), and click on “New pull request”.

This will bring you to a “Comparing changes” page. Click “Create new pull request”.



Create new pull request

Which will open up text fields to add information to your pull request.

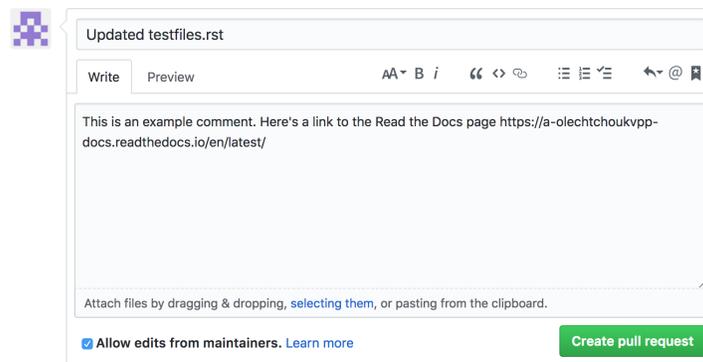


Fig. 2: Then finally click “Create pull request” to complete the pull request.

Your documents will be reviewed. To this same branch make the changes requested from the review and then push your new changes. There is no need to create another pull request.

```
$ git commit -m 'A descriptive commit message for the new changes'
$ git push origin <branch name>
```

Additional Git commands

You may find some of these Git commands useful:

Use **git diff** to quickly show the file changes and repo differences of your commits.

Use **git rm FILENAME** to stop tracking a file and to remove it from your remote branch and local directory. Use flag **-r** to remove folders/directories. E.g (**git rm -r oldfolder**)

reStructured Text Style Guide

Most of these documents are written in reStructured Text (rst). This chapter describes some of the Sphinx Markup Constructs used in these documents. The Sphinx style guide can be found at: [Sphinx Style Guide](#) For a more detailed list of Sphinx Markup Constructs please refer to: [Sphinx Markup Constructs](#)

This document is also an example of a directory structure for a document that spans multiple pages. Notice we have the file **index.rst** and the then documents that are referenced in index.rst. The referenced documents are shown at the bottom of this page.

A label is shown at the top of this page. Then the first construct describes a the document title **FD.io Style Guide**. Text usually follows under each title or heading.

A **Table of Contents** structure is shown below. Using **toctree** in this way will show the headings in a nicely in the generated documents.

Heading 1

This is the top heading level. More levels are shown below.

Heading 2

Heading 3

Heading 4

Heading 5

Bullets, Bold and Italics

Bold text can be show with **Bold Text**, Italics with *Italic text*. Bullets like so:

- Bullet 1
- Bullet 2

Notes

A note can be used to describe something not in the normal flow of the paragraph. This is an example of a note.

Note: Using **git commit** after adding your files saves a “Snapshot” of them, so it’s very hard to lose your work if you *commit often*.

Code Blocks

This paragraph describes how to do **Console Commands**. When showing VPP commands it is recommended that the command be executed from the linux console as shown. The Highlighting in the final documents shows up nicely this way.

```
$ sudo bash
# vppctl show interface
      Name                Idx      State      Counter      Count
TenGigabitEthernet86/0/0    1        up        rx packets    6569213
                          rx bytes    9928352943
                          tx packets    50384
                          tx bytes    3329279
TenGigabitEthernet86/0/1    2        down
VirtualEthernet0/0/0        3        up        rx packets    50384
                          rx bytes    3329279
                          tx packets    6569213
                          tx bytes    9928352943
                          drops      1498
local0                       0        down
#
```

The **code-block** construct is also used for code samples. The following shows how to include a block of “C” code.

```
#include <vlib/unix/unix.h>
abf_policy_t *
abf_policy_get (u32 index)
{
    return (pool_elt_at_index (abf_policy_pool, index));
}
```

Diffs are generated in the final docs nicely with “::” at the end of the description like so:

```
diff --git a/src/vpp/vnet/main.c b/src/vpp/vnet/main.c
index 6e136e19..69189c93 100644
--- a/src/vpp/vnet/main.c
+++ b/src/vpp/vnet/main.c
@@ -18,6 +18,8 @@
    #include <vlib/unix/unix.h>
    #include <vnet/plugin/plugin.h>
    #include <vnet/ethernet/ethernet.h>
+#include <vnet/ip/ip4_packet.h>
+#include <vnet/ip/format.h>
    #include <vpp/app/version.h>
    #include <vpp/api/vpe_msg_enum.h>
    #include <limits.h>
@@ -400,6 +402,63 @@ VLIB_CLI_COMMAND (test_crash_command, static) = {
    #endif
```

Labels, References

A link or reference to other paragraphs within these documents can be done with following construct.

In this example the reference points the label **showintcommand**. The label **styleguide03** is shown at the top of this page. A label used in this way must be above a heading or title.

Show Interface command.

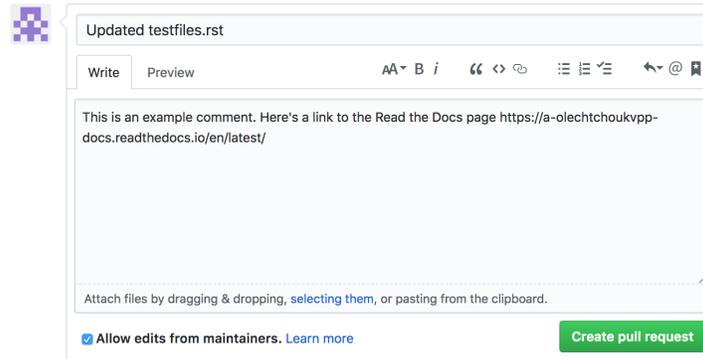
External Links

An external link is done with the following construct:

Sphinx Markup Constructs

Images

Images should be placed in the directory docs/_images. They can then be referenced with following construct. This is the image created to show a pull request.



Including a file

A complete file should be included with the following construct. It is recommended it be included with its own .rst file describing the file included. This is an example of an xml file is included.

An XML File

An example of an XML file.

```
<domain type='kvm' id='54'>
  <name>iperf-server</name>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <memoryBacking>
    <hugepages>
      <page size='2048' unit='KiB' />
    </hugepages>
  </memoryBacking>
  <vcpu placement='static'>1</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='x86_64' machine='pc-i440fx-xenial'>hvm</type>
    <boot dev='hd' />
  </os>
  <features>
    <acpi />
    <apic />
  </features>
  <cpu mode='host-model'>
    <model fallback='allow'></model>
  </cpu>
</domain>
```

(continues on next page)

(continued from previous page)

```

<numa>
  <cell id='0' cpus='0' memory='262144' unit='KiB' memAccess='shared' />
</numa>
</cpu>
<clock offset='utc'>
  <timer name='rtc' tickpolicy='catchup' />
  <timer name='pit' tickpolicy='delay' />
  <timer name='hpet' present='no' />
</clock>
<on_poweroff>destroy</on_poweroff>
<on_reboot>restart</on_reboot>
<on_crash>restart</on_crash>
<pm>
  <suspend-to-mem enabled='no' />
  <suspend-to-disk enabled='no' />
</pm>
<devices>
  <emulator>/usr/bin/kvm</emulator>
  <disk type='file' device='disk'>
    <driver name='qemu' type='qcow2' />
    <source file='/tmp/xenial-mod.img' />
    <backingStore />
    <target dev='vda' bus='virtio' />
    <alias name='virtio-disk0' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x07' function='0x0' />
  </disk>
  <disk type='file' device='cdrom'>
    <driver name='qemu' type='raw' />
    <source file='/scratch/jdenisco/sae/configs/cloud-config.iso' />
    <backingStore />
    <target dev='hda' bus='ide' />
    <readonly />
    <alias name='ide0-0-0' />
    <address type='drive' controller='0' bus='0' target='0' unit='0' />
  </disk>
  <controller type='usb' index='0' model='ich9-ehci1'>
    <alias name='usb' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x7' />
  </controller>
  <controller type='pci' index='0' model='pci-root'>
    <alias name='pci.0' />
  </controller>
  <controller type='ide' index='0'>
    <alias name='ide' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1' />
  </controller>
  <controller type='virtio-serial' index='0'>
    <alias name='virtio-serial0' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0' />
  </controller>
  <interface type='vhostuser'>
    <mac address='52:54:00:4c:47:f2' />
    <source type='unix' path='/tmp//vm00.sock' mode='server' />
    <model type='virtio' />
    <alias name='net1' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0' />
  </interface>

```

(continues on next page)

(continued from previous page)

```

<serial type='pty'>
  <source path='/dev/pts/2'/>
  <target port='0'/>
  <alias name='serial0'/>
</serial>
<console type='pty' tty='/dev/pts/2'>
  <source path='/dev/pts/2'/>
  <target type='serial' port='0'/>
  <alias name='serial0'/>
</console>
<input type='mouse' bus='ps2'/>
<input type='keyboard' bus='ps2'/>
<graphics type='vnc' port='5900' autoport='yes' listen='127.0.0.1'>
  <listen type='address' address='127.0.0.1'/>
</graphics>
<memballoon model='virtio'>
  <alias name='balloon0'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x08' function='0x0'/>
</memballoon>
</devices>
<seclabel type='dynamic' model='apparmor' relabel='yes'>
  <label>libvirt-2c4c9317-c7a5-4b37-b789-386ccda7348a</label>
  <imagelabel>libvirt-2c4c9317-c7a5-4b37-b789-386ccda7348a</imagelabel>
</seclabel>
</domain>

```

Raw HTML

An html frame can be included with the following construct. It is recommended that references to raw html be included with it's own .rst file.

Raw HTML Example

This example shows how to include include a CSIT performance graph.

Markdown Style Guide

Most of these documents are written using *reStructured Text Style Guide* (rst), but pages can also be written in Markdown. This chapter describes some constructs used to write these documents. For a more detailed description of Markdown refer to [Markdown Wikipedia](#)

Heading 1

This is the top heading level. More levels are shown below.

Heading 2

Heading 3

Heading 4

Heading 5

Bullets, Bold and Italics

Bold text can be show with **Bold Text**, Italics with *Italic text*. Bullets like so:

- Bullet 1
- Bullet 2

Code Blocks

This paragraph describes how to do **Console Commands**. When showing VPP commands it is reccomended that the command be executed from the linux console as shown. The Highlighting in the final documents shows up nicely this way.

```
$ sudo bash
# vppctl show interface
      Name                Idx      State      Counter      Count
TenGigabitEthernet86/0/0    1        up        rx packets    6569213
                          rx bytes    9928352943
                          tx packets    50384
                          tx bytes    3329279
TenGigabitEthernet86/0/1    2        down
VirtualEthernet0/0/0        3        up        rx packets    50384
                          rx bytes    3329279
                          tx packets    6569213
                          tx bytes    9928352943
                          drops      1498
local0                      0        down
#
```

The **code-block** construct is also used for code samples. The following shows how to include a block of “C” code.

```
#include <vlib/unix/unix.h>
abf_policy_t *
abf_policy_get (u32 index)
{
    return (pool_elt_at_index (abf_policy_pool, index));
}
```

Diffs are generated in the final docs nicely with “:” at the end of the description like so:

```
diff --git a/src/vpp/vnet/main.c b/src/vpp/vnet/main.c
index 6e136e19..69189c93 100644
--- a/src/vpp/vnet/main.c
+++ b/src/vpp/vnet/main.c
@@ -18,6 +18,8 @@
```

(continues on next page)

(continued from previous page)

```
#include <vlib/unix/unix.h>
#include <vnet/plugin/plugin.h>
#include <vnet/ethernet/ethernet.h>
+#include <vnet/ip/ip4_packet.h>
+#include <vnet/ip/format.h>
#include <vpp/app/version.h>
#include <vpp/api/vpe_msg_enum.h>
#include <limits.h>
@@ -400,6 +402,63 @@ VLIB_CLI_COMMAND (test_crash_command, static) = {

#endif
```

Labels, References

A link or reference to other paragraphs within these documents can be done with following construct.

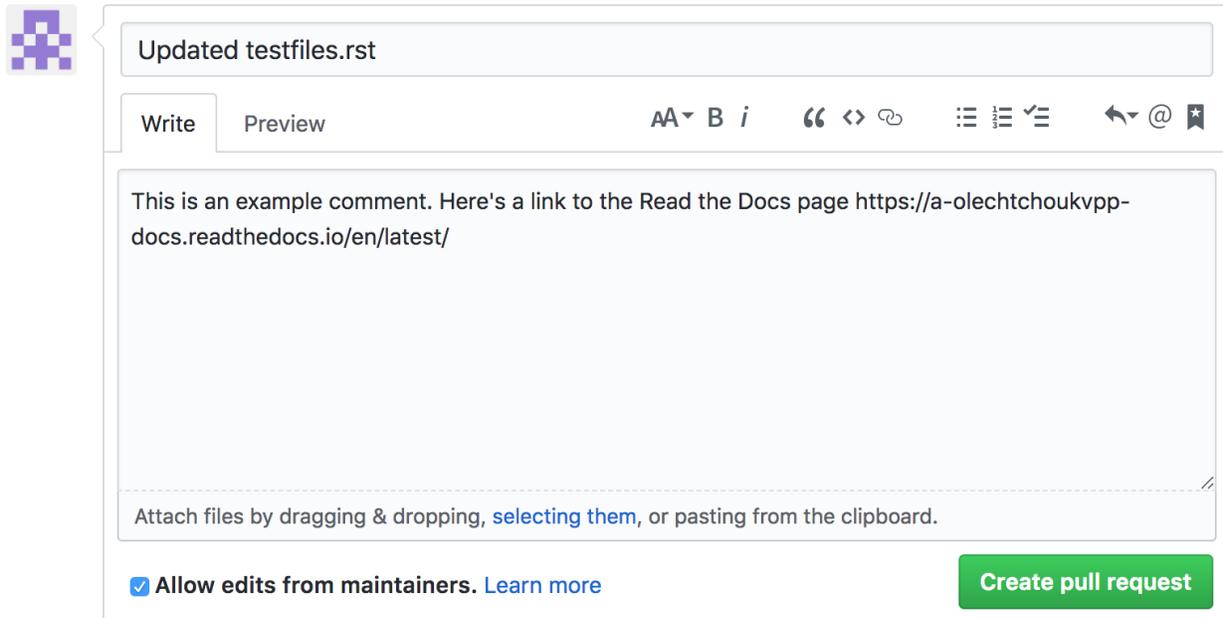
External Links

An external link is done with the following construct:

Sphinx Markup Constructs

Images

Images should be placed in the directory docs/_images. They can then be referenced with following construct. This is the image created to show a pull request.



2.1.3 How to Report an Issue

Reporting Bugs

Although every situation is different, this page describes how to collect data which will help make efficient use of everyone's time when dealing with vpp bugs.

Before you press the Jira button to create a bug report - or email vpp-dev@lists.fd.io - please ask yourself whether there's enough information for someone else to understand and possibly to reproduce the issue given a reasonable amount of effort. **Unicast emails to maintainers, committers, and the project PTL are strongly discouraged.**

A good strategy for clear-cut bugs: file a detailed Jira ticket, and then send a short description of the issue to vpp-dev@lists.fd.io, perhaps from the Jira ticket description. It's fine to send email to vpp-dev@lists.fd.io to ask a few questions **before** filing Jira tickets.

Data to include in bug reports

Image version and operating environment

Please make sure to include the vpp image version.

```
$ sudo bash
# vppctl show version verbose

vpp v1.0.0-188-geef4d99 built by vagrant on localhost at Wed Feb 24 08:52:13 PST 2016
Built in /home/vagrant/git/vpp
Compiled with GCC 4.8.4
DPDK version is RTE 2.2.0
DPDK EAL init arguments: -c 1 -n 4 --socket-mem 1024 --huge-dir /run/vpp/hugepages
--file-prefix vpp -b 0000:02:00.0 -b 0000:02:01.0 --master-lcore 0
```

With respect to the operating environment: if misbehavior involving a specific VM / container / bare-metal environment is involved, please describe the environment in detail:

- Linux Distro (e.g. Ubuntu 14.04.3 LTS, CentOS-7, etc.)
- NIC type(s) (ixgbe, i40e, enic, etc. etc.), vhost-user, tuntap
- NUMA configuration if applicable

Please note the CPU architecture (x86_86, aarch64), and hardware platform.

When practicable, please report issues against released software, or unmodified master/latest software.

“Show” command output

Every situation is different. If the issue involves a sequence of debug CLI command, please enable CLI command logging, and send the sequence involved. Note that the debug CLI is a developer's tool - **no warranty express or implied** - and that we may choose not to fix debug CLI bugs.

Please include “show error” [error counter] output. It's often helpful to “clear error”, send a bit of traffic, then “show error” particularly when running vpp on a noisy networks.

Please include ip4 / ip6 / mpls FIB contents (“show ip fib”, “show ip6 fib”, “show mpls fib”, “show mpls tunnel”).

Please include “show hardware”, “show interface”, and “show interface address” output

Here is a consolidated set of commands that are generally useful before/after sending traffic. Before sending traffic.

```
vppctl clear hardware
vppctl clear interface
vppctl clear error
vppctl clear run
```

Send some traffic and then issue the following commands.

```
vppctl show version verbose
vppctl show hardware
vppctl show hardware address
vppctl show interface
vppctl show run
vppctl show error
```

Here are some protocol specific show commands that may also make sense. Only include those features which have been configured.

```
vppctl show l2fib
vppctl show bridge-domain

vppctl show ip fib
vppctl show ip arp

vppctl show ip6 fib
vppctl show ip6 neighbors

vppctl show mpls fib
vppctl show mpls tunnel
```

Network Topology

Please include a crisp description of the network topology, including L2 / IP / MPLS / segment-routing addressing details. If you expect folks to reproduce and debug issues, this is a must.

At or above a certain level of topological complexity, it becomes problematic to reproduce the original setup.

Packet Tracer Output

If you capture packet tracer output which seems relevant, please include it.

```
vppctl trace add dpdk-input 100 # or similar
```

send-traffic

```
vppctl show trace
```

Capturing post-mortem data

It should go without saying, but anyhow: **please put post-mortem data in obvious, accessible places.** Time wasted trying to acquire accounts, credentials, and IP addresses simply delays problem resolution.

Please remember to add post-mortem data location information to Jira tickets.

Syslog Output

The vpp signal handler typically writes a certain amount of data in `/var/log/syslog` before exiting. Make sure to check for evidence, e.g via “`grep /usr/bin/vpp /var/log/syslog`” or similar.

Binary API Trace

If the issue involves a sequence of control-plane API messages - even a very long sequence - please enable control-plane API tracing. Control-plane API post-mortem traces end up in `/tmp/api_post_mortem.<pid>`.

Please remember to put post-mortem binary api traces in accessible places.

These API traces are especially helpful in cases where the vpp engine is throwing traffic on the floor, e.g. for want of a default route or similar.

Make sure to leave the default stanza “... `api-trace { on } ...`” in the vpp startup configuration file `/etc/vpp/startup.conf`, or to include it in the command line arguments passed by orchestration software.

Core Files

Production systems, as well as long-running pre-production soak-test systems, **must** arrange to collect core images. There are various ways to configure core image capture, including e.g. the Ubuntu “corekeeper” package. In a pinch, the following very basic sequence will capture usable vpp core files in `/tmp/dumps`.

```
# mkdir -p /tmp/dumps
# sysctl -w debug.exception-trace=1
# sysctl -w kernel.core_pattern="/tmp/dumps/%e-%t"
# ulimit -c unlimited
# echo 2 > /proc/sys/fs/suid_dumpable
```

Vpp core files often appear enormous. Gzip typically compresses them to manageable sizes. A multi-GByte corefile often compresses to 10-20 Mbytes.

Please remember to put compressed core files in accessible places.

Make sure to leave the default stanza “... `unix { ... full-coredump ... } ...`” in the vpp startup configuration file `/etc/vpp/startup.conf`, or to include it in the command line arguments passed by orchestration software.

Core files from private, modified images are discouraged. If it’s necessary to go that route, please copy the **exact** Debian packages (or RPMs) corresponding to the core file to the same public place as the core file. In particular.

- `vpp-<version>_<arch>.deb` # the vpp executable
- `vpp-dbg-<version>_<arch>.deb` # debug symbols
- `vpp-dev-<version>_<arch>.deb` # development package
- `vpp-lib-<version>_<arch>.deb` # shared libraries
- `vpp-plugins-<version>_<arch>.deb` # plugins

Please include the full commit-ID the Jira ticket.

If we go through the setup process only to discover that the image and core files don’t match, it will simply delay resolution of the issue. And it will annoy the heck out of the engineer who just wasted their time. Exact means **exact**, not “oh, gee, I added a few lines of debug scaffolding since then...”

2.2 Developers

This chapter will describe how developers can get started to with FD.io VPP. In this section I can see writing a VPP plugin, How to use the python api.

2.2.1 Building FD.io VPP

Once FD.io VPP has been installed, this getting started guide will be useful to aid you in building FD.io VPP.

Building Commands

The following are instructions on how to build FD.io VPP.

Building VPP Commands

Set up Proxies

Depending on the environment, proxies may need to be set. You may run these commands:

```
$ export http_proxy=http://<proxy-server-name>.com:<port-number>
$ export https_proxy=https://<proxy-server-name>.com:<port-number>
```

Build VPP Dependencies

Before building, make sure there are no FD.io VPP or DPDK packages installed by entering the following commands:

```
$ dpkg -l | grep vpp
$ dpkg -l | grep DPDK
```

There should be no output, or packages showing after each of the above commands.

Run this to install the dependencies for FD.io VPP. If it hangs during downloading at any point, you may need to set up *proxies for this to work*.

```
$ make install-dep
Hit:1 http://us.archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://us.archive.ubuntu.com/ubuntu xenial-updates InRelease [109 kB]
Get:3 http://security.ubuntu.com/ubuntu xenial-security InRelease [107 kB]
Get:4 http://us.archive.ubuntu.com/ubuntu xenial-backports InRelease [107 kB]
Get:5 http://us.archive.ubuntu.com/ubuntu xenial-updates/main amd64 Packages [803 kB]
Get:6 http://us.archive.ubuntu.com/ubuntu xenial-updates/main i386 Packages [732 kB]
...
...

Update-alternatives: using /usr/lib/jvm/java-8-openjdk-amd64/bin/jmap to provide /usr/
↳bin/jmap (jmap) in auto mode
Setting up default-jdk-headless (2:1.8-56ubuntu2) ...
Processing triggers for libc-bin (2.23-0ubuntu3) ...
Processing triggers for systemd (229-4ubuntu6) ...
Processing triggers for ureadahead (0.100.0-19) ...
Processing triggers for ca-certificates (20160104ubuntu1) ...
```

(continues on next page)

(continued from previous page)

```
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...

done.
done.
```

Build VPP (Debug Mode)

This build version contains debug symbols which is useful to modify VPP. The command below will build debug version of VPP. This build will come with /build-root/vpp_debug-native.

```
:: $ make build make[1]: Entering directory '/home/vagrant/vpp-master/build-root' @@@@ Arch for platform 'vpp' is native @@@@ @@@@ Finding source for dpdk @@@@ @@@@ Makefile fragment found in /home/vagrant/vpp-master/build-data/packages/dpdk.mk @@@@ @@@@ Source found in /home/vagrant/vpp-master/dpdk @@@@ @@@@ Arch for platform 'vpp' is native @@@@ @@@@ Finding source for vpp @@@@ @@@@ Makefile fragment found in /home/vagrant/vpp-master/build-data/packages/vpp.mk @@@@ @@@@ Source found in /home/vagrant/vpp-master/src @@@@ ... .. make[5]: Leaving directory '/home/vagrant/vpp-master/build-root/build-vpp_debug-native/vpp/vpp-api/java' make[4]: Leaving directory '/home/vagrant/vpp-master/build-root/build-vpp_debug-native/vpp/vpp-api/java' make[3]: Leaving directory '/home/vagrant/vpp-master/build-root/build-vpp_debug-native/vpp' make[2]: Leaving directory '/home/vagrant/vpp-master/build-root/build-vpp_debug-native/vpp' @@@@ Installing vpp: nothing to do @@@@ make[1]: Leaving directory '/home/vagrant/vpp-master/build-root'
```

Build VPP (Release Version)

To build the release version of FD.io VPP. This build is optimized and will not create debug symbols. This build will come with /build-root/build-vpp-native

```
$ make release
```

Building Necessary Packages

To build the debian packages, one of the following commands below depending on the system:

Building Debian Packages

```
$ make pkg-deb
```

Building RPM Packages

```
$ make pkg-rpm
```

Note: Please follow the commands that the Operating System prompts, after running one of the commands above

The packages will be found in the build-root directory.

```
$ ls *.deb
```

If packages built correctly, this should be the Output

```
vpp_18.07-rc0~456-gb361076_amd64.deb          vpp-dbg_18.07-rc0~456-gb361076_amd64.
↪deb
vpp-api-java_18.07-rc0~456-gb361076_amd64.deb  vpp-dev_18.07-rc0~456-gb361076_amd64.
↪deb
vpp-api-lua_18.07-rc0~456-gb361076_amd64.deb   vpp-lib_18.07-rc0~456-gb361076_amd64.
↪deb
vpp-api-python_18.07-rc0~456-gb361076_amd64.deb vpp-plugins_18.07-rc0~456-gb361076_
↪amd64.deb
```

Packages built installed end up in build-root directory. Finally, the command below installs all built packages.

```
$ sudo bash
# dpkg -i *.deb
```

Useful Directories

Useful Directories

After pulling and building FD.io VPP there are a few directories worth looking at. `src/vpp/conf`

This directory contains default configuration files.

```
# ls
80-vpp.conf  startup.conf
```

User Tools

This directory is provided with DPDK. The two import scripts are `cpu_layout.py` and `dpdk-devbind.py`

`build-root/build-vpp_debug-native/dpdk/dpdk-17.02/usertools/`

```
# ls ./build-root/build-vpp_debug-native/dpdk/dpdk-17.02/usertools/
cpu_layout.py  dpdk-devbind.py  dpdk-pmdinfo.py  dpdk-setup.sh
```

VPP/bin

`build-root/install-vpp_debug-native/vpp/bin/`

- This directory contains the vpp executables.
- The most useful files are `vpp` and `vppctl`.
- These files are copied to `/usr/bin` after FDIO is installed.
- You can use the binary file “`vpp`” located in this directory with `gdb` to help debug FDIO.

```
root@tf-ucs-3# ls ./build-root/install-vpp_debug-native/vpp/bin
elftool svmbdtool svmtool vpp vppapigen vpp_api_test vppctl vpp_get_metrics _
↪vpp_json_test vpp_restart
```

Devbind

dpdk-devbind.py

- The dpdk-devbind.py script is provided with the Intel DPDK.
- It is included with FD.io VPP.
- After FD.io VPP is built, this script and other DPDK tools can be found in build-root/build-vpp_debug-native/dpdk/dpdk-17.02/usertools/.

vNet

src/scripts/vnet/

This directory has some very useful examples using the FDIO traffic generator and general configuration.

```
# ls src/scripts/vnet/
arp4      dhcp  ip6          l2efpfilter_perf  l2flood  mcast          pcap
↪rightpeer snat_det
```

src/vnet/

This directory contains most of the important source code.

```
# ls src/vnet
adj          config.h  fib          interface.api      interface_output.c  lawful-
↪intercept  misc.c   ppp
```

src/vnet/devices/

This directory contains the device drivers. For example, the vhost driver is in src/vnet/devices/virtio.

```
# ls src/vnet/devices/virtio/
dir.dox  vhost_user.api  vhost_user_api.c  vhost-user.c  vhost-user.h
```

2.2.2 Overview

Describe the software architecture here.

Software Architecture

Note: Add Overview Section.

The fd.io vpp implementation is a third-generation vector packet processing implementation specifically related to US Patent 7,961,636, as well as earlier work. Note that the Apache-2 license specifically grants non-exclusive patent licenses; we mention this patent as a point of historical interest.

For performance, the vpp dataplane consists of a directed graph of forwarding nodes which process multiple packets per invocation. This schema enables a variety of micro-processor optimizations: pipelining and prefetching to cover dependent read latency, inherent I-cache phase behavior, vector instructions. Aside from hardware input and hardware output nodes, the entire forwarding graph is portable code.

Depending on the scenario at hand, we often spin up multiple worker threads which process ingress-hashes packets from multiple queues using identical forwarding graph replicas.

Implementation taxonomy

The vpp dataplane consists of four distinct layers:

- An infrastructure layer comprising vppinfra, vlib, svm, and binary api libraries. See `.../src/{vppinfra, vlib, svm, vlibapi, vlibmemory}`
- A generic network stack layer: vnet. See `.../src/vnet`
- An application shell: vpp. See `.../src/vpp`
- An increasingly rich set of data-plane plugins: see `.../src/plugins`

It's important to understand each of these layers in a certain amount of detail. Much of the implementation is best dealt with at the API level and otherwise left alone.

Vppinfra

Vppinfra is a collection of basic c-library services, quite sufficient to build standalone programs to run directly on bare metal. It also provides high-performance dynamic arrays, hashes, bitmaps, high-precision real-time clock support, fine-grained event-logging, and data structure serialization.

One fair comment / fair warning about vppinfra: you can't always tell a macro from an inline function from an ordinary function simply by name. Macros are used to avoid function calls in the typical case, and to cause (intentional) side-effects.

Vppinfra has been around for almost 20 years and tends not to change frequently.

Vectors

Vppinfra vectors are ubiquitous dynamically resized arrays with by user defined "headers". Many vppinfra data structures (e.g. hash, heap, pool) are vectors with various different headers.

The memory layout looks like this:

```
User header (optional, uword aligned)
Alignment padding (if needed)
Vector length in elements
```

(continues on next page)

(continued from previous page)

```
User's pointer -> Vector element 0
                  Vector element 1
                  ...
                  Vector element N-1
```

As shown above, the vector APIs deal with pointers to the 0th element of a vector. Null pointers are valid vectors of length zero.

To avoid thrashing the memory allocator, one often resets the length of a vector to zero while retaining the memory allocation. Set the vector length field to zero via the `vec_reset_length(v)` macro. [Use the macro! It's smart about NULL pointers.]

Typically, the user header is not present. User headers allow for other data structures to be built atop vppinfra vectors. Users may specify the alignment for data elements via the `vec_*_aligned` macros.

Vectors elements can be any C type e.g. (int, double, struct bar). This is also true for data types built atop vectors (e.g. heap, pool, etc.). Many macros have `_a` variants supporting alignment of vector data and `_h` variants supporting non-zero-length vector headers. The `_ha` variants support both.

Inconsistent usage of header and/or alignment related macro variants will cause delayed, confusing failures.

Standard programming error: memorize a pointer to the *i*th element of a vector, and then expand the vector. Vectors expand by 3/2, so such code may appear to work for a period of time. Correct code almost always memorizes vector **indices** which are invariant across reallocations.

In typical application images, one supplies a set of global functions designed to be called from gdb. Here are a few examples:

- `vl(v)` - prints `vec_len(v)`
- `pe(p)` - prints `pool_elts(p)`
- `pifi(p, index)` - prints `pool_is_free_index(p, index)`
- `debug_hex_bytes(p, nbytes)` - hex memory dump `nbytes` starting at `p`

Use the “show gdb” debug CLI command to print the current set.

Bitmaps

Vppinfra bitmaps are dynamic, built using the vppinfra vector APIs. Quite handy for a variety of jobs.

Pools

Vppinfra pools combine vectors and bitmaps to rapidly allocate and free fixed-size data structures with independent lifetimes. Pools are perfect for allocating per-session structures.

Hashes

Vppinfra provides several hash flavors. Data plane problems involving packet classification / session lookup often use `.../src/vppinfra/bihash_template.[ch]` bounded-index extensible hashes. These templates are instantiated multiple times, to efficiently service different fixed-key sizes.

Bihashes are thread-safe. Read-locking is not required. A simple spin-lock ensures that only one thread writes an entry at a time.

The original vppinfra hash implementation in `.../src/vppinfra/hash.[ch]` are simple to use, and are often used in control-plane code which needs exact-string-matching.

In either case, one almost always looks up a key in a hash table to obtain an index in a related vector or pool. The APIs are simple enough, but one must take care when using the unmanaged arbitrary-sized key variant. `Hash_set_mem` (`hash_table`, `key_pointer`, `value`) memorizes `key_pointer`. It is usually a bad mistake to pass the address of a vector element as the second argument to `hash_set_mem`. It is perfectly fine to memorize constant string addresses in the text segment.

Format

Vppinfra format is roughly equivalent to `printf`.

Format has a few properties worth mentioning. Format's first argument is a (`u8 *`) vector to which it appends the result of the current format operation. Chaining calls is very easy:

```
u8 * result;

result = format (0, "junk = %d, ", junk);
result = format (result, "more junk = %d\n", more_junk);
```

As previously noted, NULL pointers are perfectly proper 0-length vectors. Format returns a (`u8 *`) vector, **not** a C-string. If you wish to print a (`u8 *`) vector, use the “%v” format string. If you need a (`u8 *`) vector which is also a proper C-string, either of these schemes may be used:

```
vec_addl (result, 0)
or
result = format (result, "<whatever>%c", 0);
```

Remember to `vec_free()` the result if appropriate. Be careful not to pass format an uninitialized `u8 *`.

Format implements a particularly handy user-format scheme via the “%U” format specification. For example:

```
u8 * format_junk (u8 * s, va_list *va)
{
    junk = va_arg (va, u32);
    s = format (s, "%s", junk);
    return s;
}

result = format (0, "junk = %U, format_junk, "This is some junk");
```

`format_junk()` can invoke other user-format functions if desired. The programmer shoulders responsibility for argument type-checking. It is typical for user format functions to blow up if the `va_arg(va, <type>)` macros don't match the caller's idea of reality.

Unformat

Vppinfra unformat is vaguely related to `scanf`, but considerably more general.

A typical use case involves initializing an `unformat_input_t` from either a C-string or a (`u8 *`) vector, then parsing via `unformat()` as follows:

```
unformat_input_t input;
```

(continues on next page)

(continued from previous page)

```
unformat_init_string (&input, "<some-C-string>");
/* or */
unformat_init_vector (&input, <u8-vector>);
```

Then loop parsing individual elements:

```
while (unformat_check_input (&input) != UNFORMAT_END_OF_INPUT)
{
    if (unformat (&input, "value1 %d", &value1))
        /* unformat sets value1 */
    else if (unformat (&input, "value2 %d", &value2))
        /* unformat sets value2 */
    else
        return clib_error_return (0, "unknown input '%U'", format_unformat_error,
                                   input);
}
```

As with format, unformat implements a user-unformat function capability via a “%U” user unformat function scheme.

Vppinfra errors and warnings

Many functions within the vpp dataplane have return-values of type `clib_error_t *`. `clib_error_t`'s are arbitrary strings with a bit of metadata [fatal, warning] and are easy to announce. Returning a NULL `clib_error_t *` indicates “A-OK, no error.”

`clib_warning(<format-args>)` is a handy way to add debugging output; `clib` warnings prepend function:line info to unambiguously locate the message source. `clib_unix_warning()` adds `perror()`-style Linux system-call information. In production images, `clib_warnings` result in `syslog` entries.

Serialization

Vppinfra serialization support allows the programmer to easily serialize and unserialize complex data structures.

The underlying primitive `serialize/unserialize` functions use network byte-order, so there are no structural issues serializing on a little-endian host and unserializing on a big-endian host.

Event-logger, graphical event log viewer

The vppinfra event logger provides very lightweight (sub-100ns) precisely time-stamped event-logging services. See `.../src/vppinfra/{elog.c, elog.h}`

Serialization support makes it easy to save and ultimately to combine a set of event logs. In a distributed system running NTP over a local LAN, we find that event logs collected from multiple system elements can be combined with a temporal uncertainty no worse than 50us.

A typical event definition and logging call looks like this:

```
ELOG_TYPE_DECLARE (e) =
{
    .format = "tx-msg: stream %d local seq %d attempt %d",
    .format_args = "i4i4i4",
};
struct { u32 stream_id, local_sequence, retry_count; } * ed;
```

(continues on next page)

(continued from previous page)

```
ed = ELOG_DATA (m->elog_main, e);
ed->stream_id = stream_id;
ed->local_sequence = local_sequence;
ed->retry_count = retry_count;
```

The ELOG_DATA macro returns a pointer to 20 bytes worth of arbitrary event data, to be formatted (offline, not at runtime) as described by format_args. Aside from obvious integer formats, the CLIB event logger provides a couple of interesting additions. The “t4” format pretty-prints enumerated values:

```
ELOG_TYPE_DECLARE (e) =
{
    .format = "get_or_create: %s",
    .format_args = "t4",
    .n_enum_strings = 2,
    .enum_strings = { "old", "new", },
};
```

The “t” format specifier indicates that the corresponding datum is an index in the event’s set of enumerated strings, as shown in the previous event type definition.

The “T” format specifier indicates that the corresponding datum is an index in the event log’s string heap. This allows the programmer to emit arbitrary formatted strings. One often combines this facility with a hash table to keep the event-log string heap from growing arbitrarily large.

Noting the 20-octet limit per-log-entry data field, the event log formatter supports arbitrary combinations of these data types. As in: the “.format” field may contain one or more instances of the following:

- i1 - 8-bit unsigned integer
- i2 - 16-bit unsigned integer
- i4 - 32-bit unsigned integer
- i8 - 64-bit unsigned integer
- f4 - float
- f8 - double
- s - NULL-terminated string - be careful
- sN - N-byte character array
- t1,2,4 - per-event enumeration ID
- T4 - Event-log string table offset

The vpp engine event log is thread-safe, and is shared by all threads. Take care not to serialize the computation. Although the event-logger is about as fast as practicable, it’s not appropriate for per-packet use in hard-core data plane code. It’s most appropriate for capturing rare events - link up-down events, specific control-plane events and so forth.

The vpp engine has several debug CLI commands for manipulating its event log:

```
vpp# event-logger clear
vpp# event-logger save <filename> # for security, writes into /tmp/<filename>.
                                # <filename> must not contain '.' or '/' characters
vpp# show event-logger [all] [<nnn>] # display the event log
                                # by default, the last 250 entries
```

The event log defaults to 128K entries. The command-line argument “... vlib { elog-events <nnn> }” configures the size of the event log.

As described above, the vpp engine event log is thread-safe and shared. To avoid confusing non-appearance of events logged by worker threads, make sure to code `&vlib_global_main.elog_main` - instead of `&vm->elog_main`. The latter form is correct in the main thread, but will almost certainly produce bad results in worker threads.

G2 graphical event viewer

The g2 graphical event viewer can display serialized vppinfra event logs directly, or via the `c2cpel` tool.

Note: Todo: please convert wiki page and figures

VLIB

Vlib provides vector processing support including graph-node scheduling, reliable multicast support, ultra-lightweight cooperative multi-tasking threads, a CLI, plug in .DLL support, physical memory and Linux `epoll` support. Parts of this library embody US Patent 7,961,636.

Init function discovery

vlib applications register for various [initialization] events by placing structures and `__attribute__((constructor))` functions into the image. At appropriate times, the vlib framework walks constructor-generated singly-linked structure lists, calling the indicated functions. vlib applications create graph nodes, add CLI functions, start cooperative multi-tasking threads, etc. etc. using this mechanism.

vlib applications invariably include a number of `VLIB_INIT_FUNCTION` (`my_init_function`) macros.

Each `init / configure / etc.` function has the return type `clib_error_t *`. Make sure that the function returns 0 if all is well, otherwise the framework will announce an error and exit.

vlib applications must link against vppinfra, and often link against other libraries such as VNET. In the latter case, it may be necessary to explicitly reference symbol(s) otherwise large portions of the library may be AWOL at runtime.

Node Graph Initialization

vlib packet-processing applications invariably define a set of graph nodes to process packets.

One constructs a `vlib_node_registration_t`, most often via the `VLIB_REGISTER_NODE` macro. At runtime, the framework processes the set of such registrations into a directed graph. It is easy enough to add nodes to the graph at runtime. The framework does not support removing nodes.

vlib provides several types of vector-processing graph nodes, primarily to control framework dispatch behaviors. The type member of the `vlib_node_registration_t` functions as follows:

- `VLIB_NODE_TYPE_PRE_INPUT` - run before all other node types
- `VLIB_NODE_TYPE_INPUT` - run as often as possible, after `pre_input` nodes
- `VLIB_NODE_TYPE_INTERNAL` - only when explicitly made runnable by adding pending frames for processing
- `VLIB_NODE_TYPE_PROCESS` - only when explicitly made runnable. "Process" nodes are actually cooperative multi-tasking threads. They **must** explicitly suspend after a reasonably short period of time.

For a precise understanding of the graph node dispatcher, please read `.../src/vlib/main.c:vlib_main_loop`.

Graph node dispatcher

Vlib_main_loop() dispatches graph nodes. The basic vector processing algorithm is diabolically simple, but may not be obvious from even a long stare at the code. Here's how it works: some input node, or set of input nodes, produce a vector of work to process. The graph node dispatcher pushes the work vector through the directed graph, subdividing it as needed, until the original work vector has been completely processed. At that point, the process recurs.

This scheme yields a stable equilibrium in frame size, by construction. Here's why: as the frame size increases, the per-frame-element processing time decreases. There are several related forces at work; the simplest to describe is the effect of vector processing on the CPU L1 I-cache. The first frame element [packet] processed by a given node warms up the node dispatch function in the L1 I-cache. All subsequent frame elements profit. As we increase the number of frame elements, the cost per element goes down.

Under light load, it is a crazy waste of CPU cycles to run the graph node dispatcher flat-out. So, the graph node dispatcher arranges to wait for work by sitting in a timed epoll wait if the prevailing frame size is low. The scheme has a certain amount of hysteresis to avoid constantly toggling back and forth between interrupt and polling mode. Although the graph dispatcher supports interrupt and polling modes, our current default device drivers do not.

The graph node scheduler uses a hierarchical timer wheel to reschedule process nodes upon timer expiration.

Process / thread model

vlib provides an ultra-lightweight cooperative multi-tasking thread model. The graph node scheduler invokes these processes in much the same way as traditional vector-processing run-to-completion graph nodes; plus-or-minus a setjmp/longjmp pair required to switch stacks. Simply set the vlib_node_registration_t type field to vlib_NODE_TYPE_PROCESS. Yes, process is a misnomer. These are cooperative multi-tasking threads.

As of this writing, the default stack size is 2^{15} ; 32kb. Initialize the node registration's process_log2_n_stack_bytes member as needed. The graph node dispatcher makes some effort to detect stack overrun, e.g. by mapping a no-access page below each thread stack.

Process node dispatch functions are expected to be "while(1) { }" loops which suspend when not otherwise occupied, and which must not run for unreasonably long periods of time.

"Unreasonably long" is an application-dependent concept. Over the years, we have constructed frame-size sensitive control-plane nodes which will use a much higher fraction of the available CPU bandwidth when the frame size is low. The classic example: modifying forwarding tables. So long as the table-builder leaves the forwarding tables in a valid state, one can suspend the table builder to avoid dropping packets as a result of control-plane activity.

Process nodes can suspend for fixed amounts of time, or until another entity signals an event, or both. See the next section for a description of the vlib process event mechanism.

When running in vlib process context, one must pay strict attention to loop invariant issues. If one walks a data structure and calls a function which may suspend, one had best know by construction that it cannot change. Often, it's best to simply make a snapshot copy of a data structure, walk the copy at leisure, then free the copy.

Process events

The vlib process event mechanism API is extremely lightweight and easy to use. Here is a typical example:

```
vlib_main_t *vm = &vlib_global_main;
uword event_type, * event_data = 0;

while (1)
{
```

(continues on next page)

(continued from previous page)

```

vlib_process_wait_for_event_or_clock (vm, 5.0 /* seconds */);

event_type = vlib_process_get_events (vm, &event_data);

switch (event_type) {
case EVENT1:
    handle_event1s (event_data);
    break;

case EVENT2:
    handle_event2s (event_data);
    break;

case ~0: /* 5-second idle/periodic */
    handle_idle ();
    break;

default: /* bug! */
    ASSERT (0);
}

vec_reset_length(event_data);
}

```

In this example, the VLIB process node waits for an event to occur, or for 5 seconds to elapse. The code demuxes on the event type, calling the appropriate handler function. Each call to `vlib_process_get_events` returns a vector of per-event-type data passed to successive `vlib_process_signal_event` calls; `vec_len (event_data) >= 1`.

It is an error to process only `event_data[0]`.

Resetting the `event_data` vector-length to 0 [instead of calling `vec_free`] means that the event scheme doesn't burn cycles continuously allocating and freeing the event data vector. This is a common vppinfra / vlib coding pattern, well worth using when appropriate.

Signaling an event is easy, for example:

```

vlib_process_signal_event (vm, process_node_index, EVENT1,
    (uword)arbitrary_event1_data); /* and so forth */

```

One can either know the process node index by construction - dig it out of the appropriate `vlib_node_registration_t` - or by finding the `vlib_node_t` with `vlib_get_node_by_name(...)`.

Buffers

vlib buffering solves the usual set of packet-processing problems, albeit at high performance. Key in terms of performance: one ordinarily allocates / frees N buffers at a time rather than one at a time. Except when operating directly on a specific buffer, one deals with buffers by index, not by pointer.

Packet-processing frames are effectively `u32[]`, not `vlib_buffer_t[]`.

Packets comprise one or more vlib buffers, chained together as required. Multiple particle sizes are supported; hardware input nodes simply ask for the required size(s). Coalescing support is available. For obvious reasons one is discouraged from writing one's own wild and wacky buffer chain traversal code.

vlib buffer headers are allocated immediately prior to the buffer data area. In typical packet processing this saves a dependent read wait: given a buffer's address, one can prefetch the buffer header [metadata] at the same time as the first cache line of buffer data.

Buffer header metadata (`vlib_buffer_t`) includes the usual rewrite expansion space, a `current_data` offset, RX and TX interface indices, packet trace information, and a opaque areas.

The opaque data is intended to control packet processing in arbitrary subgraph-dependent ways. The programmer shoulders responsibility for data lifetime analysis, type-checking, etc.

Buffers have reference-counts in support of e.g. multicast replication.

Shared-memory message API

Local control-plane and application processes interact with the vpp dataplane via asynchronous message-passing in shared memory over unidirectional queues. The same application APIs are available via sockets.

Capturing API traces and replaying them in a simulation environment requires a disciplined approach to the problem. This seems like a make-work task, but it is not. When something goes wrong in the control-plane after 300,000 or 3,000,000 operations, high-speed replay of the events leading up to the accident is a huge win.

The shared-memory message API message allocator `vl_api_msg_alloc` uses a particularly cute trick. Since messages are processed in order, we try to allocate message buffering from a set of fixed-size, preallocated rings. Each ring item has a “busy” bit. Freeing one of the preallocated message buffers merely requires the message consumer to clear the busy bit. No locking required.

Plug-ins

vlib implements a straightforward plug-in DLL mechanism. VLIB client applications specify a directory to search for plug-in .DLLs, and a name filter to apply (if desired). VLIB needs to load plug-ins very early.

Once loaded, the plug-in DLL mechanism uses `dlsym` to find and verify a `vlib_plugin_registration` data structure in the newly-loaded plug-in.

Debug CLI

Adding debug CLI commands to VLIB applications is very simple.

Here is a complete example:

```
static clib_error_t *
show_ip_tuple_match (vlib_main_t * vm,
                    unformat_input_t * input,
                    vlib_cli_command_t * cmd)
{
    vlib_cli_output (vm, "%U\n", format_ip_tuple_match_tables, &routing_main);
    return 0;
}

static VLIB_CLI_COMMAND (show_ip_tuple_command) = {
    .path = "show ip tuple match",
    .short_help = "Show ip 5-tuple match-and-broadcast tables",
    .function = show_ip_tuple_match,
};
```

This example implements the “show ip tuple match” debug cli command. In ordinary usage, the vlib cli is available via the “vppctl” application, which sends traffic to a named pipe. One can configure debug CLI telnet access on a configurable port.

The cli implementation has an output redirection facility which makes it simple to deliver cli output via shared-memory API messaging,

Particularly for debug or “show tech support” type commands, it would be wasteful to write vlib application code to pack binary data, write more code elsewhere to unpack the data and finally print the answer. If a certain cli command has the potential to hurt packet processing performance by running for too long, do the work incrementally in a process node. The client can wait.

Packet tracer

Vlib includes a frame element [packet] trace facility, with a simple vlib cli interface. The cli is straightforward: “trace add <input-node-name> <count>”.

To trace 100 packets on a typical x86_64 system running the dpdk plugin: “trace add dpdk-input 100”. When using the packet generator: “trace add pg-input 100”

Each graph node has the opportunity to capture its own trace data. It is almost always a good idea to do so. The trace capture APIs are simple.

The packet capture APIs snapshot binary data, to minimize processing at capture time. Each participating graph node initialization provides a vppinfra format-style user function to pretty-print data when required by the VLIB “show trace” command.

Set the VLIB node registration “.format_trace” member to the name of the per-graph node format function.

Here’s a simple example:

```
u8 * my_node_format_trace (u8 * s, va_list * args)
{
    vlib_main_t * vm = va_arg (*args, vlib_main_t *);
    vlib_node_t * node = va_arg (*args, vlib_node_t *);
    my_node_trace_t * t = va_arg (*args, my_node_trace_t *);

    s = format (s, "My trace data was: %d", t-><whatever>);

    return s;
}
```

The trace framework hands the per-node format function the data it captured as the packet whizzed by. The format function pretty-prints the data as desired.

Vnet

The vnet library provides vectorized layer-2 and 3 networking graph nodes, a packet generator, and a packet tracer.

In terms of building a packet processing application, vnet provides a platform-independent subgraph to which one connects a couple of device-driver nodes.

Typical RX connections include “ethernet-input” [full software classification, feeds ipv4-input, ipv6-input, arp-input etc.] and “ipv4-input-no-checksum” [if hardware can classify, perform ipv4 header checksum].

Effective graph dispatch function coding

Over the 15 years, two distinct styles have emerged: a single/dual/quad loop coding model and a fully-pipelined coding model. We seldom use the fully-pipelined coding model, so we won’t describe it in any detail

Single/dual loops

The single/dual/quad loop model is the only way to conveniently solve problems where the number of items to process is not known in advance: typical hardware RX-ring processing. This coding style is also very effective when a given node will not need to cover a complex set of dependent reads.

Feature Arcs

A significant number of vpp features are configurable on a per-interface or per-system basis. Rather than ask feature coders to manually construct the required graph arcs, we built a general mechanism to manage these mechanics.

Specifically, feature arcs comprise ordered sets of graph nodes. Each feature node in an arc is independently controlled. Feature arc nodes are generally unaware of each other. Handing a packet to “the next feature node” is quite inexpensive.

The feature arc implementation solves the problem of creating graph arcs used for steering.

At the beginning of a feature arc, a bit of setup work is needed, but only if at least one feature is enabled on the arc.

On a per-arc basis, individual feature definitions create a set of ordering dependencies. Feature infrastructure performs a topological sort of the ordering dependencies, to determine the actual feature order. Missing dependencies **will** lead to runtime disorder. See <https://gerrit.fd.io/r/#/c/12753> for an example.

If no partial order exists, vpp will refuse to run. Circular dependency loops of the form “a then b, b then c, c then a” are impossible to satisfy.

Adding a feature to an existing feature arc

To nobody’s great surprise, we set up feature arcs using the typical “macro -> constructor function -> list of declarations” pattern:

```
VNET_FEATURE_INIT (mactime, static) =
{
    .arc_name = "device-input",
    .node_name = "mactime",
    .runs_before = VNET_FEATURES ("ethernet-input"),
};
```

This creates a “mactime” feature on the “device-input” arc.

Once per frame, dig up the `vnet_feature_config_main_t` corresponding to the “device-input” feature arc:

```
vnet_main_t *vnm = vnet_get_main ();
vnet_interface_main_t *im = &vnm->interface_main;
u8 arc = im->output_feature_arc_index;
vnet_feature_config_main_t *fcm;

fcm = vnet_feature_get_config_main (arc);
```

Note that in this case, we’ve stored the required arc index - assigned by the feature infrastructure - in the `vnet_interface_main_t`. Where to put the arc index is a programmer’s decision when creating a feature arc.

Per packet, set `next0` to steer packets to the next node they should visit:

```
vnet_get_config_data (&fcm->config_main,
                    &b0->current_config_index /* value-result */,
                    &next0, 0 /* # bytes of config data */);
```

Configuration data is per-feature arc, and is often unused. Note that it's normal to reset next0 to divert packets elsewhere; often, to drop them for cause:

```
next0 = MACTIME_NEXT_DROP;
b0->error = node->errors[DROP_CAUSE];
```

Creating a feature arc

Once again, we create feature arcs using constructor macros:

```
VNET_FEATURE_ARC_INIT (ip4_unicast, static) =
{
    .arc_name = "ip4-unicast",
    .start_nodes = VNET_FEATURES ("ip4-input", "ip4-input-no-checksum"),
    .arc_index_ptr = &ip4_main.lookup_main.ucast_feature_arc_index,
};
```

In this case, we configure two arc start nodes to handle the “hardware-verified ip checksum or not” cases. During initialization, the feature infrastructure stores the arc index as shown.

In the head-of-arc node, do the following to send packets along the feature arc:

```
ip_lookup_main_t *lm = &im->lookup_main;
arc = lm->ucast_feature_arc_index;
```

Once per packet, initialize packet metadata to walk the feature arc:

```
vnet_feature_arc_start (arc, sw_if_index0, &next, b0);
```

Enabling / Disabling features

Simply call `vnet_feature_enable_disable` to enable or disable a specific feature:

```
vnet_feature_enable_disable ("device-input", /* arc name */
                            "mactime",      /* feature name */
                            sw_if_index,    /* Interface sw_if_index */
                            enable_disable, /* 1 => enable */
                            0 /* (void *) feature_configuration */,
                            0 /* feature_configuration_nbytes */);
```

The `feature_configuration` opaque is seldom used.

If you wish to make a feature a *de facto* system-level concept, pass `sw_if_index=0` at all times. `Sw_if_index 0` is always valid, and corresponds to the “local” interface.

Related “show” commands

To display the entire set of features, use “show features [verbose]”. The verbose form displays arc indices, and feature indicies within the arcs

```
$ vppctl show features verbose
Available feature paths
<snip>
```

(continues on next page)

(continued from previous page)

```

[14] ip4-unicast:
 [ 0]: nat64-out2in-handoff
 [ 1]: nat64-out2in
 [ 2]: nat44-ed-hairpin-dst
 [ 3]: nat44-hairpin-dst
 [ 4]: ip4-dhcp-client-detect
 [ 5]: nat44-out2in-fast
 [ 6]: nat44-in2out-fast
 [ 7]: nat44-handoff-classify
 [ 8]: nat44-out2in-worker-handoff
 [ 9]: nat44-in2out-worker-handoff
[10]: nat44-ed-classify
[11]: nat44-ed-out2in
[12]: nat44-ed-in2out
[13]: nat44-det-classify
[14]: nat44-det-out2in
[15]: nat44-det-in2out
[16]: nat44-classify
[17]: nat44-out2in
[18]: nat44-in2out
[19]: ip4-qos-record
[20]: ip4-vxlan-gpe-bypass
[21]: ip4-reassembly-feature
[22]: ip4-not-enabled
[23]: ip4-source-and-port-range-check-rx
[24]: ip4-flow-classify
[25]: ip4-inacl
[26]: ip4-source-check-via-rx
[27]: ip4-source-check-via-any
[28]: ip4-policer-classify
[29]: ipsec-input-ip4
[30]: vpath-input-ip4
[31]: ip4-vxlan-bypass
[32]: ip4-lookup
<snip>

```

Here, we learn that the `ip4-unicast` feature arc has index 14, and that e.g. `ip4-inacl` is the 25th feature in the generated partial order.

To display the features currently active on a specific interface, use “show interface features”:

```

$ vvpctl show interface GigabitEthernet3/0/0 features
Feature paths configured on GigabitEthernet3/0/0...
<snip>
ip4-unicast:
  nat44-out2in
<snip>

```

Table of Feature Arcs

Simply search for name-strings to track down the arc definition, location of the arc index, etc.

Arc Name
<code>device-input</code>

(continues on next page)

(continued from previous page)

ethernet-output	
interface-output	
ip4-drop	
ip4-local	
ip4-multicast	
ip4-output	
ip4-punt	
ip4-unicast	
ip6-drop	
ip6-local	
ip6-multicast	
ip6-output	
ip6-punt	
ip6-unicast	
mpls-input	
mpls-output	
nsh-output	

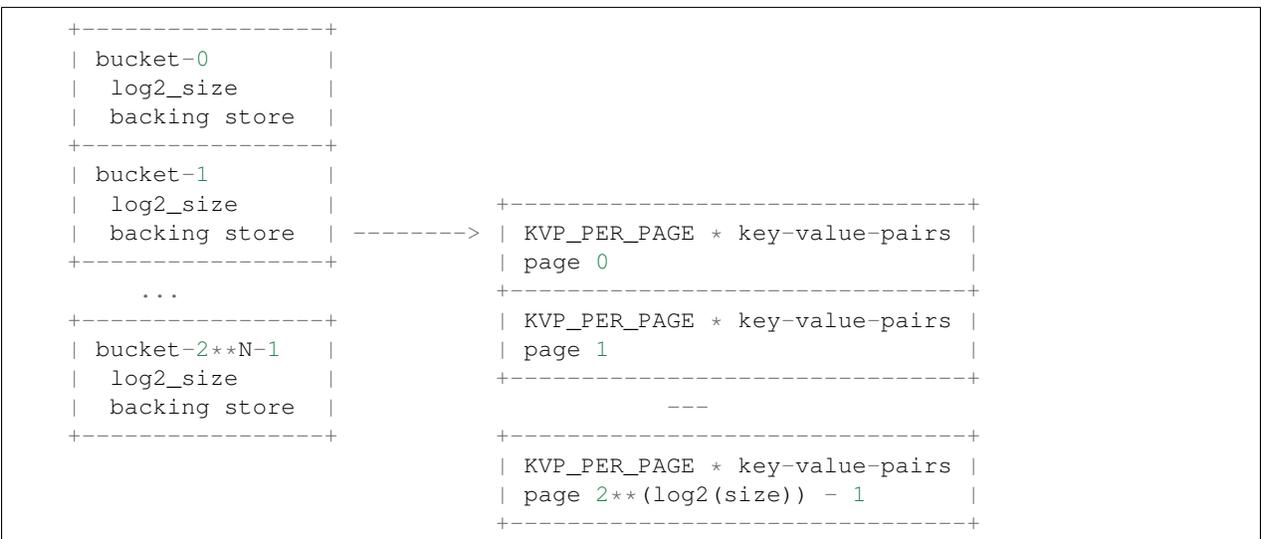
Bounded-index Extensible Hashing

Vpp uses bounded-index extensible hashing to solve a variety of exact-match (key, value) lookup problems. Benefits of the current implementation:

- Very high record count scaling, tested to 100,000,000 records.
- Lookup performance degrades gracefully as the number of records increases
- No reader locking required
- Template implementation, it's easy to support arbitrary (key,value) types

Bounded-index extensible hashing has been widely used in databases for decades.

Bihash uses a two-level data structure:



Discussion of the algorithm

This structure has a couple of major advantages. In practice, each bucket entry fits into a 64-bit integer. Coincidentally, vpp's target CPU architectures support 64-bit atomic operations. When modifying the contents of a specific bucket, we do the following:

- Make a working copy of the bucket's backing storage
- Atomically swap a pointer to the working copy into the bucket array
- Change the original backing store data
- Atomically swap back to the original

So, no reader locking is required to search a bihash table.

At lookup time, the implementation computes a key hash code. We use the least-significant N bits of the hash to select the bucket.

With the bucket in hand, we learn $\log_2(\text{nBackingPages})$ for the selected bucket. At this point, we use the next \log_2_size bits from the hash code to select the specific backing page in which the (key,value) page will be found.

Net result: we search **one** backing page, not $2^{\log_2_size}$ pages. This is a key property of the algorithm.

When sufficient collisions occur to fill the backing pages for a given bucket, we double the bucket size, rehash, and deal the bucket contents into a double-sized set of backing pages. In the future, we may represent the size as a linear combination of two powers-of-two, to increase space efficiency.

To solve the “jackpot case” where a set of records collide under hashing in a bad way, the implementation will fall back to linear search across $2^{\log_2_size}$ backing pages on a per-bucket basis.

To maintain *space* efficiency, we should configure the bucket array so that backing pages are effectively utilized. Lookup performance tends to change *very little* if the bucket array is too small or too large.

Bihash depends on selecting an effective hash function. If one were to use a truly broken hash function such as “return IULL.” bihash would still work, but it would be equivalent to poorly-programmed linear search.

We often use cpu intrinsic functions - think `crc32` - to rapidly compute a hash code which has decent statistics.

Bihash Cookbook

Using current (key,value) template instance types

It's quite easy to use one of the template instance types. As of this writing, `.../src/vppinfra` provides pre-built templates for 8, 16, 20, 24, 40, and 48 byte keys, `u8 * vector` keys, and 8 byte values.

See `.../src/vppinfra/{bihash_<key-size>_8}.h`

To define the data types, `#include` a specific template instance, most often in a subsystem header file:

```
#include <vppinfra/bihash_8_8.h>
```

If you're building a standalone application, you'll need to define the various functions by `#including` the method implementation file in a C source file.

The core vpp engine currently uses most if not all of the known bihash types, so you probably won't need to `#include` the method implementation file.

```
#include <vppinfra/bihash_template.c>
```

Add an instance of the selected bihash data structure to e.g. a “`main_t`” structure:

```
typedef struct
{
    ...
    BVT (clib_bihash) hash;
    or
    clib_bihash_8_8_t hash;
    ...
} my_main_t;
```

The BV macro concatenate its argument with the value of the preprocessor symbol BIHASH_TYPE. The BVT macro concatenates its argument with the value of BIHASH_TYPE and the fixed-string “_t”. So in the above example, BVT (clib_bihash) generates “clib_bihash_8_8_t”.

If you’re sure you won’t decide to change the template / type name later, it’s perfectly OK to code “clib_bihash_8_8_t” and so forth.

In fact, if you #include multiple template instances in a single source file, you **must** use fully-enumerated type names. The macros stand no chance of working.

Initializing a bihash table

Call the init function as shown. As a rough guide, pick a number of buckets which is approximately number_of_expected_records/BIHASH_KVP_PER_PAGE from the relevant template instance header-file. See previous discussion.

The amount of memory selected should easily contain all of the records, with a generous allowance for hash collisions. Bihash memory is allocated separately from the main heap, and won’t cost anything except kernel PTE’s until touched, so it’s OK to be reasonably generous.

For example:

```
my_main_t *mm = &my_main;
clib_bihash_8_8_t *h;

h = &mm->hash_table;

clib_bihash_init_8_8 (h, "test", (u32) number_of_buckets,
                    (uword) memory_size);
```

Add or delete a key/value pair

Use BV(clib_bihash_add_del), or the explicit type variant:

```
clib_bihash_kv_8_8_t kv;
clib_bihash_8_8_t * h;
my_main_t *mm = &my_main;
clib_bihash_8_8_t *h;

h = &mm->hash_table;
kv.key = key_to_add_or_delete;
kv.value = value_to_add_or_delete;

clib_bihash_add_del_8_8 (h, &kv, is_add /* 1=add, 0=delete */);
```

In the delete case, kv.value is irrelevant. To change the value associated with an existing (key,value) pair, simply re-add the [new] pair.

Simple search

The simplest possible (key, value) search goes like so:

```

clib_bihash_kv_8_8_t search_kv, return_kv;
clib_bihash_8_8_t * h;
my_main_t *mm = &my_main;
clib_bihash_8_8_t *h;

h = &mm->hash_table;
search_kv.key = key_to_add_or_delete;

if (clib_bihash_search_8_8 (h, &search_kv, &return_kv) < 0)
    key_not_found()
else
    key_not_found();

```

Note that it's perfectly fine to collect the lookup result

```

if (clib_bihash_search_8_8 (h, &search_kv, &search_kv))
    key_not_found();
etc.

```

Bihash vector processing

When processing a vector of packets which need a certain lookup performed, it's worth the trouble to compute the key hash, and prefetch the correct bucket ahead of time.

Here's a sketch of one way to write the required code:

Dual-loop:

- 6 packets ahead, prefetch 2x vlib_buffer_t's and 2x packet data required to form the record keys
- 4 packets ahead, form 2x record keys and call BV(clib_bihash_hash) or the explicit hash function to calculate the record hashes. Call 2x BV(clib_bihash_prefetch_bucket) to prefetch the buckets
- 2 packets ahead, call 2x BV(clib_bihash_prefetch_data) to prefetch 2x (key,value) data pages.
- In the processing section, call 2x BV(clib_bihash_search_inline_with_hash) to perform the search

Programmer's choice whether to stash the hash code somewhere in vnet_buffer(b) metadata, or to use local variables.

Single-loop:

- Use simple search as shown above.

Walking a bihash table

A fairly common scenario to build "show" commands involves walking a bihash table. It's simple enough:

```

my_main_t *mm = &my_main;
clib_bihash_8_8_t *h;
void callback_fn (clib_bihash_kv_8_8_t *, void *);

h = &mm->hash_table;

BV(clib_bihash_foreach_key_value_pair) (h, callback_fn, (void *) arg);

```

To nobody's great surprise: `clib_bihash_foreach_key_value_pair` iterates across the entire table, calling `callback_fn` with active entries.

Creating a new template instance

Creating a new template is easy. Use one of the existing templates as a model, and make the obvious changes. The `hash` and `key_compare` methods are performance-critical in multiple senses.

If the key compare method is slow, every lookup will be slow. If the hash function is slow, same story. If the hash function has poor statistical properties, space efficiency will suffer. In the limit, a bad enough hash function will cause large portions of the table to revert to linear search.

Use of the best available vector unit is well worth the trouble in the `hash` and `key_compare` functions.

This chapter contains a sample of the many ways FD.io VPP can be used. It is by no means an extensive list, but should give a sampling of the many features contained in FD.io VPP.

3.1 FD.io VPP with Virtual Machines

This chapter will describe how to use FD.io VPP with virtual machines. We describe how to create Vhost port with VPP and how to connect them to VPP. We will also discuss the limitations of Vhost.

3.1.1 Prerequisites

For this use case we will assume FD.io VPP is installed. We will also assume the user can create and start basic virtual machines. This use case will use the linux virsh commands. For more information on virsh refer to [virsh man page](#).

The image that we use is based on an Ubuntu cloud image downloaded from: [Ubuntu Cloud Images](#).

All FD.io VPP commands are being run from a su shell.

3.1.2 Topology

In this case we will use 2 systems. One system we will be running standard linux, the other will be running FD.io VPP.

3.1.3 Creating The Virtual Interface

We will start on the system running FD.io VPP and show that no Virtual interfaces have been created. We do this using the *Show Interface* command.

Notice we do not have any virtual interfaces. We do have an interface (TenGigabitEthernet86/0/0) that is up. This interface is connected to a system running, in our example standard linux. We will use this system to verify our connectivity to our VM with ping.

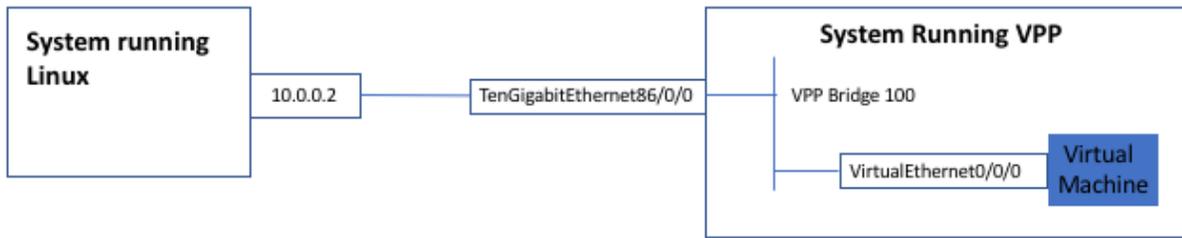


Fig. 1: Vhost Use Case Topology

```

$ sudo bash
# vppctl

  _/ _/ _ \  ( ) _  | | / / _ \ / _ \
  _/ _// // / / / _ \ | | / / _ _ / _ _ \
  /_ / _ _ ( ) _ \ _ \ | _ _ / / / _ /

vpp# clear interfaces
vpp# show int

```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up		
TenGigabitEthernet86/0/1	2	down		
local0	0	down		

```

vpp#

```

For more information on the interface commands refer to: [Interface Commands](#)

The next step will be to create the virtual port using the *Create Vhost-User* command. This command will create the virtual port in VPP and create a linux socket that the VM will use to connect to VPP.

The port can be created using VPP as the socket server or client.

Creating the VPP port:

```

vpp# create vhost socket /tmp/vm00.sock
VirtualEthernet0/0/0
vpp# show int

```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up		
TenGigabitEthernet86/0/1	2	down		
VirtualEthernet0/0/0	3	down		
local0	0	down		

```

vpp#

```

Notice the interface **VirtualEthernet0/0/0**. In this example we created the virtual interface as a client.

We can get more detail on the vhost connection with the *Show Vhost-User* command.

```

vpp# show vhost
Virtio vhost-user interfaces
Global:
  coalesce frames 32 time 1e-3

```

(continues on next page)

(continued from previous page)

```

    number of rx virtqueues in interrupt mode: 0
Interface: VirtualEthernet0/0/0 (ifindex 3)
virtio_net_hdr_sz 12
features mask (0xffffffffffffffff):
features (0x58208000):
  VIRTIO_NET_F_MRG_RXBUF (15)
  VIRTIO_NET_F_GUEST_ANNOUNCE (21)
  VIRTIO_F_ANY_LAYOUT (27)
  VIRTIO_F_INDIRECT_DESC (28)
  VHOST_USER_F_PROTOCOL_FEATURES (30)
protocol features (0x3)
  VHOST_USER_PROTOCOL_F_MQ (0)
  VHOST_USER_PROTOCOL_F_LOG_SHMFD (1)

socket filename /tmp/vm00.sock type client errno "No such file or directory"

rx placement:
tx placement: spin-lock
  thread 0 on vring 0
  thread 1 on vring 0

Memory regions (total 0)

```

Notice **No such file or directory** and **Memory regions (total 0)**. This is because the VM has not been created yet.

3.1.4 Creating the Virtual Machine

We will now create the virtual machine. We use the “virsh create command”. For the complete file we use refer to *An XML File*.

It is important to note that in the XML file we specify the socket path that is used to connect to FD.io VPP.

This is done with a section that looks like this

```

<interface type='vhostuser'>
  <mac address='52:54:00:4c:47:f2' />
  <source type='unix' path='/tmp//vm00.sock' mode='server' />
  <model type='virtio' />
  <alias name='net1' />
  <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0' />
</interface>

```

Notice the **interface type** and the **path** to the socket.

Now we create the VM. The virsh list command shows the VMs that have been created. We start with no VMs.

```

$ virsh list
Id      Name                               State
-----

```

Create the VM with the virsh create command specifying our xml file.

```

$ virsh create ./iperf3-vm.xml
Domain iperf-server3 created from ./iperf3-vm.xml

$ virsh list

```

(continues on next page)

(continued from previous page)

Id	Name	State
65	iperf-server3	running

The VM is now created.

Note: After a VM is created an xml file can be created with “virsh dumpxml”.

```
$ virsh dumpxml iperf-server3
<domain type='kvm' id='65'>
  <name>iperf-server3</name>
  <uuid>e23d37c1-10c3-4a6e-ae99-f315a4165641</uuid>
  <memory unit='KiB'>262144</memory>
  .....
```

Once the virtual machine is created notice the socket filename shows **Success** and there are **Memory Regions**. At this point the VM and FD.io VPP are connected. Also notice **qs2 256**. This system is running an older version of qemu. A queue size of 256 will affect vhost throughput. The qs2 should be 1024. On the web you should be able to find ways to install a newer version of qemu or change the queue size.

```
vpp# show vhost
Virtio vhost-user interfaces
Global:
  coalesce frames 32 time 1e-3
  number of rx virtqueues in interrupt mode: 0
Interface: VirtualEthernet0/0/0 (ifindex 3)
virtio_net_hdr_sz 12
features mask (0xfffffffffffffff):
features (0x58208000):
  VIRTIO_NET_F_MRG_RXBUF (15)
  VIRTIO_NET_F_GUEST_ANNOUNCE (21)
  VIRTIO_F_ANY_LAYOUT (27)
  VIRTIO_F_INDIRECT_DESC (28)
  VHOST_USER_F_PROTOCOL_FEATURES (30)
protocol features (0x3)
  VHOST_USER_PROTOCOL_F_MQ (0)
  VHOST_USER_PROTOCOL_F_LOG_SHMFD (1)

socket filename /tmp/vm00.sock type client errno "Success"

rx placement:
  thread 1 on vring 1, polling
tx placement: spin-lock
  thread 0 on vring 0
  thread 1 on vring 0

Memory regions (total 2)
region fd  guest_phys_addr  memory_size  userspace_addr  mmap_offset  _
↳ mmap_addr
=====
↳ =====
  0      31      0x0000000000000000  0x00000000000a0000  0x00007f1db9c00000_
↳ 0x0000000000000000  0x00007f7db0400  000
  1      32      0x00000000000c0000  0x00000000ff40000  0x00007f1db9cc0000_
↳ 0x00000000000c0000  0x00007f7d94ec0  000
```

(continues on next page)

(continued from previous page)

```

Virtqueue 0 (TX)
  qsz 256 last_avail_idx 0 last_used_idx 0
  avail.flags 0 avail.idx 256 used.flags 1 used.idx 0
  kickfd 33 callfd 34 errfd -1

Virtqueue 1 (RX)
  qsz 256 last_avail_idx 8 last_used_idx 8
  avail.flags 0 avail.idx 8 used.flags 1 used.idx 8
  kickfd 29 callfd 35 errfd -1

```

3.1.5 Bridge the Interfaces

To connect the 2 interfaces we put them on an L2 bridge.

Use the “set interface l2 bridge” command.

```

vpp# set interface l2 bridge VirtualEthernet0/0/0 100
vpp# set interface l2 bridge TenGigabitEthernet86/0/0 100
vpp# show bridge
  BD-ID  Index  BSN  Age(min)  Learning  U-Forwrd  UU-Flood  Flooding  ARP-Term
↪BVI-Intf
  100    1      0    off       on        on        on        on        off
↪N/A
vpp# show bridge 100 det
  BD-ID  Index  BSN  Age(min)  Learning  U-Forwrd  UU-Flood  Flooding  ARP-Term
↪BVI-Intf
  100    1      0    off       on        on        on        on        off
↪N/A

      Interface                If-idx  ISN  SHG  BVI  TxFlood      VLAN-Tag-Rewrite
VirtualEthernet0/0/0          3      1    0    -    *            none
TenGigabitEthernet86/0/0     1      1    0    -    *            none
vpp# show vhost

```

3.1.6 Bring the Interfaces Up

We can now bring all the pertinent interfaces up. We can then we will then be able to communicate with the VM from the remote system running Linux.

Bring the interfaces up with *Set Interface State* command.

```

vpp# set interface state VirtualEthernet0/0/0 up
vpp# set interface state TenGigabitEthernet86/0/0 up
vpp# sh int

```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up	rx packets	2
			rx bytes	180
TenGigabitEthernet86/0/1	2	down		
VirtualEthernet0/0/0	3	up	tx packets	2
			tx bytes	180
local0	0	down		

3.1.7 Ping from the VM

The remote Linux system has an ip address of “10.0.0.2” we can now reach it from the VM.

Use the “virsh console” command to attach to the VM. “ctrl-D” to exit.

```
$ virsh console iperf-server3
Connected to domain iperf-server3
Escape character is ^]

Ubuntu 16.04.3 LTS iperfvm ttyS0
.....

root@iperfvm:~# ping 10.0.0.2
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.285 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.154 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.159 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.208 ms
```

On VPP you can now see the packet counts increasing. The packets from the VM are seen as **rx packets** on **VirtualEthernet0/0/0**, they are then bridged to **TenGigabitEthernet86/0/0** and are seen leaving the system as **tx packets**. The reverse is true on the way in.

```
vpp# sh int
```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up	rx packets	16
			rx bytes	1476
			tx packets	14
			tx bytes	1260
TenGigabitEthernet86/0/1	2	down		
VirtualEthernet0/0/0	3	up	rx packets	14
			rx bytes	1260
			tx packets	16
			tx bytes	1476
local0	0	down		

```
vpp#
```

3.1.8 Cleanup

Destroy the VMs with “virsh destroy”

```
cto@tf-ucs-3:~$ virsh list
```

Id	Name	State
65	iperf-server3	running

```
cto@tf-ucs-3:~$ virsh destroy iperf-server3
Domain iperf-server3 destroyed
```

Delete the Virtual port in FD.io VPP

```
vpp# delete vhost-user VirtualEthernet0/0/0
vpp# show int
```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up	rx packets	21
			rx bytes	1928

(continues on next page)

(continued from previous page)

			tx packets	19
			tx bytes	1694
TenGigabitEthernet86/0/1	2	down		
local0	0	down		

Restart FD.io VPP

```
# service vpp restart
# vppctl show int
```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	down		
TenGigabitEthernet86/0/1	2	down		
local0	0	down		

3.1.9 The XML File

An example of a file that could be used with the virsh create command.

```
<domain type='kvm' id='54'>
  <name>iperf-server</name>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <memoryBacking>
    <hugepages>
      <page size='2048' unit='KiB' />
    </hugepages>
  </memoryBacking>
  <vcpu placement='static'>1</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='x86_64' machine='pc-i440fx-xenial'>hvm</type>
    <boot dev='hd' />
  </os>
  <features>
    <acpi />
    <apic />
  </features>
  <cpu mode='host-model'>
    <model fallback='allow'></model>
    <numa>
      <cell id='0' cpus='0' memory='262144' unit='KiB' memAccess='shared' />
    </numa>
  </cpu>
  <clock offset='utc'>
    <timer name='rtc' tickpolicy='catchup' />
    <timer name='pit' tickpolicy='delay' />
    <timer name='hpet' present='no' />
  </clock>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <pm>
    <suspend-to-mem enabled='no' />
  </pm>
</domain>
```

(continues on next page)

(continued from previous page)

```

    <suspend-to-disk enabled='no' />
</pm>
<devices>
  <emulator>/usr/bin/kvm</emulator>
  <disk type='file' device='disk'>
    <driver name='qemu' type='qcow2' />
    <source file='/tmp/xenial-mod.img' />
    <backingStore />
    <target dev='vda' bus='virtio' />
    <alias name='virtio-disk0' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x07' function='0x0' />
  </disk>
  <disk type='file' device='cdrom'>
    <driver name='qemu' type='raw' />
    <source file='/scratch/jdenisco/sae/configs/cloud-config.iso' />
    <backingStore />
    <target dev='hda' bus='ide' />
    <readonly />
    <alias name='ide0-0-0' />
    <address type='drive' controller='0' bus='0' target='0' unit='0' />
  </disk>
  <controller type='usb' index='0' model='ich9-ehci1'>
    <alias name='usb' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x7' />
  </controller>
  <controller type='pci' index='0' model='pci-root'>
    <alias name='pci.0' />
  </controller>
  <controller type='ide' index='0'>
    <alias name='ide' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x01' function='0x1' />
  </controller>
  <controller type='virtio-serial' index='0'>
    <alias name='virtio-serial0' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x05' function='0x0' />
  </controller>
  <interface type='vhostuser'>
    <mac address='52:54:00:4c:47:f2' />
    <source type='unix' path='/tmp//vm00.sock' mode='server' />
    <model type='virtio' />
    <alias name='net1' />
    <address type='pci' domain='0x0000' bus='0x00' slot='0x02' function='0x0' />
  </interface>
  <serial type='pty'>
    <source path='/dev/pts/2' />
    <target port='0' />
    <alias name='serial0' />
  </serial>
  <console type='pty' tty='/dev/pts/2'>
    <source path='/dev/pts/2' />
    <target type='serial' port='0' />
    <alias name='serial0' />
  </console>
  <input type='mouse' bus='ps2' />
  <input type='keyboard' bus='ps2' />
  <graphics type='vnc' port='5900' autoport='yes' listen='127.0.0.1'>
    <listen type='address' address='127.0.0.1' />

```

(continues on next page)

(continued from previous page)

```

</graphics>
<memballoon model='virtio'>
  <alias name='balloon0' />
  <address type='pci' domain='0x0000' bus='0x00' slot='0x08' function='0x0' />
</memballoon>
</devices>
<seclabel type='dynamic' model='apparmor' relabel='yes'>
  <label>libvirt-2c4c9317-c7a5-4b37-b789-386ccda7348a</label>
  <imagelabel>libvirt-2c4c9317-c7a5-4b37-b789-386ccda7348a</imagelabel>
</seclabel>
</domain>

```

3.2 Using VPP as a Home Gateway

Vpp running on a small system (with appropriate NICs) makes a fine home gateway. The resulting system performs far in excess of requirements: a TAG=vpp_debug image runs at a vector size of ~1.1 terminating a 90-mbit down / 10-mbit up cable modem connection.

At a minimum, install sshd and the isc-dhcp-server. If you prefer, you can use dnsmasq.

3.2.1 Configuration files

/etc/vpp/startup.conf:

```

unix {
  nodaemon
  log /var/log/vpp/vpp.log
  full-coredump
  cli-listen /run/vpp/cli.sock
  startup-config /setup.gate
  gid vpp
}
api-segment {
  gid vpp
}
dpdk {
  dev 0000:03:00.0
  dev 0000:14:00.0
  etc.
  poll-sleep 10
}

```

isc-dhcp-server configuration:

```

subnet 192.168.1.0 netmask 255.255.255.0 {
  range 192.168.1.10 192.168.1.99;
  option routers 192.168.1.1;
  option domain-name-servers 8.8.8.8;
}

```

If you decide to enable the vpp dns name resolver, substitute 192.168.1.2 for 8.8.8.8 in the dhcp server configuration.

/etc/ssh/sshd_config:

```
# What ports, IPs and protocols we listen for
Port <REDACTED-high-number-port>
# Change to no to disable tunnelled clear text passwords
PasswordAuthentication no
```

For your own comfort and safety, do NOT allow password authentication and do not answer ssh requests on port 22. Experience shows several hack attempts per hour on port 22, but none (ever) on random high-number ports.

vpp configuration:

```
comment { This is the WAN interface }
set int state GigabitEthernet3/0/0 up
comment { set int mac address GigabitEthernet3/0/0 mac-to-clone-if-needed }
set dhcp client intfc GigabitEthernet3/0/0 hostname vppgate

comment { Create a BVI loopback interface}
loop create
set int 12 bridge loop0 1 bvi
set int ip address loop0 192.168.1.1/24
set int state loop0 up

comment { Add more inside interfaces as needed ... }
set int 12 bridge GigabitEthernet0/14/0 1
set int state GigabitEthernet0/14/0 up

comment { dhcp server and host-stack access }
tap connect lstack address 192.168.1.2/24
set int 12 bridge tapcli-0 1
set int state tapcli-0 up

comment { Configure NAT}
nat44 add interface address GigabitEthernet3/0/0
set interface nat44 in loop0 out GigabitEthernet3/0/0

comment { allow inbound ssh to the <REDACTED-high-number-port>
nat44 add static mapping local 192.168.1.2 <REDACTED> external GigabitEthernet3/0/0
↔<REDACTED> tcp

comment { if you want to use the vpp DNS server, add the following }
comment { Remember to adjust the isc-dhcp-server configuration appropriately }
comment { nat44 add identity mapping external GigabitEthernet3/0/0 udp 53053 }
comment { bin dns_name_server_add_del 8.8.8.8 }
comment { bin dns_name_server_add_del 68.87.74.166 }
comment { bin dns_enable_disable }
comment { see patch below, which adds these commands }
service restart isc-dhcp-server
add default linux route via 192.168.1.1
```

3.2.2 Patches

You'll need this patch to add the “service restart” and “add default linux route” commands:

```
diff --git a/src/vpp/vnet/main.c b/src/vpp/vnet/main.c
index 6e136e19..69189c93 100644
--- a/src/vpp/vnet/main.c
+++ b/src/vpp/vnet/main.c
```

(continues on next page)

(continued from previous page)

```

@@ -18,6 +18,8 @@
#include <vlib/unix/unix.h>
#include <vnet/plugin/plugin.h>
#include <vnet/ethernet/ethernet.h>
+#include <vnet/ip/ip4_packet.h>
+#include <vnet/ip/format.h>
#include <vpp/app/version.h>
#include <vpp/api/vpe_msg_enum.h>
#include <limits.h>
@@ -400,6 +402,63 @@ VLIB_CLI_COMMAND (test_crash_command, static) = {

#ifdef

+static clib_error_t *
+restart_isc_dhcp_server_command_fn (vlib_main_t * vm,
+                                  unformat_input_t * input,
+                                  vlib_cli_command_t * cmd)
+{
+  int rv __attribute__((unused));
+  /* Wait three seconds... */
+  vlib_process_suspend (vm, 3.0);
+
+  rv = system ("/usr/sbin/service isc-dhcp-server restart");
+
+  vlib_cli_output (vm, "Restarted the isc-dhcp-server...");
+  return 0;
+}
+
+/* *INDENT-OFF* */
+VLIB_CLI_COMMAND (restart_isc_dhcp_server_command, static) = {
+  .path = "service restart isc-dhcp-server",
+  .short_help = "restarts the isc-dhcp-server",
+  .function = restart_isc_dhcp_server_command_fn,
+};
+/* *INDENT-ON* */
+
+static clib_error_t *
+add_default_linux_route_command_fn (vlib_main_t * vm,
+                                     unformat_input_t * input,
+                                     vlib_cli_command_t * c)
+{
+  int rv __attribute__((unused));
+  ip4_address_t ip4_addr;
+  u8 *cmd;
+
+  if (!unformat (input, "%U", unformat_ip4_address, &ip4_addr))
+    return clib_error_return (0, "default gateway address required...");
+
+  cmd = format (0, "/sbin/route add -net 0.0.0.0/0 gw %U",
+               format_ip4_address, &ip4_addr);
+  vec_add1 (cmd, 0);
+
+  rv = system (cmd);
+
+  vlib_cli_output (vm, "%s", cmd);
+
+  vec_free (cmd);

```

(continues on next page)

(continued from previous page)

```

+
+ return 0;
+}
+
+/* *INDENT-OFF* */
+VLIB_CLI_COMMAND (add_default_linux_route_command, static) = {
+ .path = "add default linux route via",
+ .short_help = "Adds default linux route: 0.0.0.0/0 via <addr>",
+ .function = add_default_linux_route_command_fn,
+};
+/* *INDENT-ON* */
+
+

```

3.2.3 Using the temporal mac filter plugin

If you need to restrict network access for certain devices to specific daily time ranges, configure the “mactime” plugin. Enable the feature on the NAT “inside” interfaces:

```

bin mactime_enable_disable GigabitEthernet0/14/0
bin mactime_enable_disable GigabitEthernet0/14/1
...

```

Create the required src-mac-address rule database. There are 4 rule entry types:

- allow-static - pass traffic from this mac address
- drop-static - drop traffic from this mac address
- allow-range - pass traffic from this mac address at specific times
- drop-range - drop traffic from this mac address at specific times

Here are some examples:

```

bin mactime_add_del_range name alarm-system mac 00:de:ad:be:ef:00 allow-static
bin mactime_add_del_range name unwelcome mac 00:de:ad:be:ef:01 drop-static
bin mactime_add_del_range name not-during-business-hours mac <mac> drop-range Mon -
↪Fri 7:59 - 18:01
bin mactime_add_del_range name monday-busines-hours mac <mac> allow-range Mon 7:59 -
↪18:01

```

3.3 vSwitch/vRouter

3.3.1 FD.io VPP as a vSwitch/vRouter

Note: We need to provide commands and and show how to use VPP as a vSwitch/vRouter

One of the use cases for the FD.io VPP platform is to implement it as a virtual switch or router. The following section describes examples of possible implementations that can be created with the FD.io VPP platform. For more in depth descriptions about other possible use cases, see the list of

You can use the FD.io VPP platform to create out-of-the-box virtual switches (vSwitch) and virtual routers (vRouter). The FD.io VPP platform allows you to manage certain functions and configurations of these application through a command-line interface (CLI).

Some of the functionality that a switching application can create includes:

- Bridge Domains
- Ports (including tunnel ports)
- Connect ports to bridge domains
- Program ARP termination

Some of the functionality that a routing application can create includes:

- Virtual Routing and Forwarding (VRF) tables (in the thousands)
- Routes (in the millions)

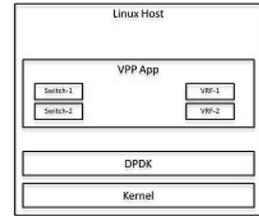


Fig. 2: Figure: Linux host as a vSwitch

This chapter describes some of the many techniques used to troubleshoot and diagnose problem with FD.io VPP implementations.

4.1 CPU Load/Usage

There are various commands and tools that can help users see FD.io VPP CPU and memory usage at runtime.

4.1.1 Linux top/htop

The Linux `top` and `htop` are decent tools to look at FD.io VPP cpu and memory usage, but they will only show preallocated memory and total CPU usage. These commands can be useful to show which cores VPP is running on.

This is an example of VPP instance that is running on cores 8 and 9. For this output type `top` and then type `1` when the tool starts.

```
$ top

top - 11:04:04 up 35 days, 3:16, 5 users, load average: 2.33, 2.23, 2.16
Tasks: 435 total, 2 running, 432 sleeping, 1 stopped, 0 zombie
%Cpu0  :  1.0 us,  0.7 sy,  0.0 ni, 98.0 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
%Cpu1  :  2.0 us,  0.3 sy,  0.0 ni, 97.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu2  :  0.7 us,  1.0 sy,  0.0 ni, 98.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu3  :  1.7 us,  0.7 sy,  0.0 ni, 97.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu4  :  2.0 us,  0.7 sy,  0.0 ni, 97.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu5  :  3.0 us,  0.3 sy,  0.0 ni, 96.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu6  :  2.3 us,  0.7 sy,  0.0 ni, 97.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu7  :  2.6 us,  0.3 sy,  0.0 ni, 97.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu8  : 96.0 us,  0.3 sy,  0.0 ni,  3.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu9  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
%Cpu10 :  1.0 us,  0.3 sy,  0.0 ni, 98.7 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
....
```

4.1.2 VPP Memory Usage

For details on VPP memory usage you can use the **show memory** command

This is the example VPP memory usage on 2 cores.

```
# vppctl show memory verbose
Thread 0 vpp_main
22043 objects, 17878k of 20826k used, 2426k free, 2396k reclaimed, 346k overhead,
↳1048572k capacity
  alloc. from small object cache: 22875 hits 39973 attempts (57.23%) replacements 5143
  alloc. from free-list: 44732 attempts, 26017 hits (58.16%), 528461 considered (per-
↳attempt 11.81)
  alloc. from vector-expand: 3430
  allocs: 52324 2027.84 clocks/call
  frees: 30280 594.38 clocks/call
Thread 1 vpp_wk_0
22043 objects, 17878k of 20826k used, 2427k free, 2396k reclaimed, 346k overhead,
↳1048572k capacity
  alloc. from small object cache: 22881 hits 39984 attempts (57.23%) replacements 5148
  alloc. from free-list: 44736 attempts, 26021 hits (58.17%), 528465 considered (per-
↳attempt 11.81)
  alloc. from vector-expand: 3430
  allocs: 52335 2027.54 clocks/call
  frees: 30291 594.36 clocks/call
```

4.1.3 VPP CPU Load

To find the VPP CPU load or how busy VPP is use the **show runtime** command.

With at least one interface in polling mode, the VPP CPU utilization is always 100%.

A good indicator of CPU load is “**average vectors/node**”. A bigger number means VPP is more busy but also more efficient. The Maximum value is 255 (unless you change `VLIB_FRAME_SIZE` in code). It basically means how many packets are processed in batch.

If VPP is not loaded it will likely poll so fast that it will just get one or few packets from the rx queue. This is the case shown below on Thread 1. As load goes up vpp will have more work to do, so it will poll less frequently, and that will result in more packets waiting in rx queue. More packets will result in more efficient execution of the code so number of clock cycles / packet will go down. When “average vectors/node” goes up close to 255, you will likely start observing rx queue tail drops.

```
# vppctl show run
Thread 0 vpp_main (lcore 8)
Time 6152.9, average vectors/node 0.00, last 128 main loops 0.00 per node 0.00
  vector rates in 0.0000e0, out 0.0000e0, drop 0.0000e0, punt 0.0000e0
  Name          State      Calls      Vectors
↳Suspends      Clocks    Vectors/Call
acl-plugin-fa-cleaner-process  event wait      0          0
↳ 1             3.66e4     0.00
admin-up-down-process          event wait      0          0
↳ 1             2.54e3     0.00
.....
-----
Thread 1 vpp_wk_0 (lcore 9)
Time 6152.9, average vectors/node 1.00, last 128 main loops 0.00 per node 0.00
  vector rates in 1.3073e2, out 1.3073e2, drop 6.5009e-4, punt 0.0000e0
```

(continues on next page)

(continued from previous page)

Name	Clocks	State	Calls	Vectors	
↳Suspends		Vectors/Call			
TenGigabitEthernet86/0/0-outpu		active	804395	804395	↳
↳ 0	6.17e2	1.00			
TenGigabitEthernet86/0/0-tx		active	804395	804395	↳
↳ 0	7.29e2	1.00			
arp-input		active	2	2	↳
↳ 0	3.82e4	1.00			
dpdk-input		polling	24239296364	804398	↳
↳ 0	1.59e7	0.00			
error-drop		active	4	4	↳
↳ 0	4.65e3	1.00			
ethernet-input		active	2	2	↳
↳ 0	1.08e4	1.00			
interface-output		active	1	1	↳
↳ 0	3.78e3	1.00			
ip4-glean		active	1	1	↳
↳ 0	6.98e4	1.00			
ip4-icmp-echo-request		active	804394	804394	↳
↳ 0	5.02e2	1.00			
ip4-icmp-input		active	804394	804394	↳
↳ 0	4.63e2	1.00			
ip4-input-no-checksum		active	804394	804394	↳
↳ 0	8.51e2	1.00			
ip4-load-balance		active	804394	804394	↳
↳ 0	5.46e2	1.00			
ip4-local		active	804394	804394	↳
↳ 0	5.79e2	1.00			
ip4-lookup		active	804394	804394	↳
↳ 0	5.71e2	1.00			
ip4-rewrite		active	804393	804393	↳
↳ 0	5.69e2	1.00			
ip6-input		active	2	2	↳
↳ 0	5.72e3	1.00			
ip6-not-enabled		active	2	2	↳
↳ 0	1.56e4	1.00			
unix-epoll-input		polling	835722	0	↳
↳ 0	3.03e-3	0.00			

5.1 Progressive VPP Tutorial

5.1.1 Introduction

This tutorial is designed for you to be able to run it on a single Ubuntu 16.04 VM on your laptop. It walks you through some very basic vpp scenarios, with a focus on learning vpp commands, doing common actions, and being able to discover common things about the state of a running vpp.

This is *not* intended to be a ‘how to run in a production environment’ set of instructions.

5.1.2 Exercise: Setting up your environment

All of these exercises are designed to be performed on an Ubuntu 16.04 (Xenial) box.

If you have an Ubuntu 16.04 box on which you have sudo, you can feel free to use that.

If you do not, a Vagrantfile is provided to setup a basic Ubuntu 16.04 box for you

5.1.3 Vagrant Set Up

Action: Install Virtualbox

If you do not already have virtualbox on your laptop (or if it is not up to date), please download and install it:

<https://www.virtualbox.org/wiki/Downloads>

Action: Install Vagrant

If you do not already have Vagrant on your laptop (or if it is not up to date), please download it:

<https://www.vagrantup.com/downloads.html>

Action: Create a Vagrant Directory

Create a directory on your laptop

```
mkdir fdio-tutorial
cd fdio-tutorial/
```

Create a Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure(2) do |config|

  config.vm.box = "puppetlabs/ubuntu-16.04-64-nocm"
  config.vm.box_check_update = false

  vmcpu=(ENV['VPP_VAGRANT_VMCPU'] || 2)
  vmram=(ENV['VPP_VAGRANT_VMRAM'] || 4096)

  config.ssh.forward_agent = true

  config.vm.provider "virtualbox" do |vb|
    vb.customize ["modifyvm", :id, "--ioapic", "on"]
    vb.memory = "#{vmram}"
    vb.cpus = "#{vmcpu}"
    #support for the SSE4.x instruction is required in some versions of VB.
    vb.customize ["setextradata", :id, "VBoxInternal/CPUM/SSE4.1", "1"]
    vb.customize ["setextradata", :id, "VBoxInternal/CPUM/SSE4.2", "1"]
  end
end
```

Action: Vagrant Up

Bring up your Vagrant VM:

```
vagrant up
```

Action: ssh to Vagrant VM

```
vagrant ssh
```

5.1.4 Exercise: Install VPP

Skills to be Learned

- Learn how to install vpp binary packages using apt-get.

Follow the instructions at [Installing VPP Binaries](#) for installing xenial vpp packages from the release repo. Please note, certain aspects of this tutorial require vpp 17.10 or later.

5.1.5 Exercise: VPP basics

Skills to be Learned

By the end of the exercise you should be able to:

- Run a vpp instance in a mode which allows multiple vpp processes to run
- Issue vpp commands from the unix shell
- Run a vpp shell and issue it commands

5.1.6 VPP command learned in this exercise

- `show ver`

5.1.7 Action: Remove dpdk plugin

In this tutorial, we will be running multiple vpp instances. DPDK does not work well with multiple instances, and so to run multiple instances we will need to disable the dpdk-plugin by removing it:

```
sudo rm -rf /usr/lib/vpp_plugins/dpdk_plugin.so
```

..how-to-run-vpp:

5.1.8 Action: Run VPP

VPP runs in userspace. In a production environment you will often run it with DPDK to connect to real NICs or vhost to connect to VMs. In those circumstances you usually run a single instance of vpp.

For purposes of this tutorial, it is going to be extremely useful to run multiple instances of vpp, and connect them to each other to form a topology. Fortunately, vpp supports this.

When running multiple vpp instances, each instance needs to have specified a ‘name’ or ‘prefix’. In the example below, the ‘name’ or ‘prefix’ is “vpp1”. Note that only one instance can use the dpdk plugin, since this plugin is trying to acquire a lock on a file.

```
sudo vpp unix {cli-listen /run/vpp/cli-vpp1.sock} api-segment { prefix vpp1 }
```

Example Output:

```
vlib_plugin_early_init:230: plugin path /usr/lib/vpp_plugins
```

Please note:

- “api-segment {prefix vpp1}” tells vpp how to name the files in /dev/shm/ for your vpp instance differently from the default.
- “unix {cli-listen /run/vpp/cli-vpp1.sock}” tells vpp to use a non-default socket file when being addressed by vppctl.

If you can’t see the vpp process running on the host, activate the nodaemon option to better understand what is happening

```
sudo vpp unix {nodaemon cli-listen /run/vpp/cli-vpp1.sock} api-segment { prefix vpp1 }
```

Example Output with errors from the dpdk plugin:

```
vlib_plugin_early_init:356: plugin path /usr/lib/vpp_plugins
load_one_plugin:184: Loaded plugin: acl_plugin.so (Access Control Lists)
load_one_plugin:184: Loaded plugin: dpdk_plugin.so (Data Plane Development Kit (DPDK))
load_one_plugin:184: Loaded plugin: flowprobe_plugin.so (Flow per Packet)
load_one_plugin:184: Loaded plugin: gtpu_plugin.so (GTPv1-U)
load_one_plugin:184: Loaded plugin: ila_plugin.so (Identifier-locator addressing for
↳IPv6)
load_one_plugin:184: Loaded plugin: ioam_plugin.so (Inbound OAM)
load_one_plugin:114: Plugin disabled (default): ixge_plugin.so
load_one_plugin:184: Loaded plugin: kubeproxy_plugin.so (kube-proxy data plane)
load_one_plugin:184: Loaded plugin: l2e_plugin.so (L2 Emulation)
load_one_plugin:184: Loaded plugin: lb_plugin.so (Load Balancer)
load_one_plugin:184: Loaded plugin: libsixrd_plugin.so (IPv6 Rapid Deployment on IPv4
↳Infrastructure (RFC5969))
load_one_plugin:184: Loaded plugin: memif_plugin.so (Packet Memory Interface
↳(experimental))
load_one_plugin:184: Loaded plugin: nat_plugin.so (Network Address Translation)
load_one_plugin:184: Loaded plugin: pppoe_plugin.so (PPPoE)
load_one_plugin:184: Loaded plugin: stn_plugin.so (VPP Steals the NIC for Container
↳integration)
vpp[10211]: vlib_pci_bind_to_uio: Skipping PCI device 0000:00:03.0 as host interface
↳eth0 is up
vpp[10211]: vlib_pci_bind_to_uio: Skipping PCI device 0000:00:04.0 as host interface
↳eth1 is up
vpp[10211]: dpdk_config:1240: EAL init args: -c 1 -n 4 --huge-dir /run/vpp/hugepages -
↳-file-prefix vpp -b 0000:00:03.0 -b 0000:00:04.0 --master-lcore 0 --socket-mem 64
EAL: No free hugepages reported in hugepages-1048576kB
EAL: Error - exiting with code: 1
Cause: Cannot create lock on '/var/run/.vpp_config'. Is another primary process
↳running?
```

5.1.9 Action: Send commands to VPP using vppctl

You can send vpp commands with a utility called *vppctl*.

When running vppctl against a named version of vpp, you will need to run:

```
sudo vppctl -s /run/vpp/cli- $\{name\}$ .sock  $\{cmd\}$ 
```

Note

```
/run/vpp/cli- $\{name\}$ .sock
```

is the particular naming convention used in this tutorial. By default you can set vpp to use what ever socket file name you would like at startup (the default config file uses `/run/vpp/cli.sock`) if two different vpps are being run (as in this tutorial) you must use distinct socket files for each one.

So to run ‘show ver’ against the vpp instance named vpp1 you would run:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show ver
```

Output:

```
vpp v17.04-rc0-177-g006eb47 built by ubuntu on fdio-ubuntu1604-sevt at Mon Jan 30
↳18:30:12 UTC 2017
```

5.1.10 Action: Start a VPP shell using vppctl

You can also use vppctl to launch a vpp shell with which you can run multiple vpp commands interactively by running:

```
sudo vppctl -s /run/vpp/cli-${name}.sock
```

which will give you a command prompt.

Try doing show ver that way:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock  
vpp# show ver
```

Output:

```
vpp v17.04-rc0~177-g006eb47 built by ubuntu on fdio-ubuntu1604-sevt at Mon Jan 30  
↪18:30:12 UTC 2017  
vpp#
```

To exit the vppctl shell:

```
vpp# quit
```

5.1.11 Exercise: Create an interface

Skills to be Learned

1. Create a veth interface in Linux host
2. Assign an IP address to one end of the veth interface in the Linux host
3. Create a vpp host-interface that connected to one end of a veth interface via AF_PACKET
4. Add an ip address to a vpp interface
5. Setup a 'trace'
6. View a 'trace'
7. Clear a 'trace'
8. Verify using ping from host
9. Ping from vpp
10. Examine Arp Table
11. Examine ip fib

VPP command learned in this exercise

1. create host-interface
2. set int state
3. set int ip address
4. show hardware
5. show int

6. show int addr
7. trace add
8. clear trace
9. ping
10. show ip arp
11. show ip fib

Topology

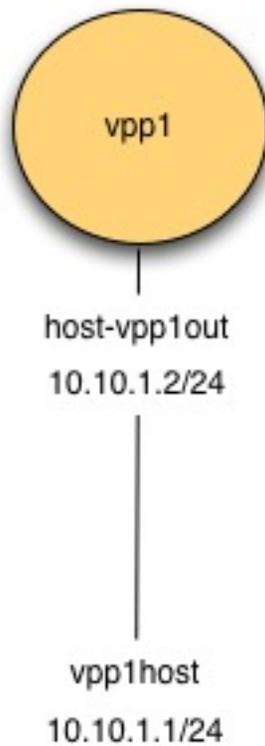


Fig. 1: Figure: Create Interface Topology

Initial State

The initial state here is presumed to be the final state from the exercise [VPP Basics](#)

Action: Create veth interfaces on host

In Linux, there is a type of interface call 'veth'. Think of a 'veth' interface as being an interface that has two ends to it (rather than one).

Create a veth interface with one end named **vpp1out** and the other named **vpp1host**

```
sudo ip link add name vpp1out type veth peer name vpp1host
```

Turn up both ends:

```
sudo ip link set dev vpp1out up
sudo ip link set dev vpp1host up
```

Assign an IP address

```
sudo ip addr add 10.10.1.1/24 dev vpp1host
```

Display the result:

```
sudo ip addr show vpp1host
```

Example Output:

```
10: vpp1host@vpp1out: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state_
↔UP group default qlen 1000
   link/ether 5e:97:e3:41:aa:b8 brd ff:ff:ff:ff:ff:ff
   inet 10.10.1.1/24 scope global vpp1host
       valid_lft forever preferred_lft forever
   inet6 fe80::5c97:e3ff:fe41:aab8/64 scope link
       valid_lft forever preferred_lft forever
```

Action: Create vpp host- interface

Create a host interface attached to **vpp1out**.

```
sudo vppctl -s /run/vpp/cli-vpp1.sock create host-interface name vpp1out
```

Output:

```
host-vpp1out
```

Confirm the interface:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show hardware
```

Example Output:

Name	Idx	Link	Hardware
host-vpp1out	1	up	host-vpp1out
Ethernet address 02:fe:48:ec:d5:a7			
Linux PACKET socket interface			
local0	0	down	local0
local			

Turn up the interface:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock set int state host-vpp1out up
```

Confirm the interface is up:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show int
```

Name	Idx	State	Counter	Count
host-vpplout	1	up		
local0	0	down		

Assign ip address 10.10.1.2/24

```
sudo vppctl -s /run/vpp/cli-vpp1.sock set int ip address host-vpplout 10.10.1.2/24
```

Confirm the ip address is assigned:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show int addr
```

```
host-vpplout (up):
 10.10.1.2/24
local0 (dn):
```

Action: Add trace

```
sudo vppctl -s /run/vpp/cli-vpp1.sock trace add af-packet-input 10
```

Action: Ping from host to vpp

```
ping -c 1 10.10.1.2
```

```
PING 10.10.1.2 (10.10.1.2) 56(84) bytes of data:
64 bytes from 10.10.1.2: icmp_seq=1 ttl=64 time=0.557 ms

--- 10.10.1.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.557/0.557/0.557/0.000 ms
```

Action: Examine Trace of ping from host to vpp

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show trace
```

```
----- Start of thread 0 vpp_main -----
Packet 1

00:09:30:397798: af-packet-input
  af_packet: hw_if_index 1 next-index 4
  tpacket2_hdr:
    status 0x20000001 len 42 snaplen 42 mac 66 net 80
    sec 0x588fd3ac nsec 0x375abde7 vlan 0 vlan_tpid 0
00:09:30:397906: ethernet-input
  ARP: fa:13:55:ac:d9:50 -> ff:ff:ff:ff:ff:ff
00:09:30:397912: arp-input
  request, type ethernet/IP4, address size 6/4
  fa:13:55:ac:d9:50/10.10.1.1 -> 00:00:00:00:00:00/10.10.1.2
00:09:30:398191: host-vpplout-output
```

(continues on next page)

(continued from previous page)

```

host-vpplout
ARP: 02:fe:48:ec:d5:a7 -> fa:13:55:ac:d9:50
reply, type ethernet/IP4, address size 6/4
02:fe:48:ec:d5:a7/10.10.1.2 -> fa:13:55:ac:d9:50/10.10.1.1

Packet 2

00:09:30:398227: af-packet-input
af_packet: hw_if_index 1 next-index 4
tpacket2_hdr:
  status 0x20000001 len 98 snaplen 98 mac 66 net 80
  sec 0x588fd3ac nsec 0x37615060 vlan 0 vlan_tpid 0
00:09:30:398295: ethernet-input
IP4: fa:13:55:ac:d9:50 -> 02:fe:48:ec:d5:a7
00:09:30:398298: ip4-input
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x9b46
  fragment id 0x894c, flags DONT_FRAGMENT
  ICMP echo_request checksum 0x83c
00:09:30:398300: ip4-lookup
fib 0 dpo-idx 5 flow hash: 0x00000000
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x9b46
  fragment id 0x894c, flags DONT_FRAGMENT
  ICMP echo_request checksum 0x83c
00:09:30:398303: ip4-local
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x9b46
  fragment id 0x894c, flags DONT_FRAGMENT
  ICMP echo_request checksum 0x83c
00:09:30:398305: ip4-icmp-input
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x9b46
  fragment id 0x894c, flags DONT_FRAGMENT
  ICMP echo_request checksum 0x83c
00:09:30:398307: ip4-icmp-echo-request
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x9b46
  fragment id 0x894c, flags DONT_FRAGMENT
  ICMP echo_request checksum 0x83c
00:09:30:398317: ip4-load-balance
fib 0 dpo-idx 10 flow hash: 0x0000000e
ICMP: 10.10.1.2 -> 10.10.1.1
  tos 0x00, ttl 64, length 84, checksum 0xbef3
  fragment id 0x659f, flags DONT_FRAGMENT
  ICMP echo_reply checksum 0x103c
00:09:30:398318: ip4-rewrite
tx_sw_if_index 1 dpo-idx 2 : ipv4 via 10.10.1.1 host-vpplout: IP4:
↔02:fe:48:ec:d5:a7 -> fa:13:55:ac:d9:50 flow hash: 0x00000000
IP4: 02:fe:48:ec:d5:a7 -> fa:13:55:ac:d9:50
ICMP: 10.10.1.2 -> 10.10.1.1
  tos 0x00, ttl 64, length 84, checksum 0xbef3
  fragment id 0x659f, flags DONT_FRAGMENT
  ICMP echo_reply checksum 0x103c
00:09:30:398320: host-vpplout-output
host-vpplout
IP4: 02:fe:48:ec:d5:a7 -> fa:13:55:ac:d9:50

```

(continues on next page)

(continued from previous page)

```
ICMP: 10.10.1.2 -> 10.10.1.1
  tos 0x00, ttl 64, length 84, checksum 0xbef3
  fragment id 0x659f, flags DONT_FRAGMENT
ICMP echo_reply checksum 0x103c
```

Action: Clear trace buffer

```
sudo vppctl -s /run/vpp/cli-vpp1.sock clear trace
```

Action: ping from vpp to host

```
sudo vppctl -s /run/vpp/cli-vpp1.sock ping 10.10.1.1
```

```
64 bytes from 10.10.1.1: icmp_seq=1 ttl=64 time=.0865 ms
64 bytes from 10.10.1.1: icmp_seq=2 ttl=64 time=.0914 ms
64 bytes from 10.10.1.1: icmp_seq=3 ttl=64 time=.0943 ms
64 bytes from 10.10.1.1: icmp_seq=4 ttl=64 time=.0959 ms
64 bytes from 10.10.1.1: icmp_seq=5 ttl=64 time=.0858 ms

Statistics: 5 sent, 5 received, 0% packet loss
```

Action: Examine Trace of ping from vpp to host

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show trace
```

```
----- Start of thread 0 vpp_main -----
Packet 1

00:12:47:155326: af-packet-input
  af_packet: hw_if_index 1 next-index 4
  tpacket2_hdr:
    status 0x20000001 len 98 snaplen 98 mac 66 net 80
    sec 0x588fd471 nsec 0x161c61ad vlan 0 vlan_tpid 0
00:12:47:155331: ethernet-input
  IP4: fa:13:55:ac:d9:50 -> 02:fe:48:ec:d5:a7
00:12:47:155334: ip4-input
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x2604
    fragment id 0x3e8f
  ICMP echo_reply checksum 0x1a83
00:12:47:155335: ip4-lookup
  fib 0 dpo-idx 5 flow hash: 0x00000000
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x2604
    fragment id 0x3e8f
  ICMP echo_reply checksum 0x1a83
00:12:47:155336: ip4-local
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x2604
```

(continues on next page)

(continued from previous page)

```

    fragment id 0x3e8f
    ICMP echo_reply checksum 0x1a83
00:12:47:155339: ip4-icmp-input
    ICMP: 10.10.1.1 -> 10.10.1.2
        tos 0x00, ttl 64, length 84, checksum 0x2604
    fragment id 0x3e8f
    ICMP echo_reply checksum 0x1a83
00:12:47:155342: ip4-icmp-echo-reply
    ICMP echo id 17572 seq 1
00:12:47:155349: error-drop
    ip4-icmp-input: unknown type

Packet 2

00:12:48:155330: af-packet-input
    af_packet: hw_if_index 1 next-index 4
    tpacket2_hdr:
        status 0x20000001 len 98 snaplen 98 mac 66 net 80
        sec 0x588fd472 nsec 0x1603e95b vlan 0 vlan_tpid 0
00:12:48:155337: ethernet-input
    IP4: fa:13:55:ac:d9:50 -> 02:fe:48:ec:d5:a7
00:12:48:155341: ip4-input
    ICMP: 10.10.1.1 -> 10.10.1.2
        tos 0x00, ttl 64, length 84, checksum 0x2565
    fragment id 0x3f2e
    ICMP echo_reply checksum 0x7405
00:12:48:155343: ip4-lookup
    fib 0 dpo-idx 5 flow hash: 0x00000000
    ICMP: 10.10.1.1 -> 10.10.1.2
        tos 0x00, ttl 64, length 84, checksum 0x2565
    fragment id 0x3f2e
    ICMP echo_reply checksum 0x7405
00:12:48:155344: ip4-local
    ICMP: 10.10.1.1 -> 10.10.1.2
        tos 0x00, ttl 64, length 84, checksum 0x2565
    fragment id 0x3f2e
    ICMP echo_reply checksum 0x7405
00:12:48:155346: ip4-icmp-input
    ICMP: 10.10.1.1 -> 10.10.1.2
        tos 0x00, ttl 64, length 84, checksum 0x2565
    fragment id 0x3f2e
    ICMP echo_reply checksum 0x7405
00:12:48:155348: ip4-icmp-echo-reply
    ICMP echo id 17572 seq 2
00:12:48:155351: error-drop
    ip4-icmp-input: unknown type

Packet 3

00:12:49:155331: af-packet-input
    af_packet: hw_if_index 1 next-index 4
    tpacket2_hdr:
        status 0x20000001 len 98 snaplen 98 mac 66 net 80
        sec 0x588fd473 nsec 0x15eb77ef vlan 0 vlan_tpid 0
00:12:49:155337: ethernet-input
    IP4: fa:13:55:ac:d9:50 -> 02:fe:48:ec:d5:a7
00:12:49:155341: ip4-input

```

(continues on next page)

(continued from previous page)

```
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x249e
  fragment id 0x3ff5
ICMP echo_reply checksum 0xf446
00:12:49:155343: ip4-lookup
  fib 0 dpo-idx 5 flow hash: 0x00000000
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x249e
  fragment id 0x3ff5
ICMP echo_reply checksum 0xf446
00:12:49:155345: ip4-local
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x249e
    fragment id 0x3ff5
  ICMP echo_reply checksum 0xf446
00:12:49:155349: ip4-icmp-input
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x249e
    fragment id 0x3ff5
  ICMP echo_reply checksum 0xf446
00:12:49:155350: ip4-icmp-echo-reply
  ICMP echo id 17572 seq 3
00:12:49:155354: error-drop
  ip4-icmp-input: unknown type

Packet 4

00:12:50:155335: af-packet-input
  af_packet: hw_if_index 1 next-index 4
  tpacket2_hdr:
    status 0x20000001 len 98 snaplen 98 mac 66 net 80
    sec 0x588fd474 nsec 0x15d2ffb6 vlan 0 vlan_tpid 0
00:12:50:155341: ethernet-input
  IP4: fa:13:55:ac:d9:50 -> 02:fe:48:ec:d5:a7
00:12:50:155346: ip4-input
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x2437
    fragment id 0x405c
  ICMP echo_reply checksum 0x5b6e
00:12:50:155347: ip4-lookup
  fib 0 dpo-idx 5 flow hash: 0x00000000
ICMP: 10.10.1.1 -> 10.10.1.2
  tos 0x00, ttl 64, length 84, checksum 0x2437
  fragment id 0x405c
ICMP echo_reply checksum 0x5b6e
00:12:50:155350: ip4-local
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x2437
    fragment id 0x405c
  ICMP echo_reply checksum 0x5b6e
00:12:50:155351: ip4-icmp-input
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x2437
    fragment id 0x405c
  ICMP echo_reply checksum 0x5b6e
00:12:50:155353: ip4-icmp-echo-reply
  ICMP echo id 17572 seq 4
```

(continues on next page)

(continued from previous page)

```

00:12:50:155356: error-drop
  ip4-icmp-input: unknown type

Packet 5

00:12:51:155324: af-packet-input
  af_packet: hw_if_index 1 next-index 4
  tpacket2_hdr:
    status 0x20000001 len 98 snaplen 98 mac 66 net 80
    sec 0x588fd475 nsec 0x15ba8726 vlan 0 vlan_tpid 0
00:12:51:155331: ethernet-input
  IP4: fa:13:55:ac:d9:50 -> 02:fe:48:ec:d5:a7
00:12:51:155335: ip4-input
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x23cc
    fragment id 0x40c7
  ICMP echo_reply checksum 0xedb3
00:12:51:155337: ip4-lookup
  fib 0 dpo-idx 5 flow hash: 0x00000000
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x23cc
    fragment id 0x40c7
  ICMP echo_reply checksum 0xedb3
00:12:51:155338: ip4-local
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x23cc
    fragment id 0x40c7
  ICMP echo_reply checksum 0xedb3
00:12:51:155341: ip4-icmp-input
  ICMP: 10.10.1.1 -> 10.10.1.2
    tos 0x00, ttl 64, length 84, checksum 0x23cc
    fragment id 0x40c7
  ICMP echo_reply checksum 0xedb3
00:12:51:155343: ip4-icmp-echo-reply
  ICMP echo id 17572 seq 5
00:12:51:155346: error-drop
  ip4-icmp-input: unknown type

Packet 6

00:12:52:175185: af-packet-input
  af_packet: hw_if_index 1 next-index 4
  tpacket2_hdr:
    status 0x20000001 len 42 snaplen 42 mac 66 net 80
    sec 0x588fd476 nsec 0x16d05dd0 vlan 0 vlan_tpid 0
00:12:52:175195: ethernet-input
  ARP: fa:13:55:ac:d9:50 -> 02:fe:48:ec:d5:a7
00:12:52:175200: arp-input
  request, type ethernet/IP4, address size 6/4
  fa:13:55:ac:d9:50/10.10.1.1 -> 00:00:00:00:00:00/10.10.1.2
00:12:52:175214: host-vpplout-output
  host-vpplout
  ARP: 02:fe:48:ec:d5:a7 -> fa:13:55:ac:d9:50
  reply, type ethernet/IP4, address size 6/4
  02:fe:48:ec:d5:a7/10.10.1.2 -> fa:13:55:ac:d9:50/10.10.1.1

```

After examining the trace, clear it again.

Action: Examine arp tables

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show ip arp
```

Time	IP4	Flags	Ethernet	Interface
570.4092	10.10.1.1	D	fa:13:55:ac:d9:50	host-vpp1out

Action: Examine routing table

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show ip fib
```

```
ipv4-VRF:0, fib_index 0, flow hash: src dst sport dport proto
0.0.0.0/0
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:0 buckets:1 uRPF:0 to:[0:0]]
  [0] [@0]: dpo-drop ip4
0.0.0.0/32
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:1 buckets:1 uRPF:1 to:[0:0]]
  [0] [@0]: dpo-drop ip4
10.10.1.1/32
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:10 buckets:1 uRPF:9 to:[5:420] via:[1:84]]
  [0] [@5]: ipv4 via 10.10.1.1 host-vpp1out: IP4: 02:fe:48:ec:d5:a7 ->
↔fa:13:55:ac:d9:50
10.10.1.0/24
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:8 buckets:1 uRPF:7 to:[0:0]]
  [0] [@4]: ipv4-glean: host-vpp1out
10.10.1.2/32
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:9 buckets:1 uRPF:8 to:[6:504]]
  [0] [@2]: dpo-receive: 10.10.1.2 on host-vpp1out
224.0.0.0/4
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:3 buckets:1 uRPF:3 to:[0:0]]
  [0] [@0]: dpo-drop ip4
240.0.0.0/4
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:2 buckets:1 uRPF:2 to:[0:0]]
  [0] [@0]: dpo-drop ip4
255.255.255.255/32
  unicast-ip4-chain
  [@0]: dpo-load-balance: [index:4 buckets:1 uRPF:4 to:[0:0]]
  [0] [@0]: dpo-drop ip4
```

5.1.12 Exercise: Connecting two vpp instances

Background

memif is a very high performance, direct memory interface type which can be used between vpp instances to form a topology. It uses a file socket for a control channel to set up that shared memory.

Skills to be Learned

You will learn the following new skill in this exercise:

1. Create a memif interface between two vpp instances

You should be able to perform this exercise with the following skills learned in previous exercises:

1. Run a second vpp instance
2. Add an ip address to a vpp interface
3. Ping from vpp

Topology

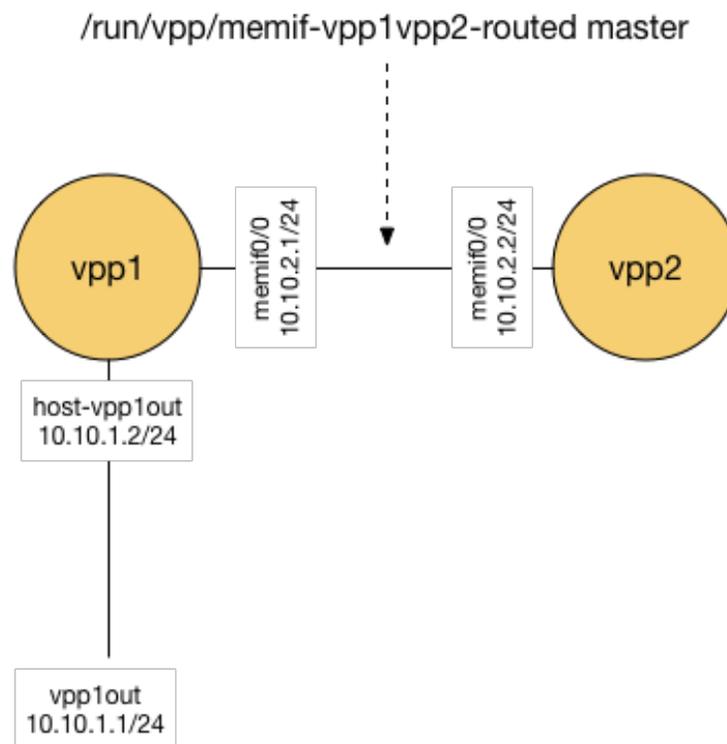


Fig. 2: Connect two vpp topology

Initial state

The initial state here is presumed to be the final state from the exercise [Create an Interface](#)

Action: Running a second vpp instances

You should already have a vpp instance running named: vpp1.

Run a second vpp instance named: vpp2.

Action: Create memif interface on vpp1

Create a memif interface on vpp1:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock create memif id 0 master
```

This will create an interface on vpp1 memif0/0 using /run/vpp/memif as its socket file. The role of vpp1 for this memif interface is 'master'.

Use your previously used skills to:

1. Set the memif0/0 state to up.
2. Assign IP address 10.10.2.1/24 to memif0/0
3. Examine memif0/0 via show commands

Action: Create memif interface on vpp2

We want vpp2 to pick up the 'slave' role using the same run/vpp/memif-vpp1vpp2 socket file

```
sudo vppctl -s /run/vpp/cli-vpp2.sock create memif id 0 slave
```

This will create an interface on vpp2 memif0/0 using /run/vpp/memif as its socket file. The role of vpp1 for this memif interface is 'slave'.

Use your previously used skills to:

1. Set the memif0/0 state to up.
2. Assign IP address 10.10.2.2/24 to memif0/0
3. Examine memif0/0 via show commands

Action: Ping from vpp1 to vpp2

Ping 10.10.2.2 from vpp1

Ping 10.10.2.1 from vpp2

5.1.13 Exercise: Routing

Skills to be Learned

In this exercise you will learn these new skills:

1. Add route to Linux Host routing table
2. Add route to vpp routing table

And revisit the old ones:

1. Examine vpp routing table
2. Enable trace on vpp1 and vpp2
3. ping from host to vpp
4. Examine and clear trace on vpp1 and vpp2
5. ping from vpp to host
6. Examine and clear trace on vpp1 and vpp2

vpp command learned in this exercise

1. `ip route add`

Topology

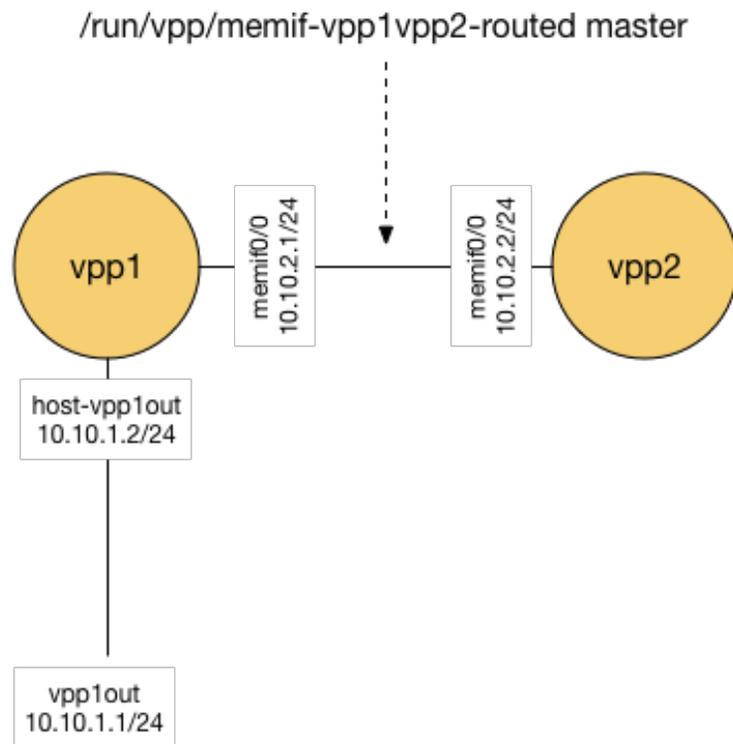


Fig. 3: Connect two vpp topology

Initial State

The initial state here is presumed to be the final state from the exercise [Connecting two vpp instances](#)

Action: Setup host route

```
sudo ip route add 10.10.2.0/24 via 10.10.1.2
ip route
```

```
default via 10.0.2.2 dev enp0s3
10.0.2.0/24 dev enp0s3 proto kernel scope link src 10.0.2.15
10.10.1.0/24 dev vpp1host proto kernel scope link src 10.10.1.1
10.10.2.0/24 via 10.10.1.2 dev vpp1host
```

Setup return route on vpp2

```
sudo vppctl -s /run/vpp/cli-vpp2.sock ip route add 10.10.1.0/24 via 10.10.2.1
```

Ping from host through vpp1 to vpp2

1. Setup a trace on vpp1 and vpp2
2. Ping 10.10.2.2 from the host
3. Examine the trace on vpp1 and vpp2
4. Clear the trace on vpp1 and vpp2

5.1.14 Exercise: Switching

Skills to be Learned

1. Associate an interface with a bridge domain
2. Create a loopback interface
3. Create a BVI (Bridge Virtual Interface) for a bridge domain
4. Examine a bridge domain

vpp command learned in this exercise

1. show bridge
2. show bridge detail
3. set int l2 bridge
4. show l2fib verbose

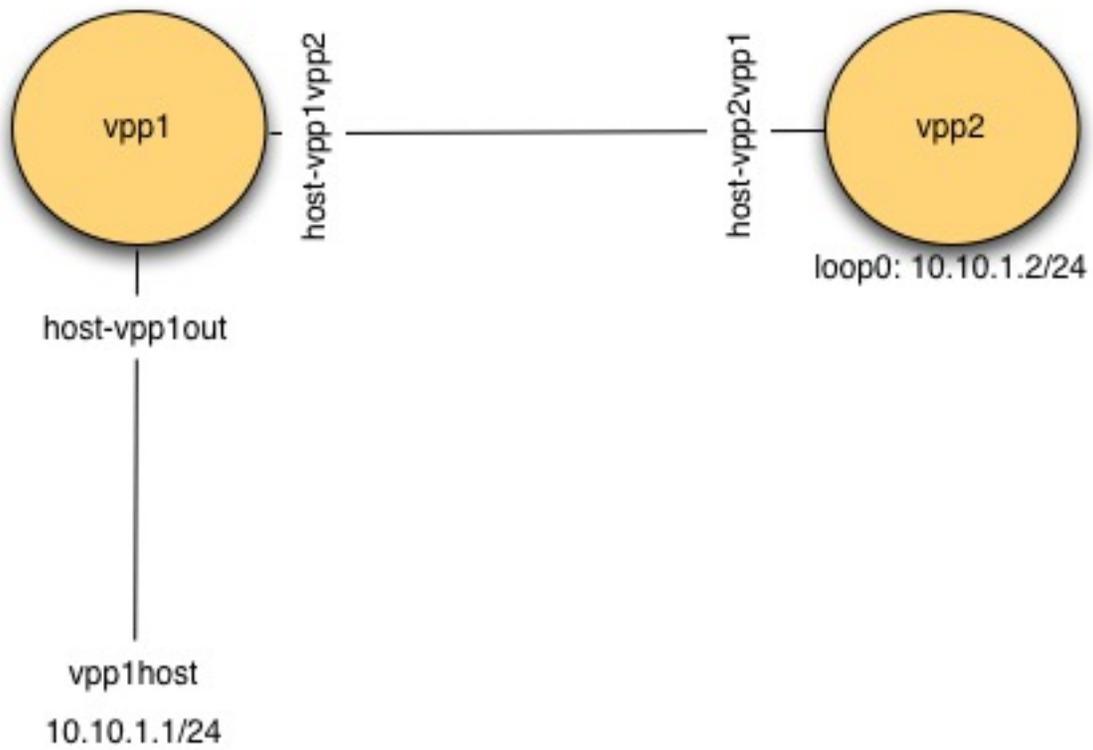


Fig. 4: Switching Topology

Topology

Initial state

Unlike previous exercises, for this one you want to start tabula rasa.

Note: You will lose all your existing config in your vpp instances!

To clear existing config from previous exercises run:

```
ps -ef | grep vpp | awk '{print $2}' | xargs sudo kill
sudo ip link del dev vpp1host
sudo ip link del dev vpp1vpp2
```

Action: Run vpp instances

1. Run a vpp instance named **vpp1**
2. Run a vpp instance named **vpp2**

Action: Connect vpp1 to host

1. Create a veth with one end named vpp1host and the other named vpp1out.
2. Connect vpp1out to vpp1
3. Add ip address 10.10.1.1/24 on vpp1host

Action: Connect vpp1 to vpp2

1. Create a veth with one end named vpp1vpp2 and the other named vpp2vpp1.
2. Connect vpp1vpp2 to vpp1.
3. Connect vpp2vpp1 to vpp2.

Action: Configure Bridge Domain on vpp1

Check to see what bridge domains already exist, and select the first bridge domain number not in use:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show bridge-domain
```

ID	Index	Learning	U-Forwrd	UU-Flood	Flooding	ARP-Term	BVI-Intf
0	0	off	off	off	off	off	local0

In the example above, there is bridge domain ID '0' already. Even though sometimes we might get feedback as below:

```
no bridge-domains in use
```

the bridge domain ID '0' still exists, where no operations are supported. For instance, if we try to add host-vpp1out and host-vpp1vpp2 to bridge domain ID 0, we will get nothing setup.

```
sudo vppctl -s /run/vpp/cli-vpp1.sock set int l2 bridge host-vpp1out 0
sudo vppctl -s /run/vpp/cli-vpp1.sock set int l2 bridge host-vpp1vpp2 0
sudo vppctl -s /run/vpp/cli-vpp1.sock show bridge-domain 0 detail
```

```
show bridge-domain: No operations on the default bridge domain are supported
```

So we will create bridge domain 1 instead of playing with the default bridge domain ID 0.

Add host-vpp1out to bridge domain ID 1

```
sudo vppctl -s /run/vpp/cli-vpp1.sock set int l2 bridge host-vpp1out 1
```

Add host-vpp1vpp2 to bridge domain ID1

```
sudo vppctl -s /run/vpp/cli-vpp1.sock set int l2 bridge host-vpp1vpp2 1
```

Examine bridge domain 1:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show bridge-domain 1 detail
```

BD-ID	Index	BSN	Age(min)	Learning	U-Forwrd	UU-Flood	Flooding	ARP-Term	BVI-
↔Intf									
1	1	0	off	on	on	on	on	off	N/A
	Interface			If-idx	ISN	SHG	BVI	TxFlood	VLAN-Tag-Rewrite
	host-vpp1out			1	1	0	-	*	none
	host-vpp1vpp2			2	1	0	-	*	none

Action: Configure loopback interface on vpp2

```
sudo vppctl -s /run/vpp/cli-vpp2.sock create loopback interface
```

```
loop0
```

Add the ip address 10.10.1.2/24 to vpp2 interface loop0. Set the state of interface loop0 on vpp2 to 'up'

Action: Configure bridge domain on vpp2

Check to see the first available bridge domain ID (it will be 1 in this case)

Add interface loop0 as a bridge virtual interface (bvi) to bridge domain 1

```
sudo vppctl -s /run/vpp/cli-vpp2.sock set int l2 bridge loop0 1 bvi
```

Add interface vpp2vpp1 to bridge domain 1

```
sudo vppctl -s /run/vpp/cli-vpp2.sock set int l2 bridge host-vpp2vpp1 1
```

Examine the bridge domain and interfaces.

Action: Ping from host to vpp and vpp to host

1. Add trace on vpp1 and vpp2
2. ping from host to 10.10.1.2
3. Examine and clear trace on vpp1 and vpp2
4. ping from vpp2 to 10.10.1.1

5. Examine and clear trace on vpp1 and vpp2

Action: Examine l2 fib

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show l2fib verbose
```

Mac Address	BD Idx	Interface	Index	static	filter	bvi	
↪Mac Age (min)							└
de:ad:00:00:00:00	1	host-vpp1vpp2	2	0	0	0	└
↪ disabled							
c2:f6:88:31:7b:8e	1	host-vpp1out	1	0	0	0	└
↪ disabled							
2 l2fib entries							

```
sudo vppctl -s /run/vpp/cli-vpp2.sock show l2fib verbose
```

Mac Address	BD Idx	Interface	Index	static	filter	bvi	
↪Mac Age (min)							└
de:ad:00:00:00:00	1	loop0	2	1	0	1	└
↪ disabled							
c2:f6:88:31:7b:8e	1	host-vpp2vpp1	1	0	0	0	└
↪ disabled							
2 l2fib entries							

5.1.15 Source NAT

Skills to be Learned

1. Abusing networks namespaces for fun and profit
2. Configuring snat address
3. Configuring snat inside and outside interfaces

vpp command learned in this exercise

1. snat add interface address
2. set interface snat

Topology

Initial state

Unlike previous exercises, for this one you want to start tabula rasa.

Note: You will lose all your existing config in your vpp instances!

To clear existing config from previous exercises run:

```
ps -ef | grep vpp | awk '{print $2}' | xargs sudo kill
sudo ip link del dev vpp1host
sudo ip link del dev vpp1vpp2
```

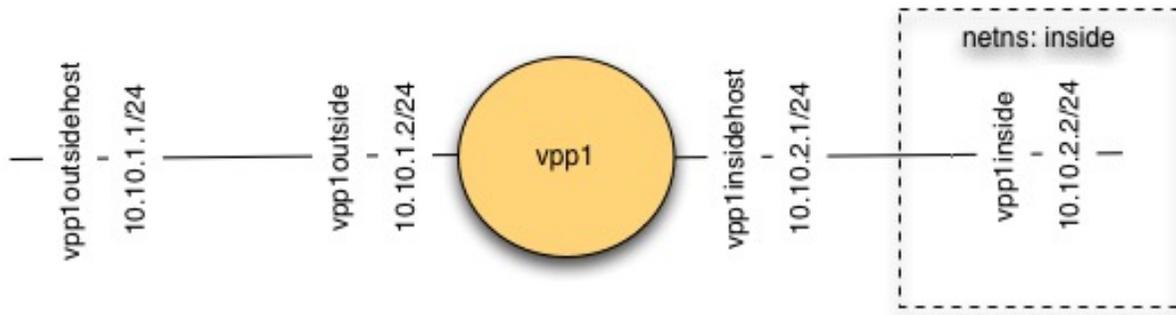


Fig. 5: SNAT Topology

Action: Install vpp-plugins

Snat is supported by a plugin, so vpp-plugins need to be installed

```
sudo apt-get install vpp-plugins
```

Action: Create vpp instance

Create one vpp instance named vpp1.

Confirm snat plugin is present:

```
sudo vppctl -s /run/vpp/cli-vpp1.sock show plugins
```

```
Plugin path is: /usr/lib/vpp_plugins
Plugins loaded:
1.ioam_plugin.so
2.ila_plugin.so
3.acl_plugin.so
4.flowperpkt_plugin.so
5.snat_plugin.so
6.libsixrd_plugin.so
7.lb_plugin.so
```

Action: Create veth interfaces

1. Create a veth interface with one end named vpp1outside and the other named vpp1outsidehost
2. Assign IP address 10.10.1.1/24 to vpp1outsidehost
3. Create a veth interface with one end named vpp1inside and the other named vpp1insidehost
4. Assign IP address 10.10.2.1/24 to vpp1insidehost

Because we'd like to be able to route *via* our vpp instance to an interface on the same host, we are going to put vpp1insidehost into a network namespace

Create a new network namespace 'inside'

```
sudo ip netns add inside
```

Move interface vpp1inside into the 'inside' namespace:

```
sudo ip link set dev vpp1insidehost up netns inside
```

Assign an ip address to vpp1insidehost

```
sudo ip netns exec inside ip addr add 10.10.2.1/24 dev vpp1insidehost
```

Create a route inside the netns:

```
sudo ip netns exec inside ip route add 10.10.1.0/24 via 10.10.2.2
```

Action: Configure vpp outside interface

1. Create a vpp host interface connected to vpp1outside
2. Assign ip address 10.10.1.2/24
3. Create a vpp host interface connected to vpp1inside
4. Assign ip address 10.10.2.2/24

Action: Configure snat

Configure snat to use the address of host-vpp1outside

```
sudo vppctl -s /run/vpp/cli-vpp1.sock snat add interface address host-vpp1outside
```

Configure snat inside and outside interfaces

```
sudo vppctl -s /run/vpp/cli-vpp1.sock set interface snat in host-vpp1inside out host-  
->vpp1outside
```

Action: Prepare to Observe Snat

Observing snat in this configuration is interesting. To do so, vagrant ssh a second time into your VM and run:

```
sudo tcpdump -s 0 -i vpp1outsidehost
```

Also enable tracing on vpp1

Action: Ping via snat

```
sudo ip netns exec inside ping -c 1 10.10.1.1
```

Action: Confirm snat

Examine the tcpdump output and vpp1 trace to confirm snat occurred.

5.2 API User Guides

This chapter describes how to use the C, Python and java APIs.

5.2.1 Downloading the jvpp jar

The following are instructions on how to download the jvpp jar

Getting jvpp jar

VPP provides java bindings which can be downloaded at:

- <https://nexus.fd.io/content/repositories/fd.io.release/io/fd/vpp/jvpp-core/18.01/jvpp-core-18.01.jar>

Getting jvpp via maven

1. Add the following to the repositories section in your `~/.m2/settings.xml` to pick up the fd.io maven repo:

```
<repository>
  <id>fd.io-release</id>
  <name>fd.io-release</name>
  <url>https://nexus.fd.io/content/repositories/fd.io.release/</url>
  <releases>
    <enabled>false</enabled>
  </releases>
  <snapshots>
    <enabled>true</enabled>
  </snapshots>
</repository>
```

For more information on setting up maven repositories in settings.xml, please look at:

- <https://maven.apache.org/guides/mini/guide-multiple-repositories.html>

2. Then you can get jvpp by putting in the dependencies section of your pom.xml file:

```
<dependency>
  <groupId>io.fd.vpp</groupId>
  <artifactId>jvpp-core</artifactId>
  <version>17.10</version>
</dependency>
```

For more information on maven dependency management, please look at:

- <https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>

6.1 Command Line Reference

This is a reference guide for the vpp debug commands that are referenced in the within these documents. This is **NOT** a complete list. For a complete list refer to the Debug CLI section of the [Source Code Documents](#).

The debug CLI can be executed from a su shell using the vppctl command.

```
# sudo bash
# vppctl show interface
```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up	rx packets	6569213
			rx bytes	9928352943
			tx packets	50384
			tx bytes	3329279
TenGigabitEthernet86/0/1	2	down		
VirtualEthernet0/0/0	3	up	rx packets	50384
			rx bytes	3329279
			tx packets	6569213
			tx bytes	9928352943
local0	0	down	drops	1498

Commands can also be executed from the vppct shell.

```
# vppctl
```



```
vpp# show interface
```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up	rx packets	6569213

(continues on next page)

(continued from previous page)

			rx bytes	9928352943
			tx packets	50384
			tx bytes	3329279
TenGigabitEthernet86/0/1	2	down		
VirtualEthernet0/0/0	3	up	rx packets	50384
			rx bytes	3329279
			tx packets	6569213
			tx bytes	9928352943
			drops	1498
local0	0	down		

6.1.1 Interface Commands

Show Hardware-Interfaces

Display more detailed information about all or a list of given interfaces. The verbosity of the output can be controlled by the following optional parameters:

- **brief:** Only show name, index and state (default for bonded interfaces).
- **verbose:** Also display additional attributes (default for all other interfaces).
- **detail:** Also display all remaining attributes and extended statistics.

To limit the output of the command to bonded interfaces and their slave interfaces, use the “*bond*” optional parameter.

Summary/Usage

```
show hardware-interfaces [brief|verbose|detail] [bond] [<interface> [<interface> [...]]
↳]]] [<sw_idx> [<sw_idx> [...]]].
```

Examples

Example of how to display default data for all interfaces:

```
vpp# show hardware-interfaces
      Name                Idx  Link  Hardware
GigabitEthernet7/0/0      1    up   GigabitEthernet7/0/0
  Ethernet address ec:f4:bb:c0:bc:fc
  Intel e1000
  carrier up full duplex speed 1000 mtu 9216
  rx queues 1, rx desc 1024, tx queues 3, tx desc 1024
  cpu socket 0
GigabitEthernet7/0/1      2    up   GigabitEthernet7/0/1
  Ethernet address ec:f4:bb:c0:bc:fd
  Intel e1000
  carrier up full duplex speed 1000 mtu 9216
  rx queues 1, rx desc 1024, tx queues 3, tx desc 1024
  cpu socket 0
VirtualEthernet0/0/0      3    up   VirtualEthernet0/0/0
  Ethernet address 02:fe:a5:a9:8b:8e
```

(continues on next page)

(continued from previous page)

```

VirtualEthernet0/0/1      4      up   VirtualEthernet0/0/1
  Ethernet address 02:fe:c0:4e:3b:b0
VirtualEthernet0/0/2      5      up   VirtualEthernet0/0/2
  Ethernet address 02:fe:1f:73:92:81
VirtualEthernet0/0/3      6      up   VirtualEthernet0/0/3
  Ethernet address 02:fe:f2:25:c4:68
local0                    0      down local0
  local

```

Example of how to display ‘*verbose*’ data for an interface by name and software index (where 2 is the software index):

```

vpp# show hardware-interfaces GigabitEthernet7/0/0 2 verbose
      Name          Idx  Link  Hardware
GigabitEthernet7/0/0      1   up   GigabitEthernet7/0/0
  Ethernet address ec:f4:bb:c0:bc:fc
  Intel e1000
    carrier up full duplex speed 1000 mtu 9216
    rx queues 1, rx desc 1024, tx queues 3, tx desc 1024
    cpu socket 0
GigabitEthernet7/0/1      2   down GigabitEthernet7/0/1
  Ethernet address ec:f4:bb:c0:bc:fd
  Intel e1000
    carrier up full duplex speed 1000 mtu 9216
    rx queues 1, rx desc 1024, tx queues 3, tx desc 1024
    cpu socket 0

```

Clear Hardware-Interfaces

Clear the extended statistics for all or a list of given interfaces (statistics associated with the ‘*show hardware-interfaces*’ command).

Summary/Usage

```
clear hardware-interfaces [<interface> [<interface> [..]] [<sw_idx> [<sw_idx> [..]]].
```

Examples

Example of how to clear the extended statistics for all interfaces:

```
vpp# clear hardware-interfaces
```

Example of how to clear the extended statistics for an interface by name and software index (where 2 is the software index):

```
vpp# clear hardware-interfaces GigabitEthernet7/0/0 2
```

Interface Commands

Show Interface

Shows software interface information including counters and features

Summary/Usage

```
show interface [address|addr|features|feat] [<interface> [<interface> [...]]]
```

Examples

Example of how to show the interface counters:

```
vpp# show int
```

Name	Idx	State	Counter	Count
TenGigabitEthernet86/0/0	1	up	rx packets	6569213
			rx bytes	9928352943
			tx packets	50384
			tx bytes	3329279
TenGigabitEthernet86/0/1	2	down		
VirtualEthernet0/0/0	3	up	rx packets	50384
			rx bytes	3329279
			tx packets	6569213
			tx bytes	9928352943
local0	0	down	drops	1498

Example of how to display the interface placement:

```
vpp# show interface rx-placement
Thread 1 (vpp_wk_0):
  node dpdk-input:
    GigabitEthernet7/0/0 queue 0 (polling)
  node vhost-user-input:
    VirtualEthernet0/0/12 queue 0 (polling)
    VirtualEthernet0/0/12 queue 2 (polling)
    VirtualEthernet0/0/13 queue 0 (polling)
    VirtualEthernet0/0/13 queue 2 (polling)
Thread 2 (vpp_wk_1):
  node dpdk-input:
    GigabitEthernet7/0/1 queue 0 (polling)
  node vhost-user-input:
    VirtualEthernet0/0/12 queue 1 (polling)
    VirtualEthernet0/0/12 queue 3 (polling)
    VirtualEthernet0/0/13 queue 1 (polling)
    VirtualEthernet0/0/13 queue 3 (polling)
```

Clear Interfaces

Clear the statistics for all interfaces (statistics associated with the *'show interface'* command).

Summary/Usage

```
clear interfaces
```

Example

Example of how to clear the statistics for all interfaces:

```
vpp# clear interfaces
```

Set Interface Mac Address

The `'set interface mac address'` command allows to set MAC address of given interface. In case of NIC interfaces the one has to support MAC address change. A side effect of MAC address change are changes of MAC addresses in FIB tables (ipv4 and ipv6).

Summary/Usage

```
set interface mac address <interface> <mac-address>.
```

Examples

Examples of how to change MAC Address of interface:

```
vpp# set interface mac address GigabitEthernet0/8/0 aa:bb:cc:dd:ee:01
vpp# set interface mac address host-vpp0 aa:bb:cc:dd:ee:02
vpp# set interface mac address tap-0 aa:bb:cc:dd:ee:03
vpp# set interface mac address pg0 aa:bb:cc:dd:ee:04
```

Set Interface Mtu

Summary/Usage

```
set interface mtu [packet|ip4|ip6|mpls] <value> <interface>.
```

Set Interface Promiscuous

Summary/Usage

```
set interface promiscuous [on|off] <interface>.
```

Set Interface State

This command is used to change the admin state (up/down) of an interface.

If an interface is down, the optional `'punt'` flag can also be set. The `'punt'` flag implies the interface is disabled for forwarding but punt all traffic to slow-path. Use the `'enable'` flag to clear `'punt'` flag (interface is still down).

Summary/Usage

```
set interface state <interface> [up|down|punt|enable].
```

Examples

Example of how to configure the admin state of an interface to **up**:

```
vpp# set interface state GigabitEthernet2/0/0 up
```

Example of how to configure the admin state of an interface to **down**:

```
vpp# set interface state GigabitEthernet2/0/0 down
```

Create Sub-Interfaces

This command is used to add VLAN IDs to interfaces, also known as subinterfaces. The primary input to this command is the *interface* and *subId* (subinterface Id) parameters. If no additional VLAN ID is provide, the VLAN ID is assumed to be the *subId*. The VLAN ID and *subId* can be different, but this is not recommended.

This command has several variations:

- **create sub-interfaces <interface> <subId>** - Create a subinterface to process packets with a given 802.1q VLAN ID (same value as the *subId*).
- **create sub-interfaces <interface> <subId> default** - Adding the *default* parameter indicates that packets with VLAN IDs that do not match any other subinterfaces should be sent to this subinterface.
- **create sub-interfaces <interface> <subId> untagged** - Adding the *untagged* parameter indicates that packets no VLAN IDs should be sent to this subinterface.
- **create sub-interfaces <interface> <subId>-<subId>** - Create a range of subinterfaces to handle a range of VLAN IDs.
- **create sub-interfaces <interface> <subId> dot1q|dot1ad <vlanId>|any [exact-match]** - Use this command to specify the outer VLAN ID, to either be explicited or to make the VLAN ID different from the *subId*.
- **create sub-interfaces <interface> <subId> dot1q|dot1ad <vlanId>|any inner-dot1q <vlanId>|any [exact-match]** - Use this command to specify the outer VLAN ID and the innner VLAN ID.

When *dot1q* or *dot1ad* is explicitly entered, subinterfaces can be configured as either exact-match or non-exact match. Non-exact match is the CLI default. If *exact-match* is specified, packets must have the same number of VLAN tags as the configuration. For non-exact-match, packets must at least that number of tags. L3 (routed) interfaces must be configured as exact-match. L2 interfaces are typically configured as non-exact-match. If *dot1q* or *dot1ad* is NOT entered, then the default behavior is exact-match.

Use the *show interface* command to display all subinterfaces.

Summary/Usage

```
create sub-interfaces <interface> {<subId> [default|untagged]} | {<subId>-<subId>} | {  
↪<subId> dot1q|dot1ad <vlanId>|any [inner-dot1q <vlanId>|any] [exact-match]}.
```

Examples

Example of how to create a VLAN subinterface 11 to process packets on 802.1q VLAN ID 11:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 11
```

The previous example is shorthand and is equivalent to:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 11 dot1q 11 exact-match
```

Example of how to create a subinterface number that is different from the VLAN ID:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 11 dot1q 100
```

Examples of how to create q-in-q and q-in-any subinterfaces:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 11 dot1q 100 inner-dot1q 200
vpp# create sub-interfaces GigabitEthernet2/0/0 12 dot1q 100 inner-dot1q any
```

Examples of how to create dot1ad interfaces:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 11 dot1ad 11
vpp# create sub-interfaces GigabitEthernet2/0/0 12 dot1ad 100 inner-dot1q 200
```

Examples of ‘*exact-match*’ versus non-exact match. A packet with outer VLAN 100 and inner VLAN 200 would match this interface, because the default is non-exact match:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 5 dot1q 100
```

However, the same packet would NOT match this interface because ‘*exact-match*’ is specified and only one VLAN is configured, but packet contains two VLANs:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 5 dot1q 100 exact-match
```

Example of how to create a subinterface to process untagged packets:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 5 untagged
```

Example of how to create a subinterface to process any packet with a VLAN ID that does not match any other subinterface:

```
vpp# create sub-interfaces GigabitEthernet2/0/0 7 default
```

When subinterfaces are created, they are in the down state. Example of how to enable a newly created subinterface:

```
vpp# set interface GigabitEthernet2/0/0.7 up
```

6.1.2 Vhost User Commands

Create Vhost-User

Create a vHost User interface. Once created, a new virtual interface will exist with the name ‘*VirtualEthernet0/0/x*’, where ‘*x*’ is the next free index.

There are several parameters associated with a vHost interface:

- **socket <socket-filename>** - Name of the linux socket used by hypervisor and VPP to manage the vHost interface. If in *'server'* mode, VPP will create the socket if it does not already exist. If in *'client'* mode, hypervisor will create the socket if it does not already exist. The VPP code is indifferent to the file location. However, if SELinux is enabled, then the socket needs to be created in *'/var/run/vpp/*.
- **server** - Optional flag to indicate that VPP should be the server for the linux socket. If not provided, VPP will be the client. In *'server'* mode, the VM can be reset without tearing down the vHost Interface. In *'client'* mode, VPP can be reset without bringing down the VM and tearing down the vHost Interface.
- **feature-mask <hex>** - Optional virtio/vhost feature set negotiated at startup. **This is intended for debugging only.** It is recommended that this parameter not be used except by experienced users. By default, all supported features will be advertised. Otherwise, provide the set of features desired.
 - 0x000008000 (15) - VIRTIO_NET_F_MRG_RXBUF
 - 0x000020000 (17) - VIRTIO_NET_F_CTRL_VQ
 - 0x000200000 (21) - VIRTIO_NET_F_GUEST_ANNOUNCE
 - 0x000400000 (22) - VIRTIO_NET_F_MQ
 - 0x004000000 (26) - VHOST_F_LOG_ALL
 - 0x008000000 (27) - VIRTIO_F_ANY_LAYOUT
 - 0x010000000 (28) - VIRTIO_F_INDIRECT_DESC
 - 0x040000000 (30) - VHOST_USER_F_PROTOCOL_FEATURES
 - 0x100000000 (32) - VIRTIO_F_VERSION_1
- **hwaddr <mac-addr>** - Optional ethernet address, can be in either X:X:X:X:X:X unix or X.X.X cisco format.
- **renumber <dev_instance>** - Optional parameter which allows the instance in the name to be specified. If instance already exists, name will be used anyway and multiple instances will have the same name. Use with caution.

Summary/Usage

```
create vhost-user socket <socket-filename> [server] [feature-mask <hex>] [hwaddr <mac-  
↪addr>] [renumber <dev_instance>]
```

Examples

Example of how to create a vhost interface with VPP as the client and all features enabled:

```
vpp# create vhost-user socket /var/run/vpp/vhost1.sock  
VirtualEthernet0/0/0
```

Example of how to create a vhost interface with VPP as the server and with just multiple queues enabled:

```
vpp# create vhost-user socket /var/run/vpp/vhost2.sock server feature-mask 0x40400000  
VirtualEthernet0/0/1
```

Once the vHost interface is created, enable the interface using:

```
vpp# set interface state VirtualEthernet0/0/0 up
```

Show Vhost-User

Display the attributes of a single vHost User interface (provide interface name), multiple vHost User interfaces (provide a list of interface names separated by spaces) or all Vhost User interfaces (omit an interface name to display all vHost interfaces).

Summary/Usage

```
show vhost-user [<interface> [<interface> [...]] [descriptors].
```

Examples

Example of how to display a vhost interface:

```
vpp# show vhost-user VirtualEthernet0/0/0
Virtio vhost-user interfaces
Global:
  coalesce frames 32 time 1e-3
Interface: VirtualEthernet0/0/0 (ifindex 1)
virtio_net_hdr_sz 12
features mask (0xffffffffffffffff):
features (0x50408000):
  VIRTIO_NET_F_MRG_RXBUF (15)
  VIRTIO_NET_F_MQ (22)
  VIRTIO_F_INDIRECT_DESC (28)
  VHOST_USER_F_PROTOCOL_FEATURES (30)
protocol features (0x3)
  VHOST_USER_PROTOCOL_F_MQ (0)
  VHOST_USER_PROTOCOL_F_LOG_SHMFD (1)

socket filename /var/run/vpp/vhost1.sock type client errno "Success"

rx placement:
  thread 1 on vring 1
  thread 1 on vring 5
  thread 2 on vring 3
  thread 2 on vring 7
tx placement: spin-lock
  thread 0 on vring 0
  thread 1 on vring 2
  thread 2 on vring 0

Memory regions (total 2)
region fd    guest_phys_addr  memory_size  userspace_addr  mmap_offset  _
↳ mmap_addr
=====
↳=====
  0    60    0x0000000000000000  0x000000000000a000  0x00002aaaaac00000_
↳0x0000000000000000  0x00002aab2b400000
  1    61    0x000000000000c000  0x000000003ff40000  0x00002aaaaacc0000_
↳0x000000000000c000  0x00002aababcc0000

Virtqueue 0 (TX)
  qsz 256 last_avail_idx 0 last_used_idx 0
```

(continues on next page)

(continued from previous page)

```

avail.flags 1 avail.idx 128 used.flags 1 used.idx 0
kickfd 62 callfd 64 errfd -1

Virtqueue 1 (RX)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 0 used.flags 1 used.idx 0
kickfd 65 callfd 66 errfd -1

Virtqueue 2 (TX)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 128 used.flags 1 used.idx 0
kickfd 63 callfd 70 errfd -1

Virtqueue 3 (RX)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 0 used.flags 1 used.idx 0
kickfd 72 callfd 74 errfd -1

Virtqueue 4 (TX disabled)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 0 used.flags 1 used.idx 0
kickfd 76 callfd 78 errfd -1

Virtqueue 5 (RX disabled)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 0 used.flags 1 used.idx 0
kickfd 80 callfd 82 errfd -1

Virtqueue 6 (TX disabled)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 0 used.flags 1 used.idx 0
kickfd 84 callfd 86 errfd -1

Virtqueue 7 (RX disabled)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 0 used.flags 1 used.idx 0
kickfd 88 callfd 90 errfd -1

```

The optional *'descriptors'* parameter will display the same output as the previous example but will include the descriptor table for each queue. The output is truncated below:

```

vpp# show vhost-user VirtualEthernet0/0/0 descriptors

Virtio vhost-user interfaces
Global:
  coalesce frames 32 time 1e-3
Interface: VirtualEthernet0/0/0 (ifindex 1)
virtio_net_hdr_sz 12
features mask (0xffffffffffffffff):
features (0x50408000):
  VIRTIO_NET_F_MRG_RXBUF (15)
  VIRTIO_NET_F_MQ (22)
:
Virtqueue 0 (TX)
qsz 256 last_avail_idx 0 last_used_idx 0
avail.flags 1 avail.idx 128 used.flags 1 used.idx 0

```

(continues on next page)

(continued from previous page)

```
kickfd 62 callfd 64 errfd -1

descriptor table:
  id      addr          len  flags  next  user_addr
=====
0      0x0000000010b6e974 2060 0x0002 1      0x00002aabbc76e974
1      0x0000000010b6e034 2060 0x0002 2      0x00002aabbc76e034
2      0x0000000010b6d6f4 2060 0x0002 3      0x00002aabbc76d6f4
3      0x0000000010b6cdb4 2060 0x0002 4      0x00002aabbc76cdb4
4      0x0000000010b6c474 2060 0x0002 5      0x00002aabbc76c474
5      0x0000000010b6bb34 2060 0x0002 6      0x00002aabbc76bb34
6      0x0000000010b6b1f4 2060 0x0002 7      0x00002aabbc76b1f4
7      0x0000000010b6a8b4 2060 0x0002 8      0x00002aabbc76a8b4
8      0x0000000010b69f74 2060 0x0002 9      0x00002aabbc769f74
9      0x0000000010b69634 2060 0x0002 10     0x00002aabbc769634
10     0x0000000010b68cf4 2060 0x0002 11     0x00002aabbc768cf4
:
249   0x0000000000000000 0      0x0000 250   0x00002aab2b400000
250   0x0000000000000000 0      0x0000 251   0x00002aab2b400000
251   0x0000000000000000 0      0x0000 252   0x00002aab2b400000
252   0x0000000000000000 0      0x0000 253   0x00002aab2b400000
253   0x0000000000000000 0      0x0000 254   0x00002aab2b400000
254   0x0000000000000000 0      0x0000 255   0x00002aab2b400000
255   0x0000000000000000 0      0x0000 32768 0x00002aab2b400000

Virtqueue 1 (RX)
  qsz 256 last_avail_idx 0 last_used_idx 0
```

Debug Vhost-User

Turn on/off debug for vhost

Summary/Usage

```
debug vhost-user <on | off>.
```

Delete Vhost-User

Delete a vHost User interface using the interface name or the software interface index. Use the *'show interface'* command to determine the software interface index. On deletion, the linux socket will not be deleted.

Summary/Usage

```
delete vhost-user {<interface> | sw_if_index <sw_idx>}.
```

Examples

Example of how to delete a vhost interface by name:

```
vpp# delete vhost-user VirtualEthernet0/0/1
```

Example of how to delete a vhost interface by software interface index:

```
vpp# delete vhost-user sw_if_index 1
```

6.2 VPP with Containers

This is the guide to using VPP with Vagrant (in a VM with two containers).

6.2.1 Overview

This section will describe how to install a Virtual Machine (VM) for Vagrant, and install containers inside that VM.

Containers are environments similar to VM's, but are known to be faster since they do not simulate separate kernels and hardware, as VM's do. You can read more about [Linux containers here](#).

In this section, we'll use Vagrant to run our VirtualBox VM. **Vagrant** automates the configuration of virtual environments by giving you the ability to create and destroy VM's quick and seamlessly. You have the git cloned repo of VPP locally on your machine.

Prerequisites

You have the git cloned repo of VPP locally on your machine.

Installing VirtualBox

First, download VirtualBox, which is virtualization software for creating VM's.

If you're on CentOS, follow the [steps here](#).

If you're on Ubuntu, perform:

```
$ sudo apt-get install virtualbox
```

Installing Vagrant

Now its time to install Vagrant.

Here we are on a 64-bit version of CentOS, downloading and installing Vagrant 2.1.1:

```
$ yum -y install https://releases.hashicorp.com/vagrant/2.1.1/vagrant_2.1.1_x86_64.rpm
```

Note: This is an installation of Vagrant 2.1.1 on a 64-bit CentOS machine.

If you don't have 64-bit CentOS or want to download a newer version of Vagrant, go to the [Vagrant download page](#), copy the download link for your specified version, and replace the [https://](#) link above and use the install command for the OS of your current system (*yum install* for CentOS or *apt-get install* for Ubuntu).

Vagrantfiles

The *Vagrantfile* contains the configuration settings for the machine and software requirements of your VM. Thus, any user with your *Vagrantfile* can instantiate a VM with those exact settings.

The syntax of Vagrantfiles is the programming language *Ruby*, but experience with Ruby is not completely necessary as most modifications to your Vagrantfile is changing variable values.

The Vagrantfile creates a **Vagrant Box**, which is a “development-ready box” that can be copied to other machines to recreate the same environment. The [Vagrant website for boxes](#) shows you all the available Vagrant Boxes containing different operating systems.

6.2.2 Creating your VM

As a prerequisite, you should already have the Git VPP directory on your machine.

Change directories to your *vpp/extras/vagrant* directory.

Looking at the **Vagrantfile**, we can see that the default OS is Ubuntu 16.04:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

Vagrant.configure(2) do |config|

  # Pick the right distro and bootstrap, default is ubuntu1604
  distro = ( ENV['VPP_VAGRANT_DISTRO'] || "ubuntu1604")
  if distro == 'centos7'
    config.vm.box = "centos/7"
    config.vm.box_version = "1708.01"
    config.ssh.insert_key = false
  elsif distro == 'opensuse'
    config.vm.box = "opensuse/openSUSE-42.3-x86_64"
    config.vm.box_version = "1.0.4.20170726"
  else
    config.vm.box = "puppetlabs/ubuntu-16.04-64-nocm"
```

As mentioned in the preface above, if you want a box configured to a different OS, you can specify which OS box you want on the [Vagrant boxes page](#).

Since there already exists a *Vagrantfile* from our repo, all you need to do is:

```
$ vagrant up
```

Note that doing this above command may take quite some time, since you are installing a VM. Take a break and get some scooby snacks.

To confirm it is up, we can do:

```
$ vagrant global-status
```

You will have only one machine running, but I have multiple as shown below:

```
[centos@dsk109 vpp-userdemo]$ vagrant global-status
id      name      provider  state  directory
-----
↪ ---
d90a17b default virtualbox poweroff /home/centos/andrew-vpp/vppsb/vpp-userdemo
```

(continues on next page)

(continued from previous page)

```
77b085e default virtualbox poweroff /home/centos/andrew-vpp/vppsb2/vpp-userdemo
c1c8952 default virtualbox poweroff /home/centos/andrew-vpp/testingVPPSB/extras/
↪vagrant
c199140 default virtualbox running /home/centos/andrew-vpp/vppsb3/vpp-userdemo
```

Note: To poweroff your VM, type **vagrant halt <id>**. If you want to try other commands on your box, visit the [Vagrant CLI Page](#).

6.2.3 Accessing your VM

Lets ssh into our newly created box:

```
$ vagrant ssh <id>
```

Now you're in your VM.

```
[[centos@dsk109 vpp-userdemo]$ vagrant ssh c1c
Welcome to Ubuntu 16.04 LTS (GNU/Linux 4.4.0-21-generic x86_64)

 * Documentation:  https://help.ubuntu.com/
Last login: Mon Jun 25 08:05:38 2018 from 10.0.2.2
vagrant@localhost:~$
```

Note: Type **exit** if you want to exit your VM, or container (which we'll get to soon.)

Let's set up the hugepages:

```
$ sysctl -w vm.nr_hugepages=1024
```

```
vagrant@localhost:~$ sysctl: permission denied on key 'vm.nr_hugepages'
```

Oh no! What happened? We're not root. Lets change to root.

```
$ sudo bash
```

Then we can perform the previous sysctl command with no issues.

To check if it was set correctly:

```
$ HUGE_PAGES=`sysctl -n vm.nr_hugepages`
$ echo $HUGE_PAGES
```

Which should output 1024.

Now we want to add the VPP repo as to our sources list in our VM. We append the FD.io binary repo to a file called 99fd.io.list, so *apt-get update* and *install* can use it:

```
ls /etc/apt # here is where you can see your sources.list.d directory after doing_
↪this command below

echo "deb [trusted=yes] https://nexus.fd.io/content/repositories/fd.io.ubuntu.xenial.
↪main/ ./" | sudo tee -a /etc/apt/sources.list.d/99fd.io.list
```

Do an *apt-get* to make sure the VM and its libraries are updated:

```
$ apt-get update
```

Now we want to install VPP and lxc (for our containers):

```
$ apt-get install vpp vpp-lib vpp-dpdk-dkms bridge-utils lxc
```

Now we can start running VPP on our host VM:

```
$ service vpp start
```

Check if we installed lxc:

```
$ lxc-checkconfig
```

6.2.4 Creating Containers

The system configuration is located at `/etc/lxc/lxc.conf` or `~/.config/lxc/lxc.conf` for unprivileged containers.

This configuration file is used to set values such as default lookup paths and storage backend settings for LXC. It can be found in each container's `/sys/class/net` directory.

The command below configures the LXC (Linux container) networks to create an interface for a Linux bridge and an unconsumed second interface to be used by each container.

For more information on linux containers with Ubuntu, visit the [lxc server guide](#).

```
echo -e "lxc.network.name = veth0\nlxc.network.type = veth\nlxc.network.name = veth_\n↪link1" | sudo tee -a /etc/lxc/default.conf
```

This next command will create an Ubuntu Xenial container named “cone”.

```
$ sudo lxc-create -t download -n cone -- --dist ubuntu --release xenial --arch amd64 -\n↪-keyserver hkp://p80.pool.sks-keyservers.net:80
```

If successful, you’ll get an output similar to this:

```
root@localhost:~# You just created an Ubuntu xenial amd64 (20180625_07:42) container.
```

```
To enable SSH, run: apt install openssh-server\nNo default root or user password are set by LXC.
```

You can make another container “ctwo”.

```
$ sudo lxc-create -t download -n ctwo -- --dist ubuntu --release xenial --arch amd64 -\n↪-keyserver hkp://p80.pool.sks-keyservers.net:80
```

Afterwards, you can list your containers:

```
$ sudo lxc-ls
```

```
root@localhost:~# cone ctwo
```

Here are some [lxc container commands](#) you may find useful:

```
sudo lxc-ls --fancy
sudo lxc-start --name u1 --daemon
sudo lxc-info --name u1
sudo lxc-stop --name u1
sudo lxc-destroy --name u1
```

Lets start the first container:

```
$ sudo lxc-start --name cone
```

Verify its running:

```
$ sudo lxc-ls --fancy
```

```
NAME STATE   AUTOSTART GROUPS IPV4 IPV6
cone RUNNING 0         -    -    -
ctwo STOPPED 0         -    -    -
```

6.2.5 Container prerequisites

Lets go into container *cone* and install prerequisites such as VPP, as well as some additional commands:

To enter our container via the shell, type:

```
$ sudo lxc-attach -n cone
```

Which should output:

```
root@cone:/#
```

Now run the linux DHCP setup and install VPP:

```
$ sudo bash
$ resolvconf -d eth0
$ dhclient
$ apt-get install -y wget
$ echo "deb [trusted=yes] https://nexus.fd.io/content/repositories/fd.io.ubuntu.
↪xenial.main/ ./" | sudo tee -a /etc/apt/sources.list.d/99fd.io.list
$ apt-get update
$ apt-get install -y --force-yes vpp
$ sh -c 'echo "\\ndpdk {\\n  no-pci\\n}" >> /etc/vpp/startup.conf'
```

And lets start VPP in this container as well:

```
$ service vpp start
```

Now repeat this process for the second container, *ctwo*, and also don't forget to "start" it with **sudo lxc-start --name *ctwo***.

6.2.6 Routing two Containers

Now lets go through the process of connecting these two linux containers to VPP and pinging between them.

In container *cone*, lets check our current network configuration:

```
$ ip -o a
```

We can see that we have three network interfaces, *lo*, *veth0*, and *veth_link1*.

```
root@cone:/# ip -o a
1: lo      inet 127.0.0.1/8 scope host lo\          valid_lft forever preferred_lft forever
1: lo      inet6 ::1/128 scope host \          valid_lft forever preferred_lft forever
30: veth0   inet 10.0.3.157/24 brd 10.0.3.255 scope global veth0\          valid_lft_
↳forever preferred_lft forever
30: veth0   inet6 fe80::216:3eff:fee2:d0ba/64 scope link \          valid_lft forever_
↳preferred_lft forever
32: veth_link1 inet6 fe80::2c9d:83ff:fe33:37e/64 scope link \          valid_lft_
↳forever preferred_lft forever
```

Notice that *veth_link1* has no assigned IP.

We can also check if our interfaces are down or up:

```
$ ip link
```

```
root@cone:/# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT_
↳group default qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
30: veth0@if31: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP_
↳mode DEFAULT group default qlen 1000
   link/ether 00:16:3e:e2:d0:ba brd ff:ff:ff:ff:ff:ff link-netnsid 0
32: veth_link1@if33: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state_
↳UP mode DEFAULT group default qlen 1000
   link/ether 2e:9d:83:33:03:7e brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

Note: Take note that **our** network index for *veth_link1* is 32, and that its parent index is 33, shown by *veth_link1@if33*. Yours will probably be different, but take note of these index's.

Lets make sure your loopback interface is up, and lets assign an IP and gateway to *veth_link1*.

```
$ ip link set dev lo up
$ ip addr add 172.16.1.2/24 dev veth_link1
$ ip link set dev veth_link1 up
$ ip route add default via 172.16.1.1 dev veth_link1
```

Here, the IP is 172.16.1.2/24 and the gateway is 172.16.1.1.

When I try to add the gateway, I get an error:

```
root@cone:/# ip route add default via 172.16.1.1 dev veth_link1
RTNETLINK answers: File exists
```

Fix this by renewing the DHCP leases, and then trying again:

```
root@cone:/# dhclient -r
Killed old client process
root@cone:/# ip route add default via 172.16.1.1 dev veth_link1
root@cone:/#
```

Now it works! :)

We can run some commands to verify our setup:

```

root@cone:/# ip -o a
1: lo      inet 127.0.0.1/8 scope host lo\          valid_lft forever preferred_lft forever
1: lo      inet6 ::1/128 scope host \          valid_lft forever preferred_lft forever
30: veth0   inet6 fe80::216:3eff:fee2:d0ba/64 scope link \          valid_lft forever
↳preferred_lft forever
32: veth_link1  inet 172.16.1.2/24 scope global veth_link1\          valid_lft forever
↳preferred_lft forever
32: veth_link1  inet6 fe80::2c9d:83ff:fe33:37e/64 scope link \          valid_lft
↳forever preferred_lft forever
root@cone:/# route
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
default          172.16.1.1     0.0.0.0         UG    0     0      0 veth_link1
172.16.1.0      *              255.255.255.0   U     0     0      0 veth_link1

```

We see that the IP has been assigned, as well as our default gateway.

Now exit this container and repeat this setup with **ctwo**, except with IP 172.16.2.2/24 and gateway 172.16.2.1.

After that's done, if you're still in a container, go back into your VM:

```
$ exit
```

Now, in the VM, if we run **ip link** we can see the host *veth* network interfaces, and their connection with the container *veth's*.

```

vagrant@localhost:~$ ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noqueue state DOWN mode DEFAULT group default qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
↳DEFAULT group default qlen 1000
   link/ether 08:00:27:33:82:8a brd ff:ff:ff:ff:ff:ff
3: enp0s8: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
↳DEFAULT group default qlen 1000
   link/ether 08:00:27:d9:9f:ac brd ff:ff:ff:ff:ff:ff
4: enp0s9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode
↳DEFAULT group default qlen 1000
   link/ether 08:00:27:78:84:9d brd ff:ff:ff:ff:ff:ff
5: lxcbr0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode
↳DEFAULT group default qlen 1000
   link/ether 00:16:3e:00:00:00 brd ff:ff:ff:ff:ff:ff
19: veth0C2FL7@if18: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
↳lxcbr0 state UP mode DEFAULT group default qlen 1000
   link/ether fe:0d:da:90:c1:65 brd ff:ff:ff:ff:ff:ff link-netnsid 1
21: veth8NA72P@if20: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
↳UP mode DEFAULT group default qlen 1000
   link/ether fe:1c:9e:01:9f:82 brd ff:ff:ff:ff:ff:ff link-netnsid 1
31: vethXQMY4C@if30: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
↳lxcbr0 state UP mode DEFAULT group default qlen 1000
   link/ether fe:9a:d9:29:40:bb brd ff:ff:ff:ff:ff:ff link-netnsid 0
33: vethQL7KOC@if32: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state
↳UP mode DEFAULT group default qlen 1000
   link/ether fe:ed:89:54:47:a2 brd ff:ff:ff:ff:ff:ff link-netnsid 0

```

Remember our network interface index 32 in *cone*? We can see at the bottom the name of the 33rd index **vethQL7KOC@if32**. Take note of this network interface name for the veth connected to cone, and the other network interface name for ctwo.

With VPP in our VM, we can show our current VPP interfaces:

```
$ sudo vppctl show inter
```

Which should only show local0.

Based on these names, which are specific to my systems, we can setup the VPP host-interfaces:

```
$ sudo vppctl create host-interface name vethQL7K0C
$ sudo vppctl create host-interface name veth8NA72P
```

Verify they have been setup:

```
$ sudo vppctl show inter
```

Which should output **three** interfaces, lo, and the other two network interfaces we just set up.

Change the links state to up:

```
$ sudo vppctl set interface state host-vethQL7K0C up
$ sudo vppctl set interface state host-veth8NA72P up
```

Add IP addresses for the other end of each veth link:

```
$ sudo vppctl set interface ip address host-vethQL7K0C 172.16.1.1/24
$ sudo vppctl set interface ip address host-veth8NA72P 172.16.2.1/24
```

Verify the interfaces are up with the previous show inter command, or you can also see the L3 table, or FIB by doing:

```
$ sudo vppctl show ip fib
```

At long last you probably want to see some pings:

```
$ sudo lxc-attach -n cone -- ping -c3 172.16.2.2
$ sudo lxc-attach -n ctwo -- ping -c3 172.16.1.2
```

Which should send/recv three packets for each command.