
openNetVM

GWU Cloud Lab

Aug 03, 2022

GETTING STARTED

1	Notes	3
2	Installing	5
3	Using OpenNetVM	7
4	Creating NFs	9
5	Dockerize NFs	11
6	TCP Stack	13
7	Citing OpenNetVM	15
7.1	Installation Guide	15
7.2	Development Environment	21
7.3	Debugging Guide	26
7.4	Cloudlab Tutorial	27
7.5	Three Node Topology Tutorial	31
7.6	Packet Generation Using Pktgen	38
7.7	Load Balancer Tutorial	41
7.8	Running OpenNetVM in Docker	47
7.9	MoonGen Installation (DPDK-2.0 Version)	50
7.10	Wireshark and Pdump for Packet Sniffing	54
7.11	OpenNetVM Examples	54
7.12	NF API	57
7.13	NF Development	60
7.14	Contribution Guidelines	64
7.15	Release Notes	65
7.16	Frequently Asked Questions	86
7.17	CI Development	88

OpenNetVM is a high performance NFV platform based on [DPDK](#) and [Docker](#) containers. OpenNetVM provides a flexible framework for deploying network functions and interconnecting them to build service chains.

OpenNetVM is an open source version of the NetVM platform described in our [NSDI 2014](#) and [HotMiddlebox 2016](#) papers, released under the [BSD](#) license.

The [develop](#) branch tracks experimental builds (active development) whereas the [master](#) branch tracks verified stable releases. Please read our [releases](#) document for more information about our releases and release cycle.

You can find information about research projects building on [OpenNetVM](#) at the [UCR/GW SDNFV project site](#). OpenNetVM is supported in part by NSF grants CNS-1422362 and CNS-1522546.

NOTES

We have updated our DPDK submodule to point to a new version, v18.11. If you have already cloned this repository, please update your DPDK submodule by running:

```
1 git submodule sync
2 git submodule update --init
```

And then rebuild DPDK using the install guide or running these commands:

```
1 cd dpdk
2 make config T=$RTE_TARGET
3 make T=$RTE_TARGET -j 8
4 make install T=$RTE_TARGET -j 8
```

See our release document for more information.

INSTALLING

To install OpenNetVM, please see the [OpenNetVM Installation](#) guide for a thorough walkthrough.

USING OPENNETVM

OpenNetVM comes with several sample network functions. To get started with some examples, please see the [Example Uses](#) guide

CREATING NFS

The [NF Development](#) guide will provide what you need to start creating your own NFs.

DOCKERIZE NFS

NFs can be run inside docker containers, with the NF being automatically or hand started. For more informations, see our [Docker guide](#).

TCP STACK

OpenNetVM can run mTCP applications as NFs. For more information, visit [mTCP](#).

CITING OPENNETVM

If you use OpenNetVM in your work, please cite our paper:

```
1 @inproceedings{zhang_opennetvm:_2016,  
2   title = {{OpenNetVM}: {A} {Platform} for {High} {Performance} {Network} {Service}  
   ↳ {Chains}},  
3   booktitle = {Proceedings of the 2016 {ACM} {SIGCOMM} {Workshop} on {Hot} {Topics} in  
   ↳ {Middleboxes} and {Network} {Function} {Virtualization}},  
4   publisher = {ACM},  
5   author = {Zhang, Wei and Liu, Guyue and Zhang, Wenhui and Shah, Neel and Lopreiato,   
   ↳ Phillip and Todeschi, Gregoire and Ramakrishnan, K.K. and Wood, Timothy},  
6   month = aug,  
7   year = {2016},  
8 }
```

Please let us know if you use OpenNetVM in your research by [emailing us](#) or completing this [short survey](#).

7.1 Installation Guide

This guide helps you build and install openNetVM.

7.1.1 Check System

1. Make sure your NIC is supported by Intel DPDK by comparing the following command's output against DPDK's [supported NIC list](#).

```
1 lspci | awk '/net/ {print $1}' | xargs -i% lspci -ks %
```

2. Check what operating system you have by typing:

```
1 uname -a
```

Your Kernel version should be higher than 2.6.33.

3. Install dependencies

Install the Linux Kernel headers package for your kernel version.

```
1 sudo apt-get install build-essential linux-headers-$(uname -r) git bc
```

If your distribution didn't come with Python or came with an earlier version, you will need to install Python 3 v3.4+.

See if Python is already installed with

```
1 python3 --version
```

If Python is not yet installed, use your distribution's package manager to install (Note: the command and package name may vary).

On Debian derivatives such as Ubuntu, use *apt*. Check the apt repository for the versions of Python available to you. Then, run the following command:

```
1 sudo apt-get install python3
```

Verify that Python installed correctly with

```
1 python3 --version
```

4. Ensure that your kernel supports uio

```
1 locate uio
```

5. Install libnuma

```
1 sudo apt-get install libnuma-dev
```

If installing libnuma-dev fails, your system may not be up to date. To fix this, run:

```
1 sudo apt-get update
```

7.1.2 Setup Repositories

1. Download source code

```
1 git clone https://github.com/sdnfv/openNetVM
2 cd openNetVM
3 git checkout master
```

This will ensure you are on the stable, `master` branch. If you want to use the most recent but potentially buggy features, you can use the default `develop` branch.

2. Initialize DPDK submodule

```
1 git submodule sync
2 git submodule update --init
```

From this point forward, this guide assumes that you are working out of the openNetVM source directory.

7.1.3 Set up Environment

1. Set environment variable ONVM_HOME to the path of the openNetVM source directory.

```
1 echo export ONVM_HOME=$(pwd) >> ~/.bashrc
```

2. List DPDK supported architectures:

```
1 ls dpdk/config/
```

3. Set environment variable RTE_SDK to the path of the DPDK library. Make sure that you are in the DPDK directory

```
1 cd dpdk
2 echo export RTE_SDK=$(pwd) >> ~/.bashrc
```

4. Set environment variable RTE_TARGET to the target architecture of your system. This is found in step 3.1

```
1 echo export RTE_TARGET=x86_64-native-linuxapp-gcc >> ~/.bashrc
```

5. Set environment variable ONVM_NUM_HUGEPAGES and ONVM_NIC_PCI.

ONVM_NUM_HUGEPAGES is a variable specifies how many hugepages are reserved by the user, default value of this is 1024, which could be set using:

```
1 echo export ONVM_NUM_HUGEPAGES=1024 >> ~/.bashrc
```

ONVM_NIC_PCI is a variable that specifies NIC ports to be bound to DPDK. If ONVM_NIC_PCI is not specified, the default action is to bind all non-active 10G NIC ports to DPDK. Note, NIC PCI device IDs may not be the same across all hosts. In that case, please retrieve this information for your host before setting the variable.

```
1 export ONVM_NIC_PCI=" 07:00.0 07:00.1 "
```

6. Source your shell rc file to set the environment variables:

```
1 source ~/.bashrc
```

7. Disable ASLR since it makes sharing memory with NFs harder:

```
1 sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
```

7.1.4 Configure and compile DPDK

1. Run the `install script` to compile DPDK and configure hugepages.

```
1 cd scripts
2 ./install.sh
```

The `install script` will automatically run the `environment setup script`, which configures your local environment. This should be run once for every reboot, as it loads the appropriate kernel modules and can bind your NIC ports to the DPDK driver.

7.1.5 Run DPDK HelloWorld Application

1. Enter DPDK HelloWorld directory and compile the application:

```
1 cd dpdk/examples/helloworld
2 make
```

2. Run the HelloWorld application

```
1 sudo ./build/helloworld -l 0,1 -n 1
```

If the last line of output is as follows, then DPDK works

```
1 hello from core 1
2 hello from core 0
```

7.1.6 Make and test openNetVM

1. Compile openNetVM manager and libraries

```
1 cd onvm
2 make
3 cd ..
```

Note: You may see the errors below upon compilation. Please ignore.

```
1 cat: ../openNetVM/onvm/lib/ABI_VERSION: No such file or directory found
2 cat: ../openNetVM/onvm/onvm_nflib/ABI_VERSION: No such file or directory found
```

2. Compile example NFs

```
1 cd examples
2 make
3 cd ..
```

3. Run openNetVM manager

Run openNetVM manager to use 3 cores (1 for displaying statistics, 1 for handling TX queues, 1 for handling manager RX queues; set to cores 0, 1 and 2, respectively, by default), to use 1 NIC port (hexadecimal portmask), 0xF8 for the NF coremask (cores 3, 4, 5, 6, 7), and to use stdout for the statistics console:

```
1 ./onvm/go.sh -k 1 -n 0xF8 -s stdout
```

You should see information regarding the NIC port that openNetVM is using, and openNetVM manager statistics will be displayed.

4. Run speed_tester NF

To test the system, we will run the speed_tester example NF. This NF generates a buffer of packets, and sends them to itself to measure the speed of a single NF TX thread.

In a new shell, run this command to start the speed_tester, assigning it a service ID of 1, setting its destination service ID to 1, and creating an initial batch of 16000 packets (increasing the packet count from the default 128 is especially important if you run a chain of multiple NFs):

```
1 cd examples/speed_tester
2 ./go.sh 1 -d 1 -c 16000
```

Once the NF's initialization is completed, you should see the NF display how many packets it is sending to itself. Go back to the manager to verify that *NF 1* is receiving data. If this is the case, the openNetVM is working correctly.

7.1.7 Configuring environment post reboot

After a reboot, you can configure your environment again (load kernel modules and bind the NIC) by running the [environment setup script](#).

Also, please double check if the environment variables from [Set up Environment](#) are initialized. If they are not, please go to [Set up Environment](#)

7.1.8 Troubleshooting

1. Setting up DPDK manually

Our install script helps configure DPDK by using its setup script. Sometimes, it's helpful to troubleshoot a problem by running DPDK's script directly to fix things like misconfigured `igb_uio` drivers or hugepage regions.

Here are the steps used to install the DPDK components needed by ONVM.

Run `dpdk/usertools/dpdk-setup.sh` then:

- Press [38] to compile x86_64-native-linuxapp-gcc version
- Press [45] to install `igb_uio` driver for Intel NICs
- Press [49] to setup 1024 2MB hugepages
- Press [51] to register the Ethernet ports
- Press [62] to quit the tool

After these steps, it should be possible to compile and run onvm.

2. Huge Page Configuration

You can get information about the hugepage configuration with:

```
1 grep -i huge /proc/meminfo
```

If there is not enough or no free memory, there are a few reasons why:

- **The manager crashed, but an NF(s) is still running.**
 - In this case, either kill them manually by hitting Ctrl+C or run `sudo pkill NF_NAME` for every NF that you have ran.
- **The manager and NFs are not running, but something crashed without freeing hugepages.**
 - To fix this, please run `sudo rm -rf /mnt/huge/*` to remove all files that contain hugepage data.
- **The above two cases are not met, something weird is happening:**
 - A reboot might fix this problem and free memory

3. Binding the NIC to the DPDK Driver

You can check the current status of NIC port bindings with

```
1 sudo ./usertools/dpdk-devbind.py --status
```

Output similar to below will show what driver each NIC port is bound to.

```
1 Network devices using DPDK-compatible driver
2 =====
3 <none>
4
5 Network devices using kernel driver
6 =====
7 0000:05:00.0 '82576 Gigabit Network Connection' if=eth0 drv=igb unused=igb_uio *Active*
8 0000:05:00.1 '82576 Gigabit Network Connection' if=eth1 drv=igb unused=igb_uio
9 0000:07:00.0 '82599EB 10-Gigabit SFI/SFP+ Network Connection' if=eth2 drv=ixgbe_
  ↳unused=igb_uio *Active*
10 0000:07:00.1 '82599EB 10-Gigabit SFI/SFP+ Network Connection' if=eth3 drv=ixgbe_
  ↳unused=igb_uio
```

In our example above, we see two 10G capable NIC ports that we could use with description '82599EB 10-Gigabit SFI/SFP+ Network Connection'.

One of the two NIC ports, 07:00.0, is active shown by the *Active* at the end of the line. Since the Linux Kernel is currently using that port, network interface eth2, we will not be able to use it with openNetVM. We must first disable the network interface in the Kernel, and then proceed to bind the NIC port to the DPDK Kernel module, igb_uio:

```
1 sudo ifconfig eth2 down
```

Rerun the status command, ./usertools/dpdk-devbind.py --status, to see that it is not active anymore. Once that is done, proceed to bind the NIC port to the DPDK Kernel module:

```
1 sudo ./usertools/dpdk-devbind.py -b igb_uio 07:00.0
```

Check the status again, ./usertools/dpdk-devbind.py --status, and assure the output is similar to our example below:

```
1 Network devices using DPDK-compatible driver
2 =====
3 0000:07:00.0 '82599EB 10-Gigabit SFI/SFP+ Network Connection' drv=igb_uio unused=ixgbe
4
5 Network devices using kernel driver
6 =====
7 0000:05:00.0 '82576 Gigabit Network Connection' if=eth0 drv=igb unused=igb_uio *Active*
8 0000:05:00.1 '82576 Gigabit Network Connection' if=eth1 drv=igb unused=igb_uio
9 0000:07:00.1 '82599EB 10-Gigabit SFI/SFP+ Network Connection' if=eth3 drv=ixgbe_
  ↳unused=igb_uio
```

4. Exporting \$ONVM_HOME

If the setup_environment.sh script fails because the environment variable ONVM_HOME is not set, please run this command: export ONVM_HOME=\$ONVM_HOME:CHANGEME_TO_THE_PATH_TO_ONVM_DIR

5. Poor Performance

If you are not getting the expected level of performance, try these:

- Ensure the manager and NFs are all given different core numbers. Use cores on the same sockets for best results.

- If running a long chain of NFs, ensure that there are sufficient packets to keep the chain busy. If using locally generated packets (i.e., the Speed Tester NFs) then use the `-c` flag to increase the number of packets created. For best results, run multiple Speed Tester NFs, or use an external generator like `pktgen`.

7.2 Development Environment

Visual Studio Code Remote Development allows contributors to use a consistent environment, access a development environment from multiple machines, and use runtimes not available on one's local OS. Their remote development platform can be configured with the Nimbus Cluster and CloudLab for development environments.

7.2.1 Getting Started

- Download [Visual Studio Code](#)
- Download the Remote Development Extension Pack which includes the 'Remote-SSH', 'Remote-WSL', and 'Remote-Containers' extensions:
 - From [Visual Studio Marketplace](#) or
 - Launch VSCode
 - * Click on the 'Extensions' icon on the left sidebar (`Shift + Command + X`)
 - * Search 'Remote Development' and install the 'Visual Studio Remote Development Extension Pack'
- Reload VS Code after installation is complete
- Setup and install OpenSSH on your local machine if one is not already present
- Setup your work environment with CloudLab and/or the Nimbus Cluster
- Setup Live Share so you can join and host real-time collaboration sessions

7.2.2 Setup OpenSSH

- Install [OpenSSH](#) if not already present on your machine
- Ensure that you have the `.ssh` directory in your user's home directory:

```
$ cd /Users/[Username]/.ssh
```

- If the directory does not exist on your machine, create it with the following command:

```
$ mkdir -p ~/.ssh && chmod 700 ~/.ssh
```

- If the config file does not yet exist in your `.ssh` directory, create it using command:

```
$ touch ~/.ssh/config
```

- Ensure your config file has the correct permissions using command:

```
$ chmod 600 ~/.ssh/config
```

7.2.3 CloudLab Work Environment

- Setup your [SSH public key](#) if you have not already done so
- Login to your CloudLab account and select ‘Manage SSH Keys’ under your account profile and add your public key
- When you instantiate a CloudLab experiment, a unique SSH command will be generated for you in the form: `ssh -p portNum user@hostname` listed under **Experiment -> List View -> SSH Command**
- Ensure that your generated ssh command works by running it in terminal
- Navigate to your system’s `.ssh` directory:

```
1 $ cd /Users/[Username]/.ssh
```

and modify the config file to include (**macOS** or **Linux**) :

```
1 Host hostname
2   HostName hostname
3   Port portNum
4   ForwardX11Trusted yes
5   User user_name
6   IdentityFile ~/.ssh/id_rsa
7   UseKeyChain yes
8   AddKeysToAgent yes
```

or (**Windows**) :

```
1 Host hostname
2   HostName hostname
3   Port portNum
4   User user_name
5   IdentityFile ~/.ssh/id_rsa
6   AddKeysToAgent yes
```

- Select ‘Remote-SSH: Connect to Host’ and enter `ssh -p portNum user@hostname` when prompted
- VS Code will automatically connect and set itself up
 - See [Troubleshooting tips](#) for connection issues and [Fixing SSH file permissions](#) for permissions errors
- After the connection is complete, you will be in an empty window and can then navigate to any folder or workspace using **File -> Open** or **File -> Workspace**
- To initialize and run openNetVM, select **File -> Open** and navigate to `/local/onvm/openNetVM/scripts`
 - Select **Terminal -> New Terminal** and run:

```
1 $ source setup_cloudlab.sh
2 $ sudo ifconfig ethXXX down
3 $ ./setup_environment.sh
```

where ethXXX is the NIC(s) you would like to bind to DPDK

- To disconnect from a remote host, select **File -> Close Remote Connection** or exit VS Code

7.2.4 Nimbus Cluster Work Environment

Nimbus VPN Method

- In order to connect directly to your node in the Nimbus Cluster through VS Code, you must be connected to the [Nimbus VPN](#)
- Navigate to your system's `.ssh` directory:

```
1 $ cd /Users/[Username]/.ssh
```

and modify the config file to include:

```
1 Host nimbnodeX
2   Hostname nimbnodeX
3   Username user_name
```

where 'X' is the node assigned by a Nimbus Cluster system administrator

- Launch VS Code and click on the green icon on the lower lefthand corner to open a remote window
- Select 'Remote-SSH: Connect to Host' and enter `user@nimbus.seas.gwu.edu` when prompted
- VSCode will automatically connect and set itself up - See [Troubleshooting tips](#) for connection issues and [Fixing SSH file permissions](#) for permissions errors
- After the connection is complete, you will be in an empty window and can then navigate to any folder or workspace using **File -> Open** or **File -> Workspace**
- To disconnect from a remote host, select **File -> Close Remote Connection** or exit VS Code

Alternative Method

You can also connect to the Nimbus Cluster through VS Code without using the Nimbus VPN. For instructions on how to configure this, see below.

If working with macOS or Linux:

- Navigate to your system's `.ssh` directory:

```
1 $ cd /Users/[Username]/.ssh
```

and modify the config file to include:

```
1 Host nimbnodeX
2   Username user_name
3   ProxyCommand ssh -q user_name@nimbus.seas.gwu.edu nc -q0 %h 22
```

where 'X' is the node assigned by a Nimbus Cluster system administrator

If working with Windows:

- Navigate to your system's .ssh directory:

```
1 $ cd /Users/[Username]/.ssh
```

and modify the config file to include:

```
1 Host nimbnodeX
2   Username user_name
3   ProxyCommand C:\Windows\System32\OpenSSH\ssh.exe -q user_name@nimbus.seas.gwu.edu nc -
  ↪ q0 %h 22
```

where 'X' is the node assigned by a Nimbus Cluster system administrator

Next: - Launch VS Code and click on the green icon on the lower lefthand corner to open a remote window - Select 'Remote-SSH: Connect to Host' and select the host you added, nimbnodeX, when prompted - VSCode will automatically connect and set itself up

- See [Troubleshooting tips](#) for connection issues and [Fixing SSH file permissions](#) for permissions errors
- After the connection is complete, you will be in an empty window and can then navigate to any folder or workspace using **File -> Open** or **File -> Workspace**
- To disconnect from a remote host, select **File -> Close Remote Connection** or exit VS Code

7.2.5 cpplint Setup

- [Linting](#) extensions run automatically when you save a file. Issues are shown as underlines in the code editor and in the *Problems* panel
- Install cpplint:
 - From [source](#) or
 - Mac & Linux:

```
1 $ sudo pip install cpplint
```

- Windows:

```
1 $ pip install cpplint
```

- Install the cpplint extension
 - From [Visual Studio Marketplace](#) or
 - Launch VSCode
 - * Click on the 'Extensions' icon on the left sidebar (Shift + Command + X)
 - * Search 'cpplint' and install

7.2.6 Live Share

Visual Studio [Live Share](#) allows developers to collaboratively edit in real-time through collaboration sessions.

- Install the Live Share extension:
 - From [Visual Studio Marketplace](#) or
 - Launch VSCode
 - * Click on the ‘Extensions’ icon on the left sidebar (**Shift + Command + X**)
 - * Search ‘Live Share Extension Pack’ and install
- **Note:** even if you already have the Live Share extension installed in your local VSCode application, you will have to reinstall it in your remote development environment in order to host collaboration sessions while you are connected to CloudLab or the Nimbus Cluster
- Reload VSCode after installation is complete
- **Note:** Linux users may need to follow extra [installation steps](#) to configure Live Share
- In order to join or host collaboration sessions, you must sign into Visual Studio Live Share with a Microsoft or GitHub account
 - To sign in, click on the blue ‘Live Share’ status bar item on the bottom of the window or press **Ctrl + Shift + P / Cmd + Shift + P** and select ‘Live Share: Sign in with Browser’ and proceed to sign in
- To learn about more features that Live Share provides, see the [User Guide](#)

7.2.7 Collaboration Sessions

To edit and share your code with other collaborators in real-time, you can start or join a collaboration session

- To start a session, launch VSCode and click the ‘Live Share’ status bar on the bottom of the window or press **Ctrl + Shift + P / Cmd + Shift + P** and select ‘Live Share: Start a collaboration session (Share)’
 - A unique invitation link will automatically be copied to your clipboard which can be shared with others who wish to join your session
 - To access the invitation link again, click on the session state status bar icon and select ‘Invite Others (Copy Link)’
- Once you start your session, a pop-up message will notify you that your link has been copied to your clipboard and will allow you to select ‘Make read-only’ if you wish to prevent guests from editing your files
- If ‘read-only’ mode is not enabled, hosts and guests both have access to co-edit all files within the development environment as well as view each others edits in real-time
 - Co-editing abilities may be limited, depending on [language and platform support](#)
- You will be notified as guests join your session via your invitation link which will also grant you the option to remove them from the session
- To terminate your session, open the ‘Live Share’ custom tab and select ‘Stop collaboration session’
 - After the session has ended, guests will no longer have access to any content and all temp files will be cleaned up

7.2.8 Troubleshooting

On **Windows**, connecting GitHub to Visual Studio Code often has issues with GitHub Actions Permissions. VS Code often fails to request the “Workflow” permission, which is necessary for running the GitHub Actions we have on our repository. If you run into an error when pushing to a forked branch: *[remote rejected] <branch name> -> <branch name> (refusing to allow an OAuth App to create or update workflow)*, it is likely because you don’t have the “Workflow” scope on the OAuth link you accepted to connect VS Code and GitHub.

To fix this, simply change the long OAuth link VS Code sends you to: `scope=repo` should be `scope=repo,workflow`. Once you update the link and load the page, you should be able to accept the updated permissions and push to GitHub.

7.3 Debugging Guide

7.3.1 Compile with Debugging Symbols

When programming NFs or the manager, it is often necessary to debug. By default openNetVM does not compile with debugging symbols but is able to when compiler flags are set.

For example, you may want to debug `speed_tester` with `gdb`.

- The first step is to set the correct environment variables which can be done using the following command:

```
1 export USER_FLAGS="-g -O0"
```

- `-g` produces debugging symbols and `-O0` reduces compiler optimizations to the lowest level
- Now navigate to your debug target and run `make`

```
1 cd examples/speed_tester
2 make
```

- The executable now has debugging symbols. Run it in `gdb` with the necessary args. For example:

```
1 sudo gdb --args ./build/speed_tester -l 5 -n 3 --proc-type=secondary -- -r 1 -- -d 1
```

- It is now possible to set breakpoints and perform other `gdb` operations!

Troubleshooting: If debugging symbols are not found verify that debugging flags are set with `echo $USER_FLAGS` and also try executing the following command in `gdb`:

- For `onvm_mgr`:

```
1 file onvm_mgr/x86_64-native-linuxapp-gcc/app/onvm_mgr
```

- For NFs:

```
1 file build/app/NF_NAME
```

If for some reason `USER_FLAGS` does not set correctly, it’s possible to edit the `Makefile` of your debug target and set the flags manually. It can be done by adding a line similar to this:

```
1 CFLAGS += -g
```

7.3.2 Packet capturing

When working with different packet protocols and tcp related applications it is often needed to look at the packets received/sent by the manager. DPDK provides a dpdk-pdump application that can capture packets to a pcap file.

To use dpdk-pdump set CONFIG_RTE_LIBRTE_PMD_PCAP=y in dpdk/config/common_base and then recompile dpdk.

Then execute dpdk-pdump as a secondary application when the manager is running

```
1 cd dpdk/x86_64-native-linuxapp-gcc
2 sudo ./build/app/pdump/dpdk-pdump -- --pdump 'port=0,queue=*,rx-dev=/tmp/rx.pcap'
```

Full set of options and configurations for dpdk-pdump can be found [here](#).

7.3.3 Possible crash reasons

Both primary and secondary dpdk processes must have the exact same hugepage memory mappings to function correctly. This can be an issue when using complex NFs that have a large memory footprint. When using such NFs a memory discrepancy occurs between a NF and onvm_mgr, which leads to onvm_mgr crashes.

The NF/mgr hugepage memory layout discrepancy is resolved by using the base virtual address value for onvm_mgr. Examples of complex NFs: ndpi_stats, onvm_mt看cp epserver

Example onvm_mgr setup:

```
1 ./go.sh -k 3 -n 0xF3 -s stdout
```

7.4 Cloudlab Tutorial

7.4.1 Getting Started

- **This tutorial assumes that you have access to the NSF sponsored [CloudLab](#) testbed**
 - To gain access, you must first create an account, and join a project from your university. See the [CloudLab Docs](#) for more information

7.4.2 Setting up SSH Keys for CloudLab

If you do not already have an SSH key associated with your account you will need to create one. Here are the steps to create an SSH key and upload it to CloudLab.

- Open a terminal and cd to the directory you want to store the keys in

Generally, on a Windows machine, all SSH-related files are found in following location:
/c/Users/PC_USER_NAME/.ssh/

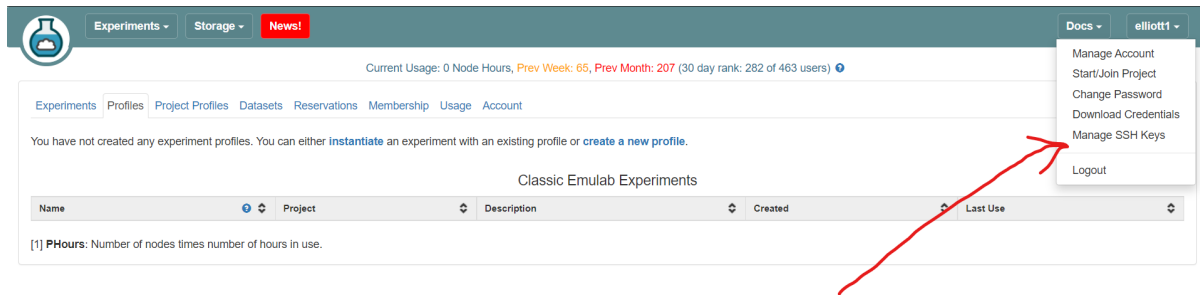
On Mac/Linux, it's typically found in the ~/.ssh directory

To generate an SSH Key:

- Type ssh-keygen to make an ssh-key
- Give it a name and an optional password
- The public key has the extension .pub

To upload the key:

- Go to cloudlab, and click on your username in the top-right corner
- Go to “manage SSH keys”
- Press “Load from file” and select the public key file
- Once it’s loaded in, select “add key”



7.4.3 Starting a ONVM Profile With 1 Node on CloudLab

- Click on experiments in the upper-left corner, then start an experiment
- For “select a profile” you can choose the desired profile (onvm is the most updated/recommended)
If you don’t have access to the GW profiles, you can find it [here](#)
- For “parameterize” use 1 host, and for node type, you can keep it as is or select a different one.
If you’re running into trouble running the manager, selecting the c220g5 node may assist you
- For “finalize” you can just click next
- For “schedule,” you don’t have to make a schedule, leaving it blank will start it immediately

To test your connection, you can connect Via a terminal

- Open a VSCode terminal and cd inside your .ssh folder
- `ssh -p 22 -i <privatekeyname> <user>@<address>`
- Your <user>@<address> can be found by going to your experiment and clicking on “list view,” it should be under “SSH Command”

Current Usage: 0 Node Hours, [Prev Week: 64](#), [Prev Month: 207](#) (30 day rank: 282 of 463 users)

▼ Your experiment is ready!

Name: `elliott1-100343`
 State: `ready`
 Profile: `onvm`
 Creator: `elliott1`
 Project: `GWCloudLab`
 Created: `Jun 12, 2021 11:13 AM`
 Started: `Jun 12, 2021 11:13 AM`
 Expires: `Jun 13, 2021 3:13 AM (in 16 hours)`

[Logs](#) [Create Disk Image](#) [Copy](#) [Extend](#) [Terminate](#)

► Profile Instructions

Topology View | List View | Manifest | Graphs | Bindings

ID	Node	Type	Status	Startup	Image	SSH command (if you provided your own key)	<input type="checkbox"/>	Actions
node1	c220g5-111325	c220g5	ready	n/a	gwcloudlab-PGO/ONVM2010_U20_build	<code>ssh -p 22 elliott1@c220g5-111325.wisc.cloudlab.us</code>	<input type="checkbox"/>	+

Powered by [emulab](#) Question or comment? Join the Help Forum Supported by NSF © 2021 The University of Utah

7.4.4 Connecting to the Node via VSCode

Before connecting, you must have uploaded your SSH key, and started an experiment. You also must have these VSCode extensions:

Remote - SSH Remote - SSH: Editing Configuration Files (may come preinstalled with Remote SSH)

These aren't necessary but may be helpful in the future:

Remote - Containers Remote - WSL (if using Windows)

Connecting Via a Remote Window

- Open the “Remote Explorer” via the sidebar (on the left by default)
- In the drop-down window at the top, select SSH Targets

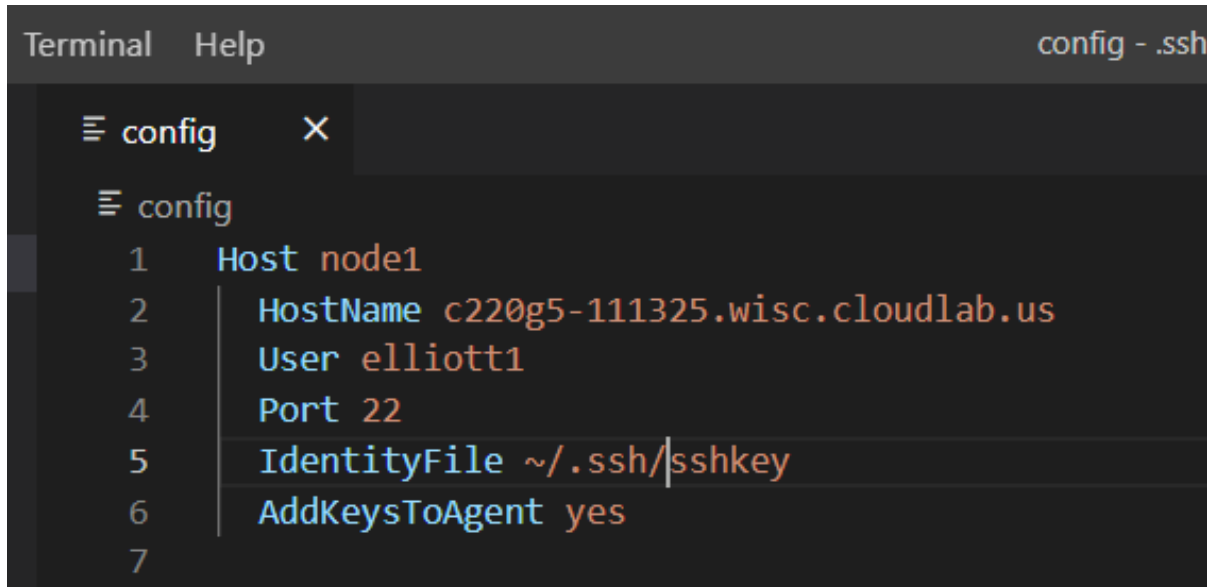
Visual Studio Code interface showing the Remote Explorer sidebar on the left. The 'SSH Targets' dropdown is selected. The main editor area displays the 'Window' settings page. The terminal at the bottom shows a PowerShell session with the following output:

```
PS C:\Users\jellio\Documents\openNetVM> cd docs
PS C:\Users\jellio\Documents\openNetVM\docs> cd source
cd : Cannot find path 'C:\Users\jellio\Documents\openNetVM\docs\source' because it does not exist.
At line:1 char:1
+ cd source
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (C:\Users\jellio\...VM\docs\source:String) [Set-Location], ItemNotFoundException
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.SetLocationCommand

PS C:\Users\jellio\Documents\openNetVM\docs> cd source/images
PS C:\Users\jellio\Documents\openNetVM\docs\source\images>
```

- To the right of the SSH Targets bar, click the plus button, and enter `ssh <user>@<address>`
- Select a configuration file (recommended to use the one in the `.ssh` folder as mentioned earlier)
- **Modify the config file so that it has the correct settings:**
It should have `Port 22 IdentityFile <privatekeyname>` and `AddKeysToAgent Yes` (on separate lines)

You can also rename `Host` to whatever you want, but `HostName` must not be changed

A screenshot of a terminal window titled "Terminal Help" with a tab labeled "config - .ssh". The terminal shows the contents of an SSH configuration file. The first line is "Host node1". The following lines are indented and show: "HostName c220g5-111325.wisc.cloudlab.us", "User elliot1", "Port 22", "IdentityFile ~/.ssh/sshkey", and "AddKeysToAgent yes".

```
Terminal Help config - .ssh
≡ config X
≡ config
1 Host node1
2   HostName c220g5-111325.wisc.cloudlab.us
3   User elliot1
4   Port 22
5   IdentityFile ~/.ssh/sshkey
6   AddKeysToAgent yes
7
```

- If it asks you to choose an operating system, select Linux

7.4.5 Running the ONVM Manager and Speed Tester NF on the node

Once you are properly connected to the node, it's time to run the manager

- First, `cd` into `/local/onvm/openNetVM/scripts` and run `source setup_cloudlab.sh`
- Depending on which node you're using, it will ask you to bind certain network devices to `dppk`
For this guide, we won't be working with real network traffic so we do not need to bind any ports
When working with 2+ nodes, you want to make sure that the two 10 GbE devices are bound (the letters/numbers before listing the device can be used as identifiers)
- Go to `/local/onvm/openNetVM/onvm` and run `make`
- Go to `/local/onvm/openNetVM/examples` and run `make`
- **Go to `/local/onvm/openNetVM` and run `sudo ./onvm/go.sh -k 1 -n 0xF8 -s stdout`**
If this gives you an error, it may be an issue with the pre-made profile, and you may have to pull a new onvm profile from GitHub in a new directory
Instructions on how to do so can be found [here](#)

The screenshot shows the 'TERMINAL' tab of the openNetVM interface. It displays network statistics for Port 0. The interface includes tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL, and PORTS. The terminal output shows the following:

```

PORTS
-----
Port 0: '00:5d:73:f0:12:93'

Port 0 - rx:      0 (      0 pps) tx:      0 (      0 pps)

NF TAG          IID / SID / CORE  rx_pps / tx_pps  rx_drop / tx_drop  out / tonf / drop
-----

```

- To run the speed tester, open a new tab while the manager is running and go to `/local/onvm/openNetVM/examples/speed_tester`
- run `./go.sh 1 -d 1 -c 16000`

7.5 Three Node Topology Tutorial

7.5.1 Getting Started

- This tutorial assumes that you have access to [CloudLab](#) and a basic working knowledge of CloudLab and [SSH](#)
- Also ensure that you have followed the development environment configuration instructions in our [Dev Environment Wiki](#) page
 - Specifically, make sure that you follow the steps in “Getting Started”, “Setup OpenSSH”, and everything up to instantiating an experiment in the “CloudLab Work Environment” section

7.5.2 Video Tutorial

If you want to follow along with this tutorial, we have created a video for your convenience.

7.5.3 Instantiating the ONVM CloudLab Profile

- Click [here](#) for the ONVM CloudLab Profile
- Click “Instantiate”

Current Usage: 6.32 Node Hours, Prev Week: 264, Prev Month: 1107 (30 day rank: 142 of 483 users)

Profile GWCloudLab/ONVM20_U20

Topology [Visualize](#) [View Source](#) [View XML](#)

Description A chain of servers running OpenNetVM v20.05 (Release Notes), mTCP, and DPDK v20.05 on Ubuntu 20.04 LTS. Each server has tools such as iperf and nghttp.

Instructions Specify the chain length (minimum of 1).
To initialize OpenNetVM run:

```
cd /local/onvm/opennetvm/scripts
source setup_cloudlab.sh
sudo ifconfig ethxxx down
./setup_environment.sh
```

where **ethxxx** is the NIC(s) you would like to bind to DPDK.
Tested on the Wisconsin site using C220g1 and C220g2 nodes.

Parameters

- Number of Hosts (minimum 1)
(default value: 1)
- Host type (e.g., c220g2 in Wisconsin site)
(default value: c220g2)

[Share](#) [Copy](#) [Instantiate](#)

Powered by [emulab](#) Question or comment? Join the Help Forum Supported by NSF © 2020 The University of Utah

- Enter the number of hosts you want — for a three-node topology, enter “3”
- Ensure that the host type is “c220g1,” “c220g2,” or “c220g5” (if available)

1. Select a Profile 2. Parameterize 3. Finalize 4. Schedule

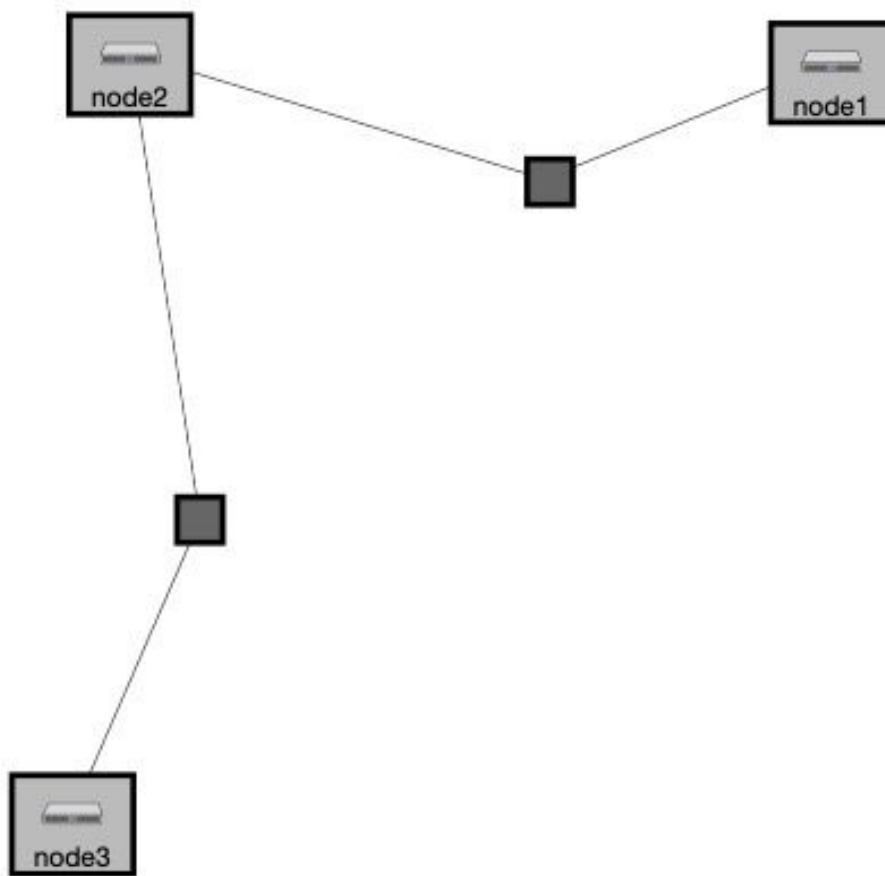
This profile is parameterized; please make your selections below, and then click **Next**. [Resource Availability](#) [Defaults](#) [Last](#) [History](#)

Number of Hosts (minimum 1)

Host type (e.g., c220g2 in Wisconsin site)

[Previous](#) [Next](#)

- Click “Next”
 - The generated topology image should somewhat resemble the image below



- Optionally, enter a name for the experiment in the “Name” field
- Click “Next”
- Click “Finish”
- Wait for the experiment to boot up

7.5.4 Connecting to CloudLab in Visual Studio Code via SSH

- Click “List View” to see the SSH commands to connect to your nodes

Topology View List View Manifest Graphs Bindings									
ID	Node	Type	Status	Startup	Image	SSH command (if you provided your own key)			Actions
node1	c220g2-011020	c220g2	Ready	Finished	gvccloudlab-P000onvm19.05	ssh -p 22 ethan@6c220g2-011020.wisc.cloudlab.us	<input type="checkbox"/>	<input type="checkbox"/>	SSH
node2	c220g2-011019	c220g2	Ready	Finished	gvccloudlab-P000onvm19.05	ssh -p 22 ethan@6c220g2-011019.wisc.cloudlab.us	<input type="checkbox"/>	<input type="checkbox"/>	SSH
node3	c220g2-011018	c220g2	Ready	Finished	gvccloudlab-P000onvm19.05	ssh -p 22 ethan@6c220g2-011018.wisc.cloudlab.us	<input type="checkbox"/>	<input type="checkbox"/>	SSH

- Ensure that your generated SSH command works by running it in terminal

For development within the Visual Studio Code environment:

- See more detailed setup instructions in our [Dev Environment Wiki](#) if you wish to use the VS Code environment for your setup

The following steps should be performed **for each node**:

- Copy relevant information into your `~/.ssh/config` file:

```
1 Host NodeXAddress
2   HostName NodeXAddress
3   Port 22
4   User CloudLabUsername
5   IdentityFile ~/.ssh/PrivateKeyFile
6   AddKeysToAgent yes
```

- Note that you can add other options as necessary

- Open Visual Studio Code
- Click the green Remote-SSH extension button (SSH logo) in the bottom-left corner
- Select Remote-SSH: **Connect to Host** from the options that appear in the command palette
- Select the address of the node you want to connect to
- Visual Studio Code will automatically connect and set itself up
 - See [Troubleshooting Tips](#) for connection issues and [Fixing SSH File Permissions](#) for permissions errors
- Once connected, navigate to the openNetVM repository folder: `cd /local/onvm/openNetVM`
- Now, finish configuring your workspace by selecting **File** → **Open** or **File** → **Workspace** and selecting the openNetVM folder (`/local/onvm/openNetVM`)

7.5.5 Setting Up a Three-Node Topology

The goal of this document is to configure the three nodes so that the first can act as a client, the third as a server, and the second node will act as a middlebox running OpenNetVM. The first and third nodes will use the kernel network stack, while the second will use DPDK.

Ensuring That Nodes Are Connected

- Connect to your CloudLab nodes in either Visual Studio Code or any SSH client
- With a three-node topology, your first node (node1) should be connected to one port in your second node (node2) and your third node (node3) should be connected to the other port in your second node (node2). Notice that this forms a “chain-like” structure like the one visualized in the topology image generated by CloudLab
- To determine which NICs are connected on each node, SSH into the node and run `ifconfig`

```

ethanb@node1:~$ ifconfig
docker0  Link encap:Ethernet  HWaddr 02:42:d9:da:32:11
          inet addr:172.17.0.1  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet  HWaddr 70:e4:22:83:62:34
          inet addr:128.105.145.94  Bcast:128.105.147.255  Mask:255.255.252.0
          inet6 addr: fe80::72e4:22ff:fe83:6234/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3813665 errors:0 dropped:0 overruns:24 frame:0
          TX packets:6903 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:229952857 (229.9 MB)  TX bytes:1093909 (1.0 MB)
          Memory:c6a00000-c6b00000

eth1      Link encap:Ethernet  HWaddr 90:e2:ba:b3:21:e0
          inet addr:192.168.1.1  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::92e2:baff:feb3:21e0/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:89 errors:0 dropped:0 overruns:0 frame:0
          TX packets:109 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:8418 (8.4 KB)  TX bytes:9558 (9.5 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:554 errors:0 dropped:0 overruns:0 frame:0
          TX packets:554 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:31302 (31.3 KB)  TX bytes:31302 (31.3 KB)

ethanb@node1:~$

```

- The connected NIC is the one with the local IP subnet. For the first node, it should be 192.168.1.1

- * Note that the local subnet is 192.168.1.x. This means that each of the NICs should have their `inet addr` field in the `ifconfig` command output start with 192.168.1..

- * For each NIC in the connection chain, the IP address should be 192.168.1.<previous + 1>. This means that the first should be 192.168.1.1, the second should be 192.168.1.2, and so on. Note that since node2 (and any other intermediate nodes in the case of a chain with more than three nodes) has two NICs configured for this, it will have two NICs with local addresses. This is seen in the below screenshot.

```

ethanb@node2:~$ ifconfig
docker0  Link encap:Ethernet  HWaddr 02:42:60:de:86:23
          inet addr:172.17.0.1  Bcast:172.17.255.255  Mask:255.255.0.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet  HWaddr 90:e2:ba:aa:fb:c8
          inet addr:192.168.1.2  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::92e2:baff:feaa:fbcb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:906 (906.0 B)

eth1      Link encap:Ethernet  HWaddr 90:e2:ba:aa:fb:c9
          inet addr:192.168.1.3  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::92e2:baff:feaa:fbcb/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:11 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:1170 (1.1 KB)

eth2      Link encap:Ethernet  HWaddr 70:e4:22:83:f6:5a
          inet addr:128.105.145.42  Bcast:128.105.147.255  Mask:255.255.252.0
          inet6 addr: fe80::72e4:22ff:fe83:f65a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:470448 errors:0 dropped:0 overruns:16 frame:0
          TX packets:1332 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:28482512 (28.4 MB)  TX bytes:197089 (197.0 KB)
          Memory:c6a00000-c6b00000

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:82 errors:0 dropped:0 overruns:0 frame:0
          TX packets:82 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:6273 (6.2 KB)  TX bytes:6273 (6.2 KB)

ethanb@node2:~$

```

- * The NIC names and ports (e.g. eth0 or eth1) can be completely random, but always have the local IP address mask (start with 192.168.1)

Bind Intermediate Nodes to DPDK

Before running the ONVM manager, we need to ensure that the connected NICs on node2 are bound to DPDK. DPDK has a script to determine whether NICs are bound or not.

- Identify which NICs are connected to the other nodes using `ifconfig` on node2 and checking the `inet addr` against the expected output above
- Navigate to the openNetVM folder that comes pre-installed on each node using `cd /local/onvm/openNetVM`
- Pull the most recent version of openNetVM from GitHub: `git pull origin master`
- Unbind the connected NICs: `sudo ifconfig ethxxx down`
- Run the ONVM `setup_environment.sh` script
 - `cd scripts`
 - `source ./setup_clouddlab.sh`
 - `./setup_environment.sh`

```
ethanb@node2:/local/onvm/openNetVM/scripts$ ./setup_environment.sh
Setting up hugepages
Removing currently reserved hugepages
Unmounting /mnt/huge and removing directory
Reserving hugepages
Creating /mnt/huge and mounting as hugetlbfs
Huge pages successfully configured
igb_uio 13476.0 - Live 0x0000000000000000 (0X)
IGB UIO module already loaded.
Checking NIC status

Network devices using kernel driver
=====
0000:01:00.0 'I350 Gigabit Network Connection 1521' if=eth2 drv=igb unused=igb_uio "Active"
0000:01:00.1 'I350 Gigabit Network Connection 1521' if=eth3 drv=igb unused=igb_uio
0000:06:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10Fb' if=eth0 drv=ixgbe unused=igb_uio
0000:06:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10Fb' if=eth1 drv=ixgbe unused=igb_uio

No 'Crypto' devices detected
=====

No 'Eventdev' devices detected
=====

No 'Mempool' devices detected
=====

No 'Compress' devices detected
=====

Binding NIC status
Bind interface 0000:06:00.0 to DPDK? [y/N] y
Binding 0000:06:00.0 to dpdk
Bind interface 0000:06:00.1 to DPDK? [y/N] y
Binding 0000:06:00.1 to dpdk
Finished Binding

Network devices using DPDK-compatible driver
=====
0000:06:00.0 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10Fb' drv=igb_uio unused=
0000:06:00.1 '82599ES 10-Gigabit SFI/SFP+ Network Connection 10Fb' drv=igb_uio unused=

Network devices using kernel driver
=====
0000:01:00.0 'I350 Gigabit Network Connection 1521' if=eth2 drv=igb unused=igb_uio "Active"
0000:01:00.1 'I350 Gigabit Network Connection 1521' if=eth3 drv=igb unused=igb_uio

No 'Crypto' devices detected
=====

No 'Eventdev' devices detected
=====

No 'Mempool' devices detected
=====

No 'Compress' devices detected
=====

Disabling hyperthreading...
CPU(s): 40
Thread(s) per core: 1
Core(s) per socket: 10
Socket(s): 2
Environment setup complete.
ethanb@node2:/local/onvm/openNetVM/scripts$
```


- Ensure that you see the **two** NICs in the “Network devices using DPDK-compatible driver”
 - If you only see one NIC, it’s possible that you did not unbind the other NIC from the kernel driver using `sudo ifconfig ethxxx down`. Instructions for that are above.

7.5.6 Verifying Node Chain Connections with openNetVM

Run the openNetVM Manager and Bridge NF

In the case of the three-node topology, we only need to run openNetVM on node2. These instructions should only be performed on all intermediate nodes in a longer chain.

- Navigate to the openNetVM folder: `cd /local/onvm/openNetVM`
- Compile the Manager: `cd onvm && make && cd ..`
- Compile the NFs: `cd examples && make && cd ..`
- Run the Manager: `./onvm/go.sh 0,1,2 3 0xF8 -s stdout`
 - The manager should show both ports running

```
PORTS
-----
Port 0: '90:e2:ba:aa:fb:c8'   Port 1: '90:e2:ba:aa:fb:c9'
Port 0 - rx:      4 (      0 pps) tx:      0 (      0 pps)
Port 1 - rx:      4 (      0 pps) tx:      0 (      0 pps)
NF TAG          IID / SID / CORE  rx_pps / tx_pps  rx_drop / tx_drop  out / tonf / drop
-----

```

- In another terminal pane, run the Bridge NF
 - `cd examples/bridge`
 - `./go.sh 1 1`

```
ethanb@node2:/local/onvm/openNetVM/examples/bridge$ ./go.sh 1 1
[Press Ctrl-C to quit ...]
EAL: Detected 20 1core(s)
EAL: Detected 2 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket_36578_d0a7b3d9f54a6
EAL: Probing VFIO support...
EAL: PCI device 0000:01:00.0 on NUMA socket 0
EAL:   probe driver: 8086:1521 net_e1000_igb
EAL: PCI device 0000:01:00.1 on NUMA socket 0
EAL:   probe driver: 8086:1521 net_e1000_igb
EAL: PCI device 0000:06:00.0 on NUMA socket 0
EAL:   probe driver: 8086:10fb net_ixgbe
EAL: PCI device 0000:06:00.1 on NUMA socket 0
EAL:   probe driver: 8086:10fb net_ixgbe
cur_index:1, action:2, destination:1

APP: Waiting for manager to assign an ID...
APP: Using Instance ID 2
APP: Using Service ID 1
APP: Running on core 3
APP: Finished Process Init.
Sending NF_READY message to manager...

```

Ping Between Nodes in Chain

When the ONVM Manager and Bridge NF are running, we can ping from node1 to node3, using node3's local IP address, despite node1 and node3 not being directly connected. We can also ping node1 from node3 using node1's local IP address. The following steps can be performed on either node1 or node3. Just ensure that you are using the opposite node's direct IP address. The direct IP of node1 should be 192.168.1.1 and the direct IP of node3 should be 192.168.1.4. Since these are not bound to DPDK, we can still verify this by running `ipconfig` on either node.

- Ping the opposite node: `ping 192.168.1.x` where `x` is the node's NIC number in the chain. You will see the number of packets sent updated in the manager
 - Note that there is no output in node3. You can verify that openNetVM is enabling the connections by closing the Manager and/or Bridge NF and repeating the ping command

7.6 Packet Generation Using Pktgen

Pktgen (*Packet Generator*) is a convenient traffic generation tool provided by DPDK and its fast packet processing framework. Pktgen's user-friendly nature and customizable packet generation structure make it a great option for multi-node testing purposes.

7.6.1 Getting Started

In order to get Pktgen up and running, please be sure to follow the step-by-step [Pktgen installation guide](#).

7.6.2 Transmitting Packets

Pktgen in ONVM supports the use of up to 2 ports for packet transmission. Before getting started, make sure that at least one 10Gb NIC is bound to the `igb_uio` module. You will need two 10Gb NICs bound if you intend to run Pktgen with 2 ports. Both nodes you are using to run Pktgen must have the same NIC ports bound to DPDK.

You can check the status of your NIC port bindings by running `sudo ./dpdk/usertools/dpdk-devbind.py --status` from the home directory of openNetVM. More guidance on how to bind/unbind NICs is provided in the [ONVM Install Guide](#).

The set up of Pktgen will require 2 nodes, Node A and Node B. Node A will run Pktgen, while Node B will run the ONVM manager along with any NFs.

As noted, both nodes must be bound to the same NIC port(s).

7.6.3 Running Pktgen with 1 Port

1. Start the manager on Node B with 1 port: `./go.sh 0,1,2 1 0xF0 -s stdout`
2. On Node A, modify `/tools/Pktgen/OpenNetVM-Scripts/pktgen-config.lua` (the Lua configuration script) to indicate the correct MAC address

```
pktgen.set_mac("0", "aa:bb:cc:dd:ee:ff")
```

where `aa:bb:cc:dd:ee:ff` is the port MAC address visible on the manager interface. 3. Run `./run-pktgen.sh 1` on Node A to start up Pktgen 4. Enter `start all` in the Pktgen command line prompt to begin packet transmission 5. To start an NF, open a new shell terminal off of Node B and run the desired NF using the respective commands see

our [NF Examples](#) folder for a list of all NFs and more information). ***Note:** *If packets don't appear to be reaching the NF, try restarting the NF with a service ID of 1.*

To stop the transmission of packets, enter `stp` in the `Pktgen` command line on Node A. To quit `Pktgen`, use `quit`.

7.6.4 Running Pktgen with 2 Ports

Before running the openNetVM manager, assure that both Node A and B have the same two 10Gb network interface bound to the DPDK driver (see `./dpdk/usertools`).

Running `Pktgen` with 2 ports is very similar to running it with just a single port.

1. Start the manager on Node B with a port mask argument of 3: `./go.sh 0,1,2 3 0xf0 -s stdout` (The 3 will indicate that 2 ports are to be used by the manager)
2. On Node A, modify the [Lua configuration script](#) to have the correct MAC addresses

```
1 pktgen.set_mac("0", "aa:bb:cc:dd:ee:ff");
2 pktgen.set_mac("1", "aa:bb:cc:dd:ee:ff");
```

where 0 and 1 refer to the port number and `aa:bb:cc:dd:ee:ff` refer to each of their MAC addresses, as visible on the manager interface.

3. Run `./run-pktgen.sh 2` on Node A, where 2 indicates the use of 2 ports.
4. To begin transmission, there are a couple of options when presented with the `Pktgen` command like on Node A:
 - `start all` will generate and send packets to both ports set up
 - `start 0` will only send generated packets to Port 0
 - `start 1` will only send generated packets to Port 1

Once one of the above commands is enter, packet transmission will begin 5. To start an NF, open a new shell terminal off of Node B and run the desired NF using the respective commands (see our [NF Examples](#) folder for a list of all NFs and more information). **Note:** *If packets don't appear to be reaching the NF, try restarting the NF with a service ID of 1.*

To stop the transmission of packets, enter `stp` in the `Pktgen` command line on Node A. To quit `Pktgen`, use `quit`.

7.6.5 Cloudlab Tutorial: Run Bridge NF with Pktgen using 2 Ports

The demo will be conducted on [CloudLab](#) and will follow the basic 2 Node `Pktgen` steps outlined above.

ONVM's [Bridge NF](#) is an example of a basic network bridge. The NF acts a connection between two ports as it sends packets from one port to the other.

1. Instantiate a Cloudlab experiment using the `2nodes-2links` profile made available by GWCloudLab. (**Note:** *If you do not have access to this profile, instructions on how to create it are outlined below.*)
2. On each node, install [ONVM](#) and [Pktgen](#) as per the instructions provided in the installation guides.

Node A will be our designated `Pktgen` node, while Node B will run the ONVM manager and NFs

3. Check the NIC port status on each node to ensure that both nodes have the same two 10Gb NICs bound to DPDK. The NIC statuses on each node should look similar to this:

From here, we should be able to follow the steps outlined in the above section.

4. On Node B, start the manager with `./go.sh 0,1,2 3 0xf0 -s stdout`:

5. On Node A, modify the Lua Pktgen configuration script. Ensure that both MAC addresses in the script match those printed by the ONVM manager:

Manager Display:

```

1 PORTS
2 -----
3 Port 0: '90:e2:ba:87:6a:f0'      Port 1: '90:e2:ba:87:6a:f1'

```

Pktgen Config File:

```

1 pktgen.set_mac("0", "90:e2:ba:87:6a:f0");
2 pktgen.set_mac("1", "90:e2:ba:87:6a:f1");

```

6. On Node A, start Pktgen with `./run-pktgen`. Enter `start all` in the command line. This should start sending packets to both Ports 0 and 1.

7. With the ONVM manager running, open a new Node B terminal and start the Bridge NF: `./go.sh 1`. If you observe the ONVM manager stats, you will notice that, with the Bridge NF, packets sent to Port 0 will be forwarded to and received at Port 1, while packets sent to Port 1 will be received at Port 0. The results should look similar to this:

```

1 PORTS
2 -----
3 Port 0: '90:e2:ba:87:6a:f0'      Port 1: '90:e2:ba:87:6a:f1'
4
5 Port 0 - rx: 32904736 ( 71713024 pps)  tx: 32903356 ( 7173056 pps)
6 Port 1 - rx: 32903840 ( 71713024 pps)  tx: 32904348 ( 7173056 pps)
7
8 NF TAG          IID / SID / CORE      rx_pps / tx_pps      rx_drop / tx_drop
9  ↳ out /      tonf / drop
10 -----
11  ↳ -----
12 bridge          1 / 1 / 4             14345920 / 14345888      0 / 0
13  ↳ 131176508 / 0 / 0

```

If you'd like to see the Bridge NF working more clearly, you can try sending packets to only one port with either `start 0` or `start 1`. This will allow you to see how the Rx count changes with Bridge NF as more packets arrive.

At any point, enter `stp` in the Pktgen command line (Node A) if you'd like to stop the transmission of packets. Use `quit` to quit Pktgen completely.

7.6.6 Customizing Packets

Several user-friendly customization functions are offered through the [Lua configuration file](#).

7.6.7 Setting Packet Size

Setting the byte size of packets is important for performance testing. The Lua configuration file allows users to do this: `pktgen.set("all", "size", 64);`

7.6.8 Specifying Protocol

If you wish to specify the protocol of each packet, this can be done by modifying the following configuration: `pktgen.set_proto("all", "udp");` **Note:** Pktgen currently supports TCP/UDP/ICMP protocols.

7.6.9 Number of Packets

You may specify the number of packets you want transmit with: `pktgen.set("all", "count", 100000);` This indicates that you'd like to transmit 100,000 packets.

All other customization options can be found by entering `all` in the Pktgen command line

7.6.10 Create a 2 Node CloudLab Profile for Pktgen

1. Log into your [CloudLab](#) profile
2. On the User Dashboard, select "Create Experiment Profile" under the "Experiments" tab
3. Enter a name for your profile and choose "Create Topology" next to Source Code
4. Drag 2 Bare Metal PCs onto the Site
5. Link the two nodes by dragging a line between them to connect them. Do this twice. This will ensure that you've created two links between the nodes, allowing for a 1 or 2 port Pktgen setup.
6. **Click on one of the nodes to customize it as specified below. Repeat this step for the second node as well.**
 - Hardware Type: c220g1 or c220g2
 - Disk Image: Ubuntu 18-64 STD (or whichever version the latest ONVM version requires)
7. Accept the changes and create your profile

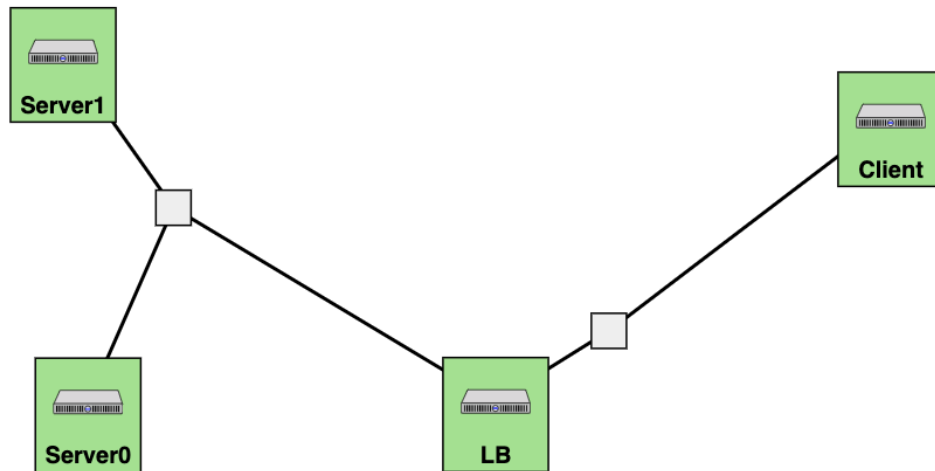
7.7 Load Balancer Tutorial

7.7.1 Getting Started

- This tutorial assumes that you have access to [CloudLab](#) and a basic working knowledge of CloudLab and [SSH](#)

7.7.2 Overview

- In the following tutorial, we will explore a means of deploying and testing ONVM's example Layer-3 round-robin load balancer. To do this, we will instantiate a Cloudlab experiment using the `ONVM_LoadBalancer` profile; this topology (shown below) includes two backend servers and a single client, in addition to the ONVM load balancer.

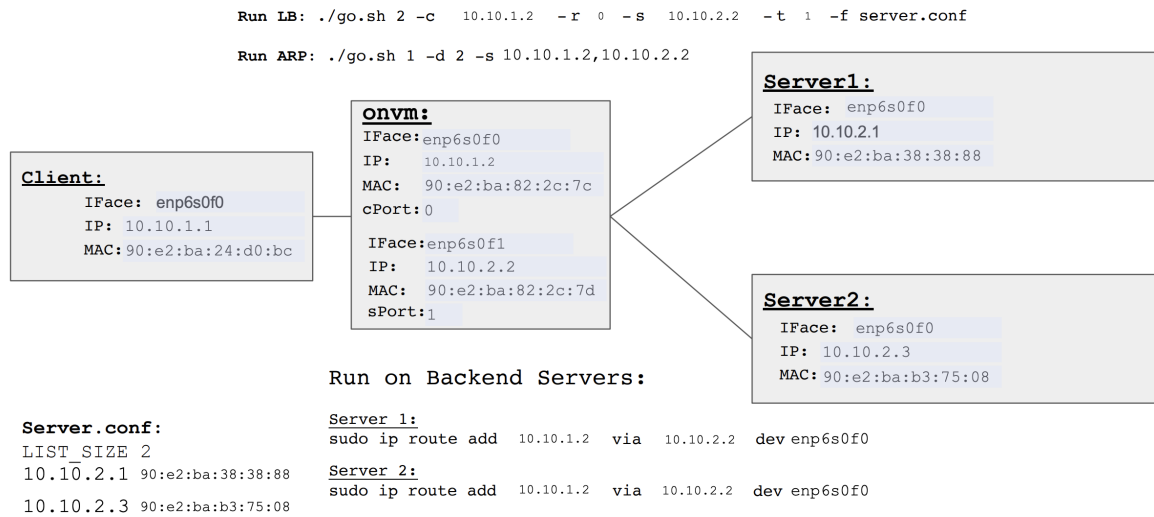


- After completion of the step-by-step guide below, you may use a number of packet generation tools to test the successful distribution of packets across multiple servers. The tools covered in this tutorial include iPerf and Pktgen, the latter being further described in the previous tutorial.

7.7.3 Cloudlab Node Setup

- Open Cloudlab and start a new experiment. When prompted to choose a profile, select `ONVM_LoadBalancer`. For this tutorial, set the number of backend servers to 2, and set the OS Image to `ONVM_UBUNTU20.04`. With regards to defining the physical node type, we will leave this blank and expect the default `c220g2` nodes. In the following section (under “Finalize”), select the Cloudlab Wisconsin cluster, and all remaining inputs are discretionary.
- Begin by SSHing into each node within the experiment, and download the **Load Balancer Topology Template** [here](#). If you are using any Apple product to complete this tutorial, avoid using Preview as your PDF editor; autofill scripts will not apply. Google Chrome or Adobe Acrobat are viable alternatives.
- For every node, use `ifconfig` to view all available network interfaces. Record the appropriate name, IPv4 (inet), and MAC address (ether) for each network interface in the Topology Template, as shown below. Note that the client side and server side nodes should be on a different IP subnets. The `ONVM_LB` node requires the use of two ports: one for connection to the client and one for connecting to the servers. It is recommended that you use the 10-Gigabit SFI/SFP+ network connections. Port IDs will be handled later.

Topology



- In the ONVM LB node, set up the environment using `setup_clouddlab.sh` in the scripts directory. Once the ports have been successfully bound to the DPDK-compatible driver, start the manager with at least two available ports. Listed below are the abbreviated steps for binding available ports to the DPDK-bound driver. To start the manager, you may use `./onvm/go.sh -k 3 -n 0xFF -s stdout`.
 - Unbind the connected NICs: `sudo ifconfig <IFACE> down` where <IFACE> represents the interface name (eg. `enp6s0f0`)
 - Navigate to the `/local/onvm/openNetVM/scripts` directory and bind the NICs to DPDK using the command `source ./setup_clouddlab.sh`
 - Ensure that you see the two NICs in the section defining “Network devices using DPDK-compatible driver.” If you only see one NIC, it’s possible that you did not unbind the other NIC from the kernel driver using `sudo ifconfig <IFACE> down`. Repeat step (i).
 - Navigate back to the openNetVM folder (`cd ..`) and compile the Manager using `cd onvm && make && cd ..`
- At the top of the manager display (pictured below), you can observe two (or more) port IDs and their associated MAC addresses. Use these ID mappings to complete the Port ID sections of the **Topology Template**.

```

PORTS
-----
Port 0: '90:e2:ba:82:2c:7c'    Port 1: '90:e2:ba:82:2c:7d'
  
```

- Now that the **Topology Template** is complete, all commands within the PDF document should be populated. To complete our LB configuration, we must:
 - Specify the backend servers’ port information in **server.conf**
 - Define a static route in each of the backend servers to specify the correct gateway for which traffic will enter.
- To specify the server information for the load balancer, go to `/examples/load_balancer/server.conf` and copy the information that is shown in the bottom left quadrant of your **Topology Template**. This includes the “`LIST_SIZE 2`” and the IP+MAC address of each port.

- To define the static routes, navigate to the two backend nodes (Server1 and Server2) and execute the respective commands shown on the bottom-center area of the **Topology Template**. This includes the `sudo ip route add * command` for each server.

7.7.4 Running The Load Balancer

- Since the ONVM environment is set, we can begin running the load balancer. In order to properly map IPs to HWAddresses, we must run the ARP NF. To do so, open a new terminal within the ONVM node; enter the `/examples/arp_response` directory and run the command shown at the top of the **Topology Template** labeled “*Run Arp:*”. Once properly running, the NF will appear as below:

```
[Press Ctrl-C to quit ...]
EAL: Detected 16 lcore(s)
EAL: Detected 2 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket_558827_34120b17d0c3c0
EAL: Selected IOVA mode 'PA'
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: Probe PCI driver: net_ixgbe (8086:10fb) device: 0000:06:00.0 (socket 0)
EAL: Probe PCI driver: net_ixgbe (8086:10fb) device: 0000:06:00.1 (socket 0)
EAL: No legacy callbacks, legacy socket not created
cur_index:1, action:2, destination:1

APP: Waiting for manager to assign an ID...
APP: Using Instance ID 1
APP: Using Service ID 1
APP: Running on core 0
APP: Finished Process Init.
APP: Sending packets to service ID 2
Sending NF_READY message to manager...
```

- We can now run the load balancer. Note that the ARP NF and the LB NF can be started in any order, but both NFs must be active in order for the load balancer to handle traffic. To run the load balancer, navigate to the `/examples/load_balancer` directory and run the command shown at the top of the **Topology Template** labeled “*Run LB:*”. The LB NF will appear as below:


```

[Press Ctrl-C to quit ...]
EAL: Detected 16 lcore(s)
EAL: Detected 2 NUMA nodes
EAL: Multi-process socket /var/run/dpdk/rte/mp_socket_559850_34129af033d578
EAL: Selected IOVA mode 'PA'
EAL: Probing VFIO support...
EAL: VFIO support initialized
EAL: Probe PCI driver: net_ixgbe (8086:10fb) device: 0000:06:00.0 (socket 0)
EAL: Probe PCI driver: net_ixgbe (8086:10fb) device: 0000:06:00.1 (socket 0)
EAL: No legacy callbacks, legacy socket not created
cur_index:1, action:2, destination:1

APP: Waiting for manager to assign an ID...
APP: Using Instance ID 2
APP: Using Service ID 2
APP: Running on core 0
APP: Finished Process Init.

Load balancer interfaces:
Client iface: ID: 0, IP: 33622538 (10.10.1.2), MAC: 90:e2:ba:82:2c:7c
Server iface: ID: 1, IP: 33688074 (10.10.2.2), MAC: 90:e2:ba:82:2c:7d

ARP config:
10.10.2.1 3c:fd:fe:b4:fa:4c
10.10.2.3 3c:fd:fe:b0:f1:74
Sending NF_READY message to manager...

```

- To check that the ports have properly been applied to the load balancer, you may confirm that the listed MAC addresses and Port IDs are correctly associated under “*Load balancer interfaces*” (in the picture above).

7.7.5 Testing The Load Balancer with iPerf (recommended):

- iPerf is a simple packet-generation tool which we may use to confirm that the load balancer is properly distributing traffic. To run iPerf, perform the following:
 - In the terminal of both backend servers, execute the command `iperf -s`. This will start a TCP server on each of the backend nodes.
 - Following, you may start the iPerf client on the client node using the command `iperf -c <X.X.X.X>` where the IP to fill is the client-side port on the ONVM node.
 - At this point, you should notice traffic being sent from the client and being received by one of the two servers. If you run the client multiple times, you should observe that the traffic is being distributed across each of the backend nodes evenly.
- iPerf Client

```

-----
Client connecting to 10.10.1.2, TCP port 5001
TCP window size: 85.0 KByte (default)
-----
[ 3] local 10.10.1.1 port 36190 connected with 10.10.1.2 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 3]  0.0-10.1 sec   131 MBytes  109 Mbits/sec

```

- iPerf Server

```
=====
Server listening on TCP port 5001
TCP window size: 128 KByte (default)
=====
[ 4] local 10.10.2.1 port 5001 connected with 10.10.1.1 port 36190
[ ID] Interval      Transfer    Bandwidth
[ 4]  0.0-10.2 sec   131 MBytes  108 Mbits/sec
```

- iPerf provides incremental throughput and bandwidth. Results can be seen below. Additional traffic information can be obtained by changing/adding command-line arguments, as discussed [here](#). This page also provides instructions for running a UDP Client and Server, rather than TCP.

7.7.6 Testing The Load Balancer with Pktgen:

- In accordance with the previous tutorial, we can use Pktgen to generate fake packets which will allow us to perform more throughput-intensive testing. Using the Pktgen tutorial, follow the directions regarding “*Running Pktgen with 1 Port.*” Ensure that Pktgen is running on the client node, and the indicated port in `/tools/Pktgen/openNetVM-Scripts/pktgen-config.lua` corresponds to the client-side port on the main ONVM node (which is running the manager). For further detail, follow the instructions below:
 - In the following, we will refer to the client node as Node A and the ONVM node as Node B
 - On Node B, start the manager, the ARP NF, and the load balancer.
 - On Node A, ensure that the one port (which you intend to send packets through) is bound to the DPDK-compatible driver. Then, go to `/tools/Pktgen/openNetVM-Scripts/pktgen-config.lua` and add the client-side port ID and Mac Address (from the ONVM node) into the script, as shown below.

```
-- set up a mac address to set flow to
--
-- TO DO LIST:
--
-- Please update this part with the destination mac address, source and destination ip address you would like to sent packets to
pktgen.set_mac("0", "90:e2:ba:82:2c:7c");
```

- In the same `/OpenNetVM-Scripts` directory, execute the command `./run-pktgen.sh 1`. This will begin Pktgen, and you can start the traffic by executing `start all`.
 - If Pktgen cannot successfully start, reference the [installation guide](#) for additional help.
- Once Pktgen is running, you should be able to view the flow of traffic on the manager, as they are received on the client-side port and sent on the server-side port. If you would like to get further information, you can run the command `sudo tcpdump -i <IFACE>` on each of the backend servers (where `<IFACE>` is the server's interface name) to view all incoming traffic.
- Please note that generation of fake packets on Cloudlab often causes many packets to be dropped, making the use of Pktgen unideal in some circumstances.

7.7.7 Troubleshooting:

- If you receive the error `connect failed: No route to host` when starting the iPerf client, it is possible that the ARP NF was unable to complete all of the necessary IP/HWAddress mappings. When running the ARP NF, please be sure that IPs are listed in the same order as the DPDK port numbers they correspond to. If this was not the issue, we can check whether the mappings are incomplete by executing `arp -n` in the command line of the client node. If the HWaddress resolves to `(incomplete)` (example shown below), then the MAC address must be mapped manually. Refer to the **Topology Template** to confirm the correct hardware address for the client-side ONVM port. Then, execute the command `sudo arp -s <X.X.X.X> <X:X:X:X:X>` where the first input is the ONVM client-side port IP and the second input is the client-side port MAC address. Using the template above, the arguments would be `sudo arp -s 10.10.1.2 90:e2:ba:82:2c:7c`. Additional manual mappings may also be needed on the backend nodes. The same process is applied, but the mapping will now correlate to the server-side ONVM port. Confirm that the HWaddress has now been added by running `arp -n`, and proceed with running the iPerf client again.

Address	Hwtype	Hwaddress	Flags	Mask	Iface
10.10.1.2		(incomplete)			enp94s0f0

7.8 Running OpenNetVM in Docker

To run openNetVM NFs inside Docker containers, use the included [Docker Script](#). We provide a [Docker image on Docker Hub](#) that is a copy of [Ubuntu 18.04](#) with a few dependencies installed. This script does the following:

- Creates a Docker container off of the [sdnfv/opennetvm Docker image](#) with a custom name
- Maps NIC devices from the host into the container
- Maps the shared hugepage region into the container
- Maps the openNetVM directory into the container
- Maps any other directories you want into the container
- Configures the container to use all of the shared memory and data structures
- Depending on the presence of a command to run, the container starts the NF automatically in detached mode or the terminal is connected to it to run the NF by hand

7.8.1 Usage

To use the script, simply run it from the command line with the following options:

```
sudo ./docker.sh -h HUGEPAGES -o ONVM -n NAME [-D DEVICES] [-d DIRECTORY] [-c COMMAND]
```

- **HUGEPAGES** - A path to where the hugepage filesystem is on the host
- **ONVM** - A path to the openNetVM directory on the host filesystem
- **NAME** - A name to give the container
- **DEVICES** - An optional comma delimited list of NIC devices to map to the container
- **DIRECTORY** - An optional additional directory to map inside the container.
- **COMMAND** - An optional command to run in the container. For example, the path to a go script or to the executable of your NF.

```
1 sudo ./docker.sh -h /mnt/huge -o /root/openNetVM -n Basic_Monitor_NF -D /dev/uio0,/dev/  
↪ uio1
```

- This will start a container with two NIC devices mapped in, /dev/uio0 and /dev/uio1, the hugepage directory at /mnt/huge mapped in, and the openNetVM source directory at /root/openNetVM mapped into the container with the name of Basic_Monitor_NF.

```
1 sudo ./docker.sh -h /mnt/huge -o /root/openNetVM -n Speed_Tester_NF -D /dev/uio0 -c "./  
↪ examples/start_nf.sh speed_tester 1 -d 1"
```

- This will start a container with one NIC device mapped in, /dev/uio0, the hugepage directory at /mnt/huge mapped in, and the openNetVM source directory at /root/openNetVM mapped into the container with the name of Speed_Tester_NF. Also, the container will be started in detached mode (no connection to it) and it will run the go script of the simple forward NF. Careful, the path needs to be correct inside the container (use absolute path, here the openNetVM directory is mapped in the /).

To remove all containers:

```
1 sudo docker rm $(sudo docker ps -aq)
```

To remove all docker images from the system:

```
1 # list all images  
2 sudo docker images -a  
3  
4 # remove specific image  
5 sudo docker rmi <IMAGE ID>  
6  
7 # clean up resources not associated with running container  
8 docker system prune  
9  
10 # clean up all resources  
11 docker system prune -a
```

7.8.2 Running NFs Inside Containers

To run an NF inside of a container, use the provided script which creates a new container and presents you with its shell. From there, if you run `ls`, you'll see that inside the root directory, there is an `openNetVM` directory. This is the same `openNetVM` directory that is on your host - it is mapped from your local host into the container. Now enter that directory and go to the example NFs or enter the other directory that you mapped into the container located in the root of the filesystem. From there, you can run the `go.sh` script to run your NF.

Some prerequisites are:

- Compile all applications from your local host. The Docker container is not configured to compile NFs.
- Make sure that the openNetVM manager is running first on your local host.

Here is an example of starting a container and then running an NF inside of it:

```
1 root@nimbnode /root/openNetVM/scripts# ./docker.sh  
2 sudo ./docker.sh -h HUGE_PAGES -o ONVM -n NAME [-D DEVICES] [-d DIRECTORY] [-c COMMAND]  
3  
4 e.g. sudo ./docker.sh -h /hugepages -o /root/openNetVM -n Basic_Monitor_NF -D /dev/uio0,/dev/uio1  
↪
```

- This will create a container with two NIC devices, uio0 and uio1, hugepages mapped from the host's /hugepage directory and openNetVM mapped from /root/openNetVM and it will name it Basic_Monitor_NF

```

1 root@nimbnode /root/openNetVM/scripts# ./docker.sh -h /mnt/huge -o /root/openNetVM -D /
  ↪ dev/uio0,/dev/uio1 -n basic_monitor
2 root@899618eaa98c:/openNetVM# ls
3 CPPLINT.cfg LICENSE Makefile README.md cscope.out docs dpdk examples onvm onvm_
  ↪ web scripts style tags tools
4 root@899618eaa98c:/openNetVM# cd examples/
5 root@899618eaa98c:/openNetVM/examples# ls
6 Makefile aes_decrypt aes_encrypt arp_response basic_monitor bridge flow_table flow_
  ↪ tracker load_balancer ndpi_stats nf_router simple_forward
7 speed_tester test_flow_dir
8 root@899618eaa98c:/openNetVM/examples# cd basic_monitor/
9 root@899618eaa98c:/openNetVM/examples/basic_monitor# ls
10 Makefile README.md build go.sh monitor.c
11 root@899618eaa98c:/openNetVM/examples/basic_monitor# ./go.sh 3 -d 1
12 ...

```

You can also use the optional command argument to run directly the NF inside of the container, without connecting to it. Then, to stop gracefully the NF (so it has time to notify onvm manager), use the docker stop command before docker rm the container. The prerequisites are the same as in the case where you connect to the container.

```

1 root@nimbnode /root/openNetVM# ./scripts/docker.sh -h /mnt/huge -o /root/openNetVM -n_
  ↪ speed_tester_nf -D /dev/uio0,/dev/uio1 -c
2 ".examples/speed_tester/go.sh 1 -d 1"
3 14daebba1adea581c2998eead16ff7ce7fdc45394c0cc5d6489228aad939711
4 root@nimbnode /root/openNetVM# sudo docker stop speed_tester_nf
5 speed_tester_nf
6 root@nimbnode /root/openNetVM# sudo docker rm speed_tester_nf
7 speed_tester_nf
8 ...

```

7.8.3 Setting Up and Updating Dockerfiles

If you need to update the Dockerfile in the future, you will need to follow these steps.

```

1 # install docker fully
2 sudo curl -sSL https://get.docker.com/ | sh

```

Make an update to scripts/Dockerfile. Create an image from the new Dockerfile.

```

1 # run inside scripts/
2 docker image build -t sdnfv/opennetvm:<some ID tag> - < ./Dockerfile

```

This command may take a while as it grabs the Ubuntu container, and installs dependencies. Test that the container built correctly. Go into scripts/docker.sh and temporarily change line 84

```

1 # from this
2 sdnfv/opennetvm \
3 # to this
4 sdnfv/opennetvm:<some ID tag> \

```

Make sure it is the same tag as the build command. This stops docker from pulling the real sdnfv/opennetvm

Test what you need to for the update and remove all containers.

```
1 sudo docker rm $(sudo docker ps -aq)
```

Create an account on Docker online and sign via CLI:

```
1 sudo docker login -u <username> docker.io
```

Make sure you are apart of the sdnfv Docker organization:

```
1 # push updated image
2 docker push sdnfv/opennetvm
3
4 # rename to update latest as well
5 docker tag sdnfv/opennetvm:<some ID tag> sdnfv/opennetvm
6 docker push sdnfv/opennetvm:latest
```

Now the image is updated, and will be the default next time someone pulls.

7.8.4 Older Dockerfiles

If you want to use an older ONVM version on Ubuntu 14, take a look at the [Available Tags](#). The 18.03 tag runs ONVM when it had been set up for an older version of Ubuntu. The latest dockerfile runs on Ubuntu 18.04 and is called latest.

7.9 MoonGen Installation (DPDK-2.0 Version)

Welcome to installation memo for [MoonGen](#), MoonGen is a “Scriptable High-Speed Packet Generator”.

7.9.1 1. Preparation steps

Installation steps are assuming that you have already have [OpenNetVM installed](#). If you have already have OpenNetVM installed, please following the steps below for a double check of your system.

1.1. Check if you have available hugepages

```
1 $grep -i huge /proc/meminfo
```

If **HugePages_Free** 's value equals to 0, which means there are no free hugepages available, there may be a few reasons why:

- The manager crashed, but an NF(s) is still running. In this case, either kill them manually by hitting Ctrl+C or run `$ sudo pkill NF_NAME` for every NF that you have ran.
- The manager and NFs are not running, but something crashed without freeing hugepages. To fix this, please run `$ sudo rm -rf /mnt/huge/*` to remove all files that contain hugepage data.
- The above two cases are not met, something weird is happening: a reboot might fix this problem and free memory:
`$ sudo reboot`

1.2. Check if you have available ports bound to DPDK

```
1 $cd directory_of_your_installed_dpdk
2 $./tools/dpdk_nic_bind.py --status
```

If you got the following similar binding information indicating that you have the two 10-Gigabit NIC ports binded with DPDK driver, jump to step 1.4, otherwise, please jump to step 1.3.

```
1 Network devices using DPDK-compatible driver
2 =====
3 0000:07:00.0 '82599EB 10-Gigabit SFI/SFP+ Network Connection' drv=igb_uio unused=ixgbe
4 0000:07:00.1 '82599EB 10-Gigabit SFI/SFP+ Network Connection' if=eth3 drv=ixgbe
5 ↪unused=ixgbe
6
7 Network devices using kernel driver
8 =====
9 0000:05:00.0 '82576 Gigabit Network Connection' if=eth0 drv=igb unused=igb_uio *Active*
10 0000:05:00.1 '82576 Gigabit Network Connection' if=eth0 drv=igb unused=igb_uio *Active*
```

1.3. Bind the 10G ports to DPDK

An example of incorrect bindings is as follows

```
1 Network devices using DPDK-compatible driver
2 =====
3 <none>
4
5 Network devices using kernel driver
6 =====
7 0000:05:00.0 '82576 Gigabit Network Connection' if=eth0 drv=igb unused=igb_uio *Active*
8 0000:05:00.1 '82576 Gigabit Network Connection' if=eth1 drv=igb unused=igb_uio
9 0000:07:00.0 '82599EB 10-Gigabit SFI/SFP+ Network Connection' if=eth2 drv=ixgbe
10 ↪unused=igb_uio *Active*
11 0000:07:00.1 '82599EB 10-Gigabit SFI/SFP+ Network Connection' if=eth3 drv=ixgbe
12 ↪unused=igb_uio
```

In our example above, we see two 10G capable NIC ports that we could use with description 82599EB 10-Gigabit SFI/SFP+ Network Connection.

One of the two NIC ports, 07:00.0, is active shown by the Active at the end of the line. Since the Linux Kernel is currently using that port–network interface eth2—we will not be able to use it with openNetVM. We must first disable the network interface in the Kernel, and then proceed to bind the NIC port to the DPDK Kernel module, igb_uio:

```
1 $ sudo ifconfig eth2 down
```

Rerun the status command, `./usertools/dpdk-devbind.py --status`, to see that it is not active anymore. Once that is done, proceed to bind the NIC port to the DPDK Kernel module:

```
1 $ sudo ./usertools/dpdk-devbind.py -b igb_uio 07:00.0
```

Check the status again, `$./usertools/dpdk-devbind.py --status`, and assure the output is similar to our example below:

```

1 Network devices using DPDK-compatible driver
2 =====
3 0000:07:00.0 '82599EB 10-Gigabit SFI/SFP+ Network Connection' drv=igb_uio unused=ixgbe
4
5 Network devices using kernel driver
6 =====
7 0000:05:00.0 '82576 Gigabit Network Connection' if=eth0 drv=igb unused=igb_uio *Active*
8 0000:05:00.1 '82576 Gigabit Network Connection' if=eth1 drv=igb unused=igb_uio
9 0000:07:00.1 '82599EB 10-Gigabit SFI/SFP+ Network Connection' if=eth3 drv=ixgbe
   ↪ unused=igb_uio

```

1.4. Check if g++ and gcc are updated with version higher than 4.7

```

1 $g++ --version
2 $gcc --version

```

if not, please add the repository using:

```

1 $sudo add-apt-repository ppa:ubuntu-toolchain-r/test

```

then, to install it use:

```

1 $sudo apt-get update
2 $sudo apt-get install g++-4.8

```

and then change the default compiler use update-alternatives:

```

1 $sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 40 --slave /usr/
   ↪ bin/g++ g++ /usr/bin/g++-4.8
2 $sudo update-alternatives --config gcc

```

Install other dependencies with:

```

1 $ sudo apt-get install -y build-essential cmake linux-headers-`uname -r` pciutils
   ↪ libnuma-dev
2 $ sudo apt install cmake
3 $ sudo apt install libtbb2

```

7.9.2 2. MoonGen Installation

2.1. Get the resource from github, and checkout the dpdk2.0 branch

```

1 $git clone https://github.com/emmericp/MoonGen
2 $cd MoonGen
3 $git checkout dpdk-19.05
4 $sudo git submodule update --init

```


2.2. Build the resource

```
$sudo ./build.sh
```

2.3. Set up hugetable

```
$sudo ./setup-hugetlbfs.sh
```

2.4. Execute the test

Configure the **quality-of-service-test.lua** with your destination ip address (ip address for the server you want to sent packets to) and your source ip address (ip address for the machine you are executing MoonGen on), and run with command:

```
$sudo ./build/MoonGen ./examples/quality-of-service-test.lua 0 1
```

and if the sample log outputs the following, your configuration is correct. Use **Ctrl+C** to stop generating packets:

```
wenhui@nimbnode16:~/MoonGen$ sudo ./build/MoonGen ./examples/quality-of-service-test.lua
↪ 0 0
Found 2 usable ports:
Ports 0: 00:1B:21:80:6A:04 (82599EB 10-Gigabit SFI/SFP+ Network Connection)
Ports 1: 00:1B:21:80:6A:05 (82599EB 10-Gigabit SFI/SFP+ Network Connection)
Waiting for ports to come up...
Port 0 (00:1B:21:80:6A:04) is up: full-duplex 10000 MBit/s
1 ports are up.
[Port 42] Sent 1460655 packets, current rate 1.46 Mpps, 1495.62 MBit/s, 1729.32 MBit/s
↪ wire rate.
[Port 43] Sent 97902 packets, current rate 0.10 Mpps, 100.18 MBit/s, 115.83 MBit/s wire
↪ rate.
[Port 42] Sent 2926035 packets, current rate 1.47 Mpps, 1500.54 MBit/s, 1735.00 MBit/s
↪ wire rate.
[Port 43] Sent 195552 packets, current rate 0.10 Mpps, 99.98 MBit/s, 115.61 MBit/s wire
↪ rate.
[Port 42] Sent 4391415 packets, current rate 1.47 Mpps, 1500.54 MBit/s, 1735.00 MBit/s
↪ wire rate.
.....
^C[Port 42] Sent 15327522 packets with 1961922816 bytes payload (including CRC).
[Port 42] Sent 1.465371 (StdDev 0.000010) Mpps, 1500.540084 (StdDev 0.009860) MBit/s,
↪ 1734.999472 (StdDev 0.011401) MBit/s wire rate on average.
[Port 43] Sent 1020600 packets with 130636800 bytes payload (including CRC).
[Port 43] Sent 0.097653 (StdDev 0.000017) Mpps, 99.996549 (StdDev 0.017340) MBit/s, 115.
↪ 621010 (StdDev 0.020049) MBit/s wire rate on average.
PMD: ixgbe_dev_tx_queue_stop(): Tx Queue 1 is not empty when stopping.
PMD: ixgbe_dev_tx_queue_stop(): Could not disable Tx Queue 0
PMD: ixgbe_dev_tx_queue_stop(): Could not disable Tx Queue 1
Background traffic: Average -9223372036854775808, Standard Deviation 0, Quartiles -
↪ -9223372036854775808/-9223372036854775808/-9223372036854775808
```

(continues on next page)

(continued from previous page)

```
24 Foreground traffic: Average -9223372036854775808, Standard Deviation 0, Quartiles -
    ↪ -9223372036854775808/-9223372036854775808/-9223372036854775808
```

7.10 Wireshark and Pdump for Packet Sniffing

Wireshark is a must have when debugging any sorts of complex multi node experiments. When running dpdk you can't run Wireshark on the interface but you can view pcap files. Try running the pdump dpdk application to capture packets and then view them in Wireshark.

1. Login using your normal ssh command but append -X -Y (when logging from nimbus jumpbox also include -X -Y)
2. Install Wireshark
3. Open it up (depending on terminal/os this might not work for everyone)

```
1 ssh your_node@cloudlab -X -Y
2 sudo apt-get update
3 sudo apt-get install wireshark
4 sudo wireshark
```

7.10.1 Packet capturing

When working with different packet protocols and TCP related applications it is often needed to look at the packets received/sent by the manager. DPDK provides a dpdk-pdump application that can capture packets to a pcap file.

To use dpdk-pdump set CONFIG_RTE_LIBRTE_PMD_PCAP=y in dpdk/config/common_base and then recompile dpdk.

Then execute dpdk-pdump as a secondary application when the manager is running

```
1 cd dpdk/x86_64-native-linuxapp-gcc
2 sudo ./build/app/pdump/dpdk-pdump -- --pdump 'port=0,queue=*,rx-dev=/tmp/rx.pcap,tx-dev=/
    ↪ tmp/tx.pcap'
```

Full set of options and configurations for dpdk-pdump can be found [here](#).

7.11 OpenNetVM Examples

7.11.1 NF Config Files

Due to a feature in NFLib, all network functions support launching from a JSON config file that contains ONVM and DPDK arguments. An example of this can be seen [here](#). In addition, the values specified in the config file can be overwritten by passing them at the command line. The general structure for launching an NF from a config file is ./go.sh -F <CONFIG_FILE.json> <DPDK ARGS> -- <ONVM ARGS> -- <NF ARGS>. Any args specified in <DPDK args> or <ONVM ARGS> will replace the corresponding args in the config file. **An important note:** If no DPDK or ONVM args are passed, but NF args are required, the -- -- is still required. Additionally, launching multiple network functions at once, including circular or linear chains, from a JSON config file is supported. For documentation on developing with config files, see [NF_Dev](#)

7.11.2 NF Starting Scripts

The example NFs can be started using the `start_nf.sh` script. The script can run any example NF based on the first argument which is the NF name (this is based on the assumption that the name matches the NF folder and the build binary). The script has 2 modes:

- Simple

```
1 ./start_nf.sh NF_NAME SERVICE_ID (NF_ARGS)
2 ./start_nf.sh speed_tester 1 -d 1
```

- Complex

```
1 ./start_nf.sh NF_NAME DPDK_ARGS -- ONVM_ARGS -- NF_ARGS
2 ./start_nf.sh speed_tester -l 4 -- -s -r 6 -- -d 5
```

All the NF directories have a symlink to `examples/go.sh` file which allows to omit the NF name argument when running the NF from its directory:

```
1 cd speed_tester && ./go.sh 1 -d 1
2 cd speed_tester && ./go.sh -l 4 -- -s -r 6 -- -d 5
```

7.11.3 Linear NF Chain

In this example, we will be setting up a chain of NFs. The length of the chain is determined by our system's CPU architecture. Some of the commands used in this example are specific to our system; in the cases where we refer to core lists or number of NFs, please run the [Core Helper Script](#) to get your numbers.

1. Determine CPU architecture and running limits:

- Based on information provided by `corehelper` script, use the appropriate core information to run the manager and each NF.

```
1 # scripts/corehelper.py
2 You supplied 0 arguments, running with flag --onvm
3
4 =====
5         openNetVM CPU Corelist Helper
6 =====
7
8 ** MAKE SURE HYPERTHREADING IS DISABLED **
9
10 openNetVM requires at least three cores for the manager:
11 one for NIC RX, one for statistics, and one for NIC TX.
12 For rates beyond 10Gbps it may be necessary to run multiple TX
13 or RX threads.
14
15 Each NF running on openNetVM needs its own core too.
16
17 Use the following information to run openNetVM on this system:
18
19     - openNetVM Manager corelist: 0,1,2
20
21     - openNetVM can handle 7 NFs on this system
```

(continues on next page)

(continued from previous page)

```

22         - NF 1: 3
23         - NF 2: 4
24         - NF 3: 5
25         - NF 4: 6
26         - NF 5: 7
27         - NF 6: 8
28         - NF 7: 9

```

- Running the script on our machine shows that the system can handle 7 NFs efficiently. The manager needs three cores, one for NIC RX, one for statistics, and one for NIC TX.

2. Run Manager:

- Run the manager in dynamic mode with the following command. We are using a corelist here to manually pin the manager to specific cores, a portmask to decide which NIC ports to use, and configuring it display manager statistics to stdout:

```
- # onvm/go.sh 0,1,2 1 0x3F8 -s stdout
```

3. Start NFs:

- First, start at most $n-1$ simple_forward NFs, where n corresponds to the total number of NFs that the system can handle. This is determined from the `scripts/coremask.py` helper script. We will only start two NFs for convenience.
- Simple forward's arguments are core to pin it to, service ID, and destination service ID. The last argument, destination service ID should be $(\text{current_id}) + 1$ if you want to forward it to the next NF in the chain. In this case, we are going to set it to 6, the last NF or basic_monitor.

```
- # examples/start_nf.sh simple_forward 1 -d 6
```

- Second, start a basic_monitor NF as the last NF in the chain:

```
- # examples/start_nf.sh basic_monitor -d 6
```

4. Start a packet generator (i.e. [Pktgen-DPDK](#))

7.11.4 Circular NF Chain

In this example, we can set up a circular chain of NFs. Here, traffic does not leave the openNetVM system, rather we are using the speed_tester NF to generate traffic and send it through a chain of NFs. This example NF can test the speed of the manager's and the NFs' TX threads.

1. Determine CPU architecture and running limits:

- Based on information provided by [Core Helper Script](#), use the appropriate core information to run the manager and each NF.

```

1  # scripts/corehelper.py
2  You supplied 0 arguments, running with flag --onvm
3
4  =====
5  openNetVM CPU Corelist Helper
6  =====
7
8  ** MAKE SURE HYPERTHREADING IS DISABLED **
9

```

(continues on next page)

(continued from previous page)

```

10 openNetVM requires at least three cores for the manager:
11 one for NIC RX, one for statistics, and one for NIC TX.
12 For rates beyond 10Gbps it may be necessary to run multiple TX
13 or RX threads.
14
15 Each NF running on openNetVM needs its own core too.
16
17 Use the following information to run openNetVM on this system:
18
19     - openNetVM Manager corelist: 0,1,2
20
21     - openNetVM can handle 7 NFs on this system
22         - NF 1: 3
23         - NF 2: 4
24         - NF 3: 5
25         - NF 4: 6
26         - NF 5: 7
27         - NF 6: 8
28         - NF 7: 9

```

- Running the script on our machine shows that the system can handle 7 NFs efficiently. The manager needs three cores, one for NIC RX, one for statistics, and one for NIC TX.

2. Run Manager:

- Run the manager in dynamic mode with the following command. We are using a corelist here to manually pin the manager to specific cores, a portmask to decide which NIC ports to use, and configuring it display manager statistics to stdout:

```
- # onvm/go.sh 0,1,2 1 0x3F8 -s stdout
```

3. Start NFs:

- First, start up to n-1 simple_forward NFs. For simplicity, we'll start one simple_forward NF.
 - The NF will have service ID of 2. It also forwards packets to the NF with service ID 1.
 - # ./examples/start_nf.sh simple_forward 2 -d 1
- Second, start up 1 speed_tester NF and have it forward to service ID 2.

```
- # ./examples/start_nf.sh speed_tester 1 -d 2 -c 16000
```

4. We now have a speed_tester sending packets to service ID 2 who then forwards packets back to service ID 1, the speed_tester. This is a circular chain of NFs.

7.12 NF API

OpenNetVM supports 2 modes, by default NFs use the packet_handler callback function that processes packets 1 by 1. The second is the advanced rings mode which gives the NF full control of its rings and allows the developer to do anything they want.

7.12.1 Default Callback mode API init sequence:

The **Bridge NF** is a simple example illustrating this process.

1. First step is initializing the onvm context

```
1 struct onvm_nf_local_ctx *nf_local_ctx;
2 nf_local_ctx = onvm_nflib_init_nf_local_ctx();
```

This configures basic meta data the NF.

2. Start the signal handler

```
1 onvm_nflib_start_signal_handler(nf_local_ctx, NULL);
```

This ensures signals will be caught to correctly shut down the NF (i.e., to notify the manager).

3. Define the function table for the NF

```
1 struct onvm_nf_function_table *nf_function_table;
2 nf_function_table = onvm_nflib_init_nf_function_table();
3 nf_function_table->pkt_handler = &packet_handler;
4 nf_function_table->setup = &nf_setup;
```

The `pkt_handler` call back will be executed on each packet arrival. The `setup` function is called only once after the NF is initialized.

4. Initialize ONVM and adjust the argc, argv based on the return value.

```
1 if ((arg_offset = onvm_nflib_init(argc, argv, NF_TAG, nf_local_ctx, nf_function_table))
    < 0) {
2     onvm_nflib_stop(nf_local_ctx);
3     if (arg_offset == ONVM_SIGNAL_TERMINATION) {
4         printf("Exiting due to user termination\n");
5         return 0;
6     } else {
7         rte_exit(EXIT_FAILURE, "Failed ONVM init\n");
8     }
9 }
10
11 argc -= arg_offset;
12 argv += arg_offset;
```

This initializes DPDK and notifies the Manager that a new NF is starting.

5. Parse NF specific args

```
1 nf_info = nf_context->nf_info;
2 if (parse_app_args(argc, argv, progname) < 0) {
3     onvm_nflib_stop(nf_context);
4     rte_exit(EXIT_FAILURE, "Invalid command-line arguments\n");
5 }
```

6. Run the NF

```
1 onvm_nflib_run(nf_local_ctx);
```

This will cause the NF to enter the run loop, trigger a callback on each new packet.

7. Stop the NF

```
1 onvm_nflib_stop(nf_local_ctx);
```

7.12.2 Advanced rings API init sequence:

The scaling NF provides a clear separation of the two modes and can be found [here](#).

1. First step is initializing the onvm context

```
1 struct onvm_nf_local_ctx *nf_local_ctx;
2 nf_local_ctx = onvm_nflib_init_nf_local_ctx();
```

2. Start the signal handler

```
1 onvm_nflib_start_signal_handler(nf_local_ctx, NULL);
```

3. Contrary to default rings Next we don't need to define the function table

4. Initialize ONVM and adjust the argc, argv based on the return value.

```
1 if ((arg_offset = onvm_nflib_init(argc, argv, NF_TAG, nf_local_ctx, NULL)) < 0) {
2     onvm_nflib_stop(nf_local_ctx);
3     if (arg_offset == ONVM_SIGNAL_TERMINATION) {
4         printf("Exiting due to user termination\n");
5         return 0;
6     } else {
7         rte_exit(EXIT_FAILURE, "Failed ONVM init\n");
8     }
9 }
10
11 argc -= arg_offset;
12 argv += arg_offset;
```

5. Parse NF specific args

```
1 nf_info = nf_context->nf_info;
2 if (parse_app_args(argc, argv, progname) < 0) {
3     onvm_nflib_stop(nf_context);
4     rte_exit(EXIT_FAILURE, "Invalid command-line arguments\n");
5 }
```

6. To start packet processing run this function to let onvm mgr know that NF is running (instead of *onvm_nflib_run* in default mode

```
1 onvm_nflib_nf_ready(nf_context->nf);
```

7. Stop the NF

```
1 onvm_nflib_stop(nf_local_ctx);
```

7.12.3 Optional configuration

If the NF needs additional NF state data it can be put into the data field, this is NF specific and won't be altered by `onvm_nflib` functions. This can be defined after the `onvm_nflib_init` has finished

```
nf_local_ctx->nf->data = (void *)rte_malloc("nf_state_data", sizeof(struct custom_state_  
data), 0);
```

7.13 NF Development

7.13.1 Overview

The openNetVM manager is comprised of two directories: one containing the source code for the `manager` and the second containing the source for the `NF Lib`. The manager is responsible for assigning cores to NFs, maintaining state between NFs, routing packets between NICs and NFs, and displaying log messages and/or statistics. The `NF_Lib` contains useful libraries to initialize and run NFs and libraries to support NF capabilities: `packet helper`, `flow table`, `flow director`, `service chains`, and `message passing`.

Currently, our platform supports at most 128 NF instances running at once with a maximum ID value of 32 for each NF. We currently support a maximum of 32 NF instances per service. These limits are defined in `onvm_common.h`. These are parameters developed for experimentation of the platform, and are subject to change.

NFs are run with different arguments in three different tiers—DPDK configuration flags, openNetVM configuration flags, and NF configuration flags—which are separated with `--`.

- DPDK configuration flags:
 - Flags to configure how DPDK is initialized and run. NFs typically use these arguments:
* `-l CPU_CORE_LIST -n 3 --proc-type=secondary`
- openNetVM configuration flags:
 - Flags to configure how the NF is managed by openNetVM. NFs can configure their service ID and, for debugging, their instance ID (the manager automatically assigns instance IDs, but sometimes it is useful to manually assign them). NFs can also select to share cores with other NFs and enable manual core selection that overrides the `onvm_mgr` core selection (if core is available), their time to live and their packet limit (which is a packet based ttl):
* `-r SERVICE_ID [-n INSTANCE_ID] [-s SHARE_CORE] [-m MANUAL_CORE_SELECTION]
[-t TIME_TO_LIVE] [-l PACKET_LIMIT]`
- NF configuration flags:
 - User defined flags to configure NF parameters. Some of our example NFs use a flag to throttle how often packet info is printed, or to specify a destination NF to send packets to. See the `simple_forward` NF for an example of them both.

Each NF needs to have a packet handler function. It must match this specification: `static int packet_handler(struct rte_mbuf* pkt, struct onvm_pkt_meta* meta);` A pointer to this function will be provided to the manager and is the entry point for packets to be processed by the NF (see NF Library section below). Once packet processing is finished, an NF can set an *action* coupled with a *destination NF ID* or *destination port ID* in the packet which tell the openNetVM manager how route the packet next. These *actions* are defined in `onvm_common`:

- `ONVM_NF_ACTION_DROP`: Drop the packet
- `ONVM_NF_ACTION_NEXT`: Forward the packet using the rule from the SDN controller stored in the flow table
- `ONVM_NF_ACTION_TONF`: Forward the packet to the specified NF

- `ONVM_NF_ACTION_OUT`: Forward the packet to the specified NIC port

7.13.2 Skeleton NF

In order to provide a baseline implementation for the development of NFs, a skeleton is provided in `openNetVM/examples/skeleton/`. This skeleton outlines all required files and functions within any standard ONVM NF, and their respective uses. These functions, which are declared in the function table or control execution, include:

- **main**: `int main(int argc, char *argv[])` Responsible for initializing the function table and local NF context. Starts and stops the NF, and handles command-line arguments.
- **setup** `static void setup(struct onvm_nf_local_ctx *nf_local_ctx)` Allows the NF to initialize non-local data or perform necessary instructions which must be executed before receiving packets or messages.
- **action** `static int action(struct onvm_nf_local_ctx *nf_local_ctx)` Continually called while the NF is running, allowing the user to perform actions or checks which are packet or message-independent.
- **packet_handler** `static int packet_handler(struct rte_mbuf *pkt, struct onvm_pkt_meta *meta, struct onvm_nf_local_ctx *nf_local_ctx)` As described previously, the handler function will handle each packet upon arrival.
- **message_handler** `static void handle_msg(void *msg_data, struct onvm_nf_local_ctx *nf_local_ctx)` Like the packet handler, the message handler function will handle messages from other NFs upon arrival.

For those who wish to use the skeleton as a training mechanism, inline commentary will provide additional clarification on the purpose of each function and where edits must be made throughout the files. The NF also includes various customary helper-functions for those seeking a baseline template; these functions can be removed if not applicable.

To demonstrate the management of data, the NF contains high-level functionality that counts the number of packets received and the time which the NF has been running. This data is stored within a state struct initialized in main and stored within the local context. The NF will print the current time and number of packets on a delay which is specified by the user at the command line (see `/examples/skeleton/README.md` for further details). To observe packets being counted, you can easily send packets from the `speed_tester` NF by following the instructions on the [:ref:`OpenNetVM Examples Page <examples>`](#) (you are looking to create a linear NF chain, but the circular NF chain instructions will provide context on how to use the `speed_tester` PCAP files). If you are looking for introductory comprehension, tracing through the skeleton NF is a logical place to begin. For developing more advanced NFs, template usage requires only a few deletions.

7.13.3 NF Library

The `NF_Lib` Library provides functions to allow each NF to interface with the manager and other NFs. This library provides the main communication protocol of the system. To include it, add the line `#include "onvm_nflib.h"` to the top of your c file.

Here are some of the frequently used functions of this library (to see the full API, please review the [NF_Lib header](#)):

- `int onvm_nflib_init(int argc, char *argv[], const char *nf_tag, struct onvm_nf_info** nf_info_p)`, initializes all the data structures and memory regions that the NF needs run and communicates with the manager about its existence. Fills the passed double pointer with the `onvm_nf_info` struct. This is required to be called in the main function of an NF.
- `int onvm_nflib_run(struct onvm_nf_info* info, void(*handler)(struct rte_mbuf* pkt, struct onvm_pkt_meta* meta))`, is the communication protocol between NF and manager, where the NF provides a pointer to a packet handler function to the manager. The manager uses this function pointer to pass packets to the NF as it is routing traffic. This function continuously loops, giving packets one-by-one to the destined NF as they arrive.

Advanced Ring Manipulation

For advanced NFs, calling `onvm_nf_run` (as described above) is actually optional. There is a second mode where NFs can interface directly with the shared data structures. Be warned that using this interface means the NF is responsible for its own packets, and the NF Guest Library can make fewer guarantees about overall system performance. The advanced rings NFs are also responsible for managing their own cores, the NF can call the `onvm_threading_core_affinitize(nf_info->core)` function, the `nf_info->core` will have the core assigned by the manager. Additionally, the NF is responsible for maintaining its own statistics. An advanced NF can call `onvm_nflib_get_nf(uint16_t id)` to get the reference to `struct onvm_nf`, which has `struct rte_ring *` for RX and TX, a stat structure for that NF, and the `struct onvm_nf_info`. Alternatively NF can call `onvm_nflib_get_rx_ring(struct onvm_nf_info *info)` or `onvm_nflib_get_tx_ring(struct onvm_nf_info *info)` to get the `struct rte_ring *` for RX and TX, respectively. Finally, note that using any of these functions precludes you from calling `onvm_nf_run`, and calling `onvm_nf_run` precludes you from calling any of these advanced functions (they will return NULL). The first interface you use is the one you get. To start receiving packets, you must first signal to the manager that the NF is ready by calling `onvm_nflib_nf_ready`. Example use of Advanced Rings can be seen in the `speed_tester` NF or the `scaling` example NF.

Multithreaded NFs, scaling

NFs can scale by running multiple threads. For launching more threads the main NF had to be launched with more than 1 core. For running a new thread the NF should call `onvm_nflib_scale(struct onvm_nf_scale_info *scale_info)`. The `struct scale_info` has all the required information for starting a new child NF, service and instance ids, NF state data, and the packet handling functions. The struct can be obtained either by calling the `onvm_nflib_get_empty_scaling_config(struct onvm_nf_info *parent_info)` and manually filling it in or by inheriting the parent behavior by using `onvm_nflib_inherit_parent_config(struct onvm_nf_info *parent_info)`. As the spawned NFs are threads they will share all the global variables with its parent, the `onvm_nf_info->data` is a void pointer that should be used for NF state data. Example use of Multithreading NF scaling functionality can be seen in the `scaling_example` NF.

Shared core mode

This is an **EXPERIMENTAL** mode for OpenNetVM. It allows multiple NFs to run on a shared core. In “normal” OpenNetVM, each NF will poll its RX queue and message queue for packets and messages respectively, monopolizing the CPU even if it has a low load. This branch adds a semaphore-based communication system so that NFs will block when there are no packets and messages available. The NF Manger will then signal the semaphore once one or more packets or messages arrive.

This code allows you to evaluate resource management techniques for NFs that share cores, however it has not been fully tested with complex NFs, therefore if you encounter any bugs please create an issue or a pull request with a proposed fix.

The code is based on the hybrid-polling model proposed in [Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization](#) by Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, and Timothy Wood, published at Co-NEXT 16 and extended in [NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains](#) by Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai and Xiaoming Fu, published at SIGCOMM '17. Note that this code does not contain the full Flurries or NFVnice systems, only the basic support for shared-Core NFs. However, we have recently released a full version of the NFVNice system as an experimental branch, which can be found [here](#).

Usage / Known Limitations:

- To enable pass a `-c` flag to the `onvm_mgr`, and use a `-s` flag when starting a NF to specify that they want to share cores
- All code for sharing CPUs is within `if (ONVM_NF_SHARE_CORES)` blocks

- When enabled, you can run multiple NFs on the same CPU core with much less interference than if they are polling for packets and messages
- This code does not provide any particular intelligence for how NFs are scheduled or when they wakeup/sleep
- Note that the manager threads all still use polling

7.13.4 Packet Helper Library

The openNetVM Packet Helper Library provides an abstraction to support development of NFs that use complex packet processing logic. Here is a selected list of capabilities that it can provide:

- Swap the source and destination MAC addresses of a packet, then return 0 on success. `onvm_pkt_mac_addr_swap` can be found [here](#)
- Check the packet type, either TCP, UDP, or IP. If the packet type is verified, these functions will return 1. They can be found [here](#)
- Extract TCP, UDP, IP, or Ethernet headers from packets. These functions return pointers to the respective headers in the packets. If provided an unsupported packet header, a NULL pointer will be returned. These are found [here](#)
- Print the whole packet or individual headers of the packet. These functions can be found [here](#).

7.13.5 Config File Library

The openNetVM Config File Library provides an abstraction that allows NFs to load values from a JSON config file. While NFLib automatically loads all DPDK and ONVM arguments when `-F` is passed, a developer can add config support directly within the NF to support passing additional values.

- NOTE: unless otherwise specified, all DPDK and ONVM arguments are **required**
- `onvm_config_parse_file(const char* filename)`: Load a JSON config file, and return a pointer to the cJSON struct.
- This is utilized to launch NFs using values specified in a config file.
- `onvm_config_parse_file` can be found [here](#)
- Additional config options can be loaded from within the NF, using cJSON. For further reference on how to access the values from the cJSON object, see the [cJSON docs](#)

Sample Config File

```

1  {
2      "dpdk": {
3          "corelist": [STRING: corelist],
4          "memory_channels": [INT: number of memory channels],
5          "portmask": [INT: portmask]
6      },
7
8      "onvm": {
9          "output": [STRING: output loc, either stdout or web],
10         "serviceid": [INT: service ID for NF],
11         "instanceid": [OPTIONAL, INT: this optional arg sets the instance ID of the NF]
12     }
13 }
```

7.13.6 Running Groups of NFs

Additionally, a developer can run the [run group script](#) to deploy multiple network functions, including linear or circular chains of multiple NFs, from a JSON config file. An example config file can be found [here](#). “NF Name” indicates the example NF to be run and must be the name of the NF folder in the [examples folder](#).

Optional “globals” in the config file include:

- “TTL”: specifies number of seconds for the NFs to run before shutdown. If no timeout is specified, the NFs will run until a raised error or a manual shutdown (Ctrl + C).
- “directory”: specifies a directory name. A directory will be created (if it does not already exist) for the output log files.
- “directory-prefix”: a directory will be created with the prefix + timestamp. If no directory name or directory-prefix is specified, the default name of the created directory will be the timestamp. Output from each NF will be continuously written to the corresponding log text file within the created or pre-existing directory. Format of the log file name will be: “log-NF name-instance ID”.

To track the output of a NF:

This script must be run within the /examples folder:

JSON Config File For Deploying Multiple NFs

7.14 Contribution Guidelines

To contribute to OpenNetVM, please follow these steps:

1. Please read our [style guide](#).
2. Create your own fork of the OpenNetVM repository
3. Add our master repository as an upstream remote:

```
git remote add upstream https://github.com/sdnfv/openNetVM
```

4. Update the develop branch before starting your work:

```
git pull upstream develop
```

5. Create a branch off of develop for your feature.

We follow the fork/branch workflow where no commits are ever made to `develop` or `master`. Instead, all development occurs on a separate feature branch. Please read [this guide](#) on the Git workflow.

6. When contributing to documentation for ONVM, please see this [Restructured Text \(reST\) guide](#) for formatting.
7. Add your commits

Good commit messages contain both a subject and body. The subject provides an overview whereas the body answers the *what* and *why*. The body is usually followed by a change list that explains the *how*, and the commit ends with a *test plan* describing how the developer verified their change. Please read [this guide](#) for more information.

8. When you’re ready to submit a pull request, rebase against `develop` and clean up any merge conflicts

```
git pull --rebase upstream develop
```

9. Please fill out the pull request template as best as possible and be very detailed/thorough.

7.15 Release Notes

7.15.1 About

Release Cycle

We track active development through the `devel`op branch and verified stable releases through the `master` branch. New releases are created and tagged with the year and month of their release. When we tag a release, we update `master` to have the latest stable code.

Versioning

As of 11/06/2017, we are retiring semantic versioning and will instead use a date based versioning system. Now, a release version can look like 17.11 where the “major” number is the year and the “minor” number is the month.

7.15.2 v21.10 (10/2021): Bug Fixes, Test cases, Dev Environment Improvements

This release focused on general bug fixing and improving our test/development environments. A CloudLab template will be available with the latest release here: <https://www.cloudlab.us/p/GWCloudLab/onvm>

New Features and NFs

- [243] Adds L3 Switch example based on DPDK `l3fwd` sample code. This NF can forward packets either using longest prefix match or a hash table lookup.
- [254] Adds Fair Queue NF that demonstrates how to use advanced rings mode to directly access packets and distribute them to a set of child NFs. Packets are “classified” using a CRC32 hash and assigned to a queue. Queues are then read from in Round Robin order to process packets of different types in a fair way. Contributed by (Rohit M P) from NITK.
- [277] Adds support for Jumbo frame packets. Enable by adding a `-j` flag to the manager’s `go.sh` script.

Testing and Development Improvements

- [296] Adds unit test for NF messaging infrastructure and fixes memory leak related to overflow of message pools [Issue 293].
- [297] Adds VS Code profile to simplify debugging of NFs.
- [302] Adds NF chain performance test to measure and plot inter-NF throughput and latency.
- [308] Adds socket ID information to NF and manager logging print statements.

Miscellaneous Bug and Documentation Fixes

- [304] Fixes the NF_TAG of aes_decrypt in openNetVM/examples/aes_decrypt/aesdecrypt.c.
- [300] Updates MoonGen installation document to work with the new DPDK version.
- [270] Fixes issues with relative path in the onvm go script to find the web directory. Now using \$ONVM_HOME instead of ...
- [272] Fixes two bugs (including Issue 233) where the NF rings would not be cleared after deallocation and an underflow bug in stats.
- [265] Updates Install README to provide further clarification as well as to include a missing package.
- [267] Fixes typos in onvm_pkt_helper.h.
- [317] Fixes the ARP NF endianness for source IP addresses.
- [306] Updates linter installation script to use newer versions of cppcheck and Ubuntu.
- [316] Fixes Speed Tester NF so that it does not crash while loading a PCAP trace with jumbo frames without the correct flags.

Contributors:

- Dennis Afanasev (dennisafa)
- Noah Chinitz (NoahChinitzGWU)
- Benjamin De Vierno (bdevierno1)
- Kevin Deems (kevindweb)
- Lauren Hahn (lhahn01)
- Elliott (Elie) Henne (elliottenne)
- Vivek Jain (vivek-anand-jain)
- Jack Kuo (JackKuo-tw)
- Catherine Meadows (catherinemeadows)
- Rohit M P (rohit-mp)
- Leslie Monis (lesliemonis)
- Peng Wu (PengWu-wp)

7.15.3 v20.10 (10/2020): OS/Dependency Updates, Bug Fixes, New NFs

A CloudLab template will be available with the latest release here: <https://www.cloudlab.us/p/GWCloudLab/onvm>

Version Updates

- [249] Updates ONVM to use DPDK 20.05 and Pktgen 20.05, as well as to run on Ubuntu 20.04 LTS

Miscellaneous Bug and Documentation Fixes

- [216] Related to issues [#200] and [#201]. Removed a few memory leaks. Replaced some libc function calls with the corresponding recommended dpdk function calls. Created new function to free allocated memory upon manager termination. Added if statements to error check after every allocation.
- [221] Updates Python scripts to use Python 3 rather than Python 2 and adds Python 3 as a dependency in the ONVM Install Guide
- [224] Use NF data instead of global variables in simple_fwd_tb NF
- [223] Fixes a bug in the load_generator NF. “Failure to obtain MAC address” error caused the NF to hang, so a function is now called to obtain a fake MAC address in this case.
- [227] Update stats updating during ONVM_NF_ACTION_OUT action
- [229] Fixed Issue [#228] by properly counting the number of ports in the port mask and comparing it to the number available
- [235] Changed Pull Request template to remind users to request onto develop instead of master
- [239] Updates existing Pktgen installation guide by providing clarity on Lua installation and improves ONVM’s Pktgen Wiki page
- [241] Updated README.md with description of config file
- [252] Improves ONVM Install Guide to reflect DPDK and Pktgen version updates
- [258] Updated README.md with description of config file. This relates to PR [#241], since there are some linter issues with 241, this PR is made to resolve that issue.

New Features and NFs

- [218] Python script that parses a JSON config file to launch multiple NFs
- [219] Manager has new suggested syntax: `./go.sh -k <port mask> -n <NF core mask> [OPTIONS]`. This replaces the positional argument-based syntax with flags. Also assumes default CPU cores unless the user uses the `-m` flag to specify manager cores
- [230] Add L2 switch NF. The functionality of this NF is similar to the DPDK l2fwd example

Contributors

- Dennis Afanasev ([dennisafa](#))
- Ethan Baron ([EthanBaron14](#))
- Benjamin De Vierno ([bdevierno1](#))
- Kevin Deems ([kevindweb](#))
- Mingyu Ma ([WilliamMaa](#))
- Catherine Meadows ([catherinemeadows](#))
- Sreya Nalla ([sreyanalla](#))

- Rohit M P ([rohit-mp](#))
- [khaledshahine](#)
- Vivek Jain ([vivek-anand-jain](#))

7.15.4 v20.05 (May 31, 2020): Bug Fixes, Usability Improvements, and Token Bucket NF

A CloudLab template will be available with the latest release here: <https://www.cloudlab.us/p/GWCloudLab/onvm>

Miscellaneous Bug and Documentation Fixes

- [158] Print a warning message when the user specifies a specific core (-1) for NF to start on but doesn't specify a -m flag to use that core for NF to run on. To force an NF to run on a specific core, the -m flag must be used, otherwise the Manager will assign the NF to a free core.
- [159] In `onvm_ft_create` instead of calling `rte_hash_create` from secondary process (NF), enqueue a message for the primary process (`onvm_mgr`) to do it and then return a pointer.
- [160] Fixes the case when packets that had an invalid out port would crash the manager. This is done by maintaining a port init array.
- [166], [209] Updates dependencies mentioned in 194 by updating Acorn's version from 5.7.3 to 5.7.4 for ONVM Web
- [173] Fixes a bug in `load_generator` that caused a seg fault, the NF wasn't calling the setup function before running.
- [180] Prevent user from running multiple managers or starting an NF prior to the manager.
- [183] Improved style and efficiency of bash shell scripts using ShellCheck linter
- [189] Fixes broken links in Moongen and Pktgen installation guides
- [197] Fixes error where manager cannot start because base address for shared memory region is already in use which would cause `Cannot mmap memory for rte_config` error.
- [202] Allows Dockers to run on Ubuntu 18.04 successfully. Bug fixes allow NFs to be run both within or outside a container where checks for a running manager and manager duplication are only done when an NF is running outside a container.
- [204] UDP source port and destination port packet headers are now converted from Big Endian to CPU order, using built-in DPDK conversion method, resulting in correct packet information printing.

New Features and NFs

- [178] Dynamically allocates memory for NF data structures when NFs start instead of statically at program initialization. Maximum number of NFs is still limited `MAX_NFS` in `onvm_common.h` (default is 128).
- [179] NFs print summary statistics when exiting
- [196] Continuous Integration improvements
 - Created a Github Action to run linter on incoming PRs. Also checks if the PR was submitted to the `develop` branch.
 - Added three static analysis tools.
 - * Pylint

- * Cppcheck
- * Shellcheck
- New scripts for researchers to install necessary dependencies and run linter locally.
- Removed CI code from the main repository
- [199] Added new Simple Forward Token Bucket Rate Limiter NF that simulates a queue with a token bucket and forwards packets to a specific destination. The NF forwards packets based on a user specified rate (-R) and depth (-D).

Contributors:

- Dennis Afanasev ([dennisafa](#))
- Ethan Baron ([EthanBaron14](#))
- Benjamin De Vierno ([bdevierno1](#))
- Kevin Deems ([kevindweb](#))
- Mingyu Ma ([WilliamMaa](#))
- Catherine Meadows ([catherinemeadows](#))
- Sreya Nalla ([sreya519](#))
- Rohit M P ([rohit-mp](#))

7.15.5 v19.07 (7/19): NFD library and example NFs, Continuous Integration updates, minor improvements and bug fixes.

A CloudLab template is available with the latest release here: <https://www.cloudlab.us/p/GWCloudLab/onvm>

Performance: This release includes a new macro `ENABLE_FLOW_LOOKUP` which controls whether a flow lookup is performed for every incoming packet. If disabled, all packets are forwarded to the default service ID which improves performance. The flow lookup is still enabled by default for backward compatibility with other applications that use ONVM.

NFD library with example NFS

Add example NFs based on NFD, a C++-based NF developing compiler designed by Wenfei Wu's group (<http://wenfei-wu.github.io/>) from IIIS, Tsinghua University, China. NFD compiles the NF logic into a common C++ program by using table-form language to model NFs' behavior.

The NFD compiler itself isn't included, only the NFs that were created with it.

A list of provided NFs using NFD library: - DNS Amplification Mitigation - Super Spread Detection - Heavy Hitter Detection - SYN Flood Detection - UDP Flood Detection - Stateless Firewall - Stateful Firewall - NAPT

Continuous Integration updates:

CI got a few major updates this release: - CI will do basic lint checks and branch checks(all PRs should be submitted against the *develop* branch) for unauthorized users - If CI is working on a request and receives another request it will append it to the queue instead of dropping it - CI will now run Pktgen as an additional test metric.

Minor Improvements

Shared core functionality for messages - Adds functionality for NFs using shared core mode to work with NF messages. This means the NF will now sleep when no messages and no packets are enqueued onto a NF's message ring and wakeup if either one is received.

NF core rebalancing - Adds functionality for `onvm_mgr` to remap a NF to a different core, if such occurs (when another NF shuts down). This is disabled by default and can be enabled using the `ONVM_NF_SHUTDOWN_CORE_REASSIGNMENT` macro.

Bug fixes:

- Fix Style guide links
- Fix Typo in Stats Header bug fix
- Fix Stats Header in Release Notes (twice)

7.15.6 v19.05 (5/19): Shared Core Mode, Major Architectural Changes, Advanced Rings Changes, Stats Updates, CI PR Review, LPM Firewall NF, Payload Search NF, TTL Flags, minor improvements and bug fixes.

A CloudLab template is available with the latest release here: <https://www.cloudlab.us/p/GWCloudLab/onvm>

This release features a lot of breaking API changes.

Performance: This release increases Pktgen benchmark performance from 7Mpps to 13.1 Mpps (measured by Pktgen sending packets to the ONVM Basic Monitor), thus fixing the major performance issue that was present in the last release.

Repo changes: Default branch has been changed to `master`, active development can still be seen in `develop`. Most of the development is now done on the public repo to improve visibility, planned projects and improvements can be seen in this [pinned issue](#), additionally pull requests and issues are now cataloged by tags. We're also starting to merge releases into master by pull requests, thus developers should branch off the develop branch and submit PRs against the develop branch.

Note: If the NFs crash with this error - Cannot mmap memory for `rte_config` at `[0x7ffff7ff3000]`, got `[0x7ffff7ff2000]`, simply use the `-a 0x7f0000000000` flag for the `onvm_mgr`, this will resolve the issue.

Shared Core Mode:

This code introduces **EXPERIMENTAL** support to allow NFs to efficiently run on **shared** CPU cores. NFs wait on semaphores when idle and are signaled by the manager when new packets arrive. Once the NF is in wake state, no additional notifications will be sent until it goes back to sleep. Shared core variables for mgr are in the `nf_wakeup_info` structs, the NF shared core vars were moved to the `onvm_nf` struct.

The code is based on the hybrid-polling model proposed in *Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization* by Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, and Timothy Wood, published at Co-NEXT 16 and extended in *NFVnice: Dynamic Backpressure and Scheduling for NFV Service Chains* by Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumathurai and Xiaoming Fu, published at SIGCOMM '17. Note that this code does not contain the full Flurries or NFVnice systems, only the basic support for shared-Core NFs. However, we have recently released a full version of the NFVNice system as an experimental branch, which can be found [here](#).

Usage and implementation details can be found [here](#).

Major Architectural Changes:

- Introduce a local `onvm_nf_init_ctx` struct allocated from the heap before starting onvm.
 - Previously the initialization sequence for NFs wasn't able to properly cleanup if a signal was received. Because of this we have introduced a new NF context struct (`onvm_nf_local_ctx`) which would be malloced before initialization begins and would help handle cleanup. This struct contains relevant information about the status of the initialization sequence and holds a reference to the `onvm_nf` struct which has all the information about the NF.
- Reworking the `onvm_nf` struct.
 - Previously the `onvm_nf` struct contained a pointer to the `onvm_nf_info`, which was used during processing. It's better to have one main struct that represents the NF, thus the contents of the `onvm_nf_info` were merged into the `onvm_nf` struct. This allows us to maintain a cleaner API where all information about the NF is stored in the `onvm_nf` struct.
- Replace the old `onvm_nf_info` with a new `onvm_nf_init_ctx` struct that is passed to `onvm_mgr` for initialization.
 - This struct contains all relevant information to spawn a new NF (service/instance IDs, flags, core, etc). When the NF is spawned this struct will be released back to the mempool.
- Adding a function table struct `onvm_nf_function_table`.
 - Finally, we introduced the `onvm_nf_function_table` struct that groups all NF callback functions that can be set by developers.

Overall, the new NF launch/shutdown sequence looks as follows:

```

1 struct onvm_nf_local_ctx *nf_local_ctx;
2 struct onvm_nf_function_table *nf_function_table;
3
4 nf_local_ctx = onvm_nflib_init_nf_local_ctx();
5 onvm_nflib_start_signal_handler(nf_local_ctx, NULL);
6
7 nf_function_table = onvm_nflib_init_nf_function_table();
8 nf_function_table->pkt_handler = &packet_handler;
9
10 if ((arg_offset = onvm_nflib_init(argc, argv, NF_TAG, nf_local_ctx, nf_function_table))
    ↪ < 0)
```

(continues on next page)

(continued from previous page)

```

11      // error checks
12
13  argc -= arg_offset;
14  argv += arg_offset;
15
16  if (parse_app_args(argc, argv, progname) < 0)
17      // error checks
18
19  onvm_nflib_run(nf_local_ctx);
20  onvm_nflib_stop(nf_local_ctx);

```

Advanced Rings Changes:

This release changes our approach to NFs using the advanced rings mode. Previously we were trying to provide APIs for advanced ring developers such as scaling, but this logic should be managed by the NFs themselves. Because of this we're reworking those APIs and letting the NF devs handle everything themselves.

- Speed Tester NF advanced rings mode is removed
- Extra APIs have been removed
- Removes support for advanced rings scaling APIs
- Scaling Example NF advanced rings mode has been reworked, the new implementation now does its own pthread creation instead of relying on the onvm scaling APIs. Also makes a clear separation between default and advanced ring mode.
- Because of these changes some internal nflib APIs were exposed to the NF (onvm_nflib_start_nf, onvm_nflib_init_nf_init_cfg, onvm_nflib_inherit_parent_init_cfg)

Stats Updates:

This release updates both console and web stats.

- For web stats this adds the Core Mappings page with the core layout for both onvm_mgr and NFs.
- For console stats this overhauls the displayed stats and adds new information, see more below.

The new default mode now displays NF tag and core ID:

```

1  PORTS
2  -----
3  Port 0: '90:e2:ba:b3:bc:6c'
4
5  Port 0 - rx:          4 (          0 pps) tx:          0 (          0 pps)
6
7  NF TAG      IID / SID / CORE  rx_pps / tx_pps      rx_drop / tx_drop
8  ↪ out / tonf / drop
9  -----
10 ↪ -----
11 speed_tester  1 / 1 / 4      1693920 / 1693920      0 / 0
12 ↪          0 / 40346970 / 0

```

Verbose mode also adds PNT (Parent ID), S|W (NF state, sleeping or working), CHLD (Children count):

```

1  PORTS
2  -----
3  Port 0: '90:e2:ba:b3:bc:6c'
4
5  Port 0 - rx:          4 (          0 pps) tx:          0 (          0 pps)

```

(continues on next page)

(continued from previous page)

```

6
7 NF TAG      IID / SID / CORE  rx_pps / tx_pps      rx / tx
  ↳ out      /  tonf      / drop
8          PNT / S|W / CHLD drop_pps / drop_pps      rx_drop / tx_drop
  ↳next      /  buf      / ret
9 -----
  ↳ -----
10 speed_tester 1 / 1 / 4      9661664 / 9661664      94494528 / 94494528
  ↳          0 / 94494487      / 0
11          0 / W / 0          0 / 0          0 / 0
  ↳          0 / 0          / 128

```

The shared core mode adds wakeup information stats:

```

1 PORTS
2 -----
3 Port 0: '90:e2:ba:b3:bc:6c'
4
5 Port 0 - rx:      5 (      0 pps) tx:      0 (      0 pps)
6
7 NF TAG      IID / SID / CORE  rx_pps / tx_pps      rx / tx
  ↳ out      /  tonf      / drop
8          PNT / S|W / CHLD drop_pps / drop_pps      rx_drop / tx_drop
  ↳next      /  buf      / ret
9                      wakeups / wakeup_rt
10 -----
  ↳ -----
11 simple_forward 2 / 2 / 4      27719 / 27719      764439 / 764439
  ↳          0 / 764439      / 0
12          0 / S / 0          0 / 0          0 / 0
  ↳          0 / 0          / 0
13                      730557 / 25344
14
15 speed_tester   3 / 1 / 5      27719 / 27719      764440 / 764439
  ↳          0 / 764440      / 0
16          0 / W / 0          0 / 0          0 / 0
  ↳          0 / 0          / 1
17                      730560 / 25347
18
19 Shared core stats
20 -----
21 Total wakeups = 1461122, Wakeup rate = 50696

```

The super verbose stats mode has also been updated to include new stats:

```

1 #YYYY-MM-DD HH:MM:SS,nic_rx_pkts,nic_rx_pps,nic_tx_pkts,nic_tx_pps
2 #YYYY-MM-DD HH:MM:SS,nf_tag,instance_id,service_id,core,parent,state,children_cnt,rx,tx,
  ↳rx_pps,tx_pps,rx_drop,tx_drop,rx_drop_rate,tx_drop_rate,act_out,act_tonf,act_drop,act_
  ↳next,act_buffer,act_returned,num_wakeups,wakeup_rate
3 2019-06-04 08:54:52,0,4,4,0,0
4 2019-06-04 08:54:53,0,4,0,0,0
5 2019-06-04 08:54:54,simple_forward,1,2,4,0,W,0,29058,29058,29058,29058,0,0,0,0,29058,0,

```

(continues on next page)

(continued from previous page)

```
↪0,0,0,28951,28951
6 2019-06-04 08:54:54,speed_tester,2,1,5,0,S,0,29058,29058,29058,29058,0,0,0,0,29059,0,0,
↪0,1,28952,28952
7 2019-06-04 08:54:55,0,4,0,0,0
8 2019-06-04 08:54:55,simple_forward,1,2,4,0,W,0,101844,101843,72785,72785,0,0,0,0,0,
↪101843,0,0,0,0,101660,101660
9 2019-06-04 08:54:55,speed_tester,2,1,5,0,W,0,101844,101843,72785,72785,0,0,0,0,101844,
↪0,0,0,1,101660,101660
```

CI PR Review:

CI is now available on the public branch. Only a specific list of whitelisted users can currently run CI for security purposes. The new CI system is able to approve/reject pull requests. CI currently performs these checks: - Check the branch (for our discussed change of develop->master as main branch) - Run performance check (speed tester currently with 35mil benchmark) - Run linter (only on the PR diff)

LPM Firewall NF:

The firewall NF drops or forwards packets based on rules provided in a JSON config file. This is achieved using DPDK's LPM (longest prefix matching) library. Default behavior is to drop a packet unless the packet matches a rule. The NF also has a debug mode to print decisions for every packet and an inverse match mode where default behavior is to forward a packet if it is not found in the table. Documentation for this NF can be found [here](#).

Payload Search NF:

The Payload Scan NF provides the functionality to search for a string within a given UDP or TCP packet payload. Packet is forwarded to its destination NF on a match, dropped otherwise. The NF also has an inverse mode to drop on match and forward otherwise. Documentation for this NF can be found [here](#).

TTL Flags:

Adds TTL and packet limit flags to stop the NF or the onvm_mgr based on time since startup or based on packets received. Default measurements for these flags are in seconds and in millions of packets received.

NF to NF Messaging:

Adds the ability for NFs to send messages to other NFs. NFs need to define a message handler to receive messages and are responsible to free the custom message data. If the message is sent to a NF that doesn't have a message handler the message is ignored.

Minor Improvements

- **Make Number of mbufs a Constant Value** - Previously the number of mbufs was calculated based on the `MAX_NFS` constant. This led to performance degradation as the requested number of mbufs was too high, changing this to a constant has significantly improved performance.
- **Reuse NF Instance IDs** - Reuse instance IDs of old NFs that have terminated. The instance IDs are still continuously incremented up to the `MAX_NFS` constant, but when that number is reached the next NF instance ID will be wrapped back to the starting value and find the first unoccupied instance ID.
- Fix all major style errors
- Check if `ONVM_HOME` is Set Before Compiling ONVM
- Add Core Information to Web Stats
- Update Install Script Hugepage Setup & Kernel Driver Installation
- Add Compatibility Changes to Run ONVM on Ubuntu 18.04.1
- Various Documentation updates and fixes
- Change `onvm-pktgen` Submodule to Upstream Pktgen

Bug fixes:

- Free Memory on `ONVM_MGR` Shutdown
- Launch Script to Handle Multi-word String Arguments
- NF Advanced Ring Thread Process NF Shutdown Messages
- Adds NF Ring Cleanup Logic On Shutdown
- Resolve Shutdown Memory Leaks
- Add NF Tag Memory Allocation
- Fix the Parse IP Helper Function
- Fix Speed Tester NF Generated Packets Counter
- Add Termination of Started but not yet Running NFs
- Add ONVM mgr web mode memory cleanup on shutdown
- Removes the Old Flow Tracker NF Launch Script
- Fix Deprecated DPDK Function in Speed Tester NF

v19.05 API Struct changes:

- Adding `onvm_nf_local_ctx` which is malloced and passed into `onvm_nflib_init`:

```

1 struct onvm_nf_local_ctx {
2     struct onvm_nf *nf;
3     rte_atomic16_t nf_init_finished;
4     rte_atomic16_t keep_running;
5 };

```

- Adding a function table for eaiser callback managing:

```

1 struct onvm_nf_function_table {
2     nf_setup_fn  setup;
3     nf_msg_handler_fn  msg_handler;
4     nf_user_actions_fn  user_actions;
5     nf_pkt_handler_fn  pkt_handler;
6 };

```

- Renaming the old onvm_nf_info -> onvm_nf_init_cfg:

```

1 struct onvm_nf_init_cfg {
2     uint16_t instance_id;
3     uint16_t service_id;
4     uint16_t core;
5     uint16_t init_options;
6     uint8_t status;
7     char *tag;
8     /* If set NF will stop after time reaches time_to_live */
9     uint16_t time_to_live;
10    /* If set NF will stop after pkts TX reach pkt_limit */
11    uint16_t pkt_limit;
12 };

```

- Consolidating previous onvm_nf_info and onvm_nf into a singular onvm_nf struct:

```

1 struct onvm_nf {
2     struct rte_ring *rx_q;
3     struct rte_ring *tx_q;
4     struct rte_ring *msg_q;
5     /* Struct for NF to NF communication (NF tx) */
6     struct queue_mgr *nf_tx_mgr;
7     uint16_t instance_id;
8     uint16_t service_id;
9     uint8_t status;
10    char *tag;
11    /* Pointer to NF defined state data */
12    void *data;
13
14    struct {
15        uint16_t core;
16        /* Instance ID of parent NF or 0 */
17        uint16_t parent;
18        rte_atomic16_t children_cnt;
19    } thread_info;
20
21    struct {
22        uint16_t init_options;
23        /* If set NF will stop after time reaches time_to_live */
24        uint16_t time_to_live;
25        /* If set NF will stop after pkts TX reach pkt_limit */
26        uint16_t pkt_limit;
27    } flags;
28
29    /* NF specific functions */

```

(continues on next page)

(continued from previous page)

```

30  struct onvm_nf_function_table *function_table;
31
32  /*
33   * Define a structure with stats from the NFs.
34   *
35   * These stats hold how many packets the NF will actually receive, send,
36   * and how many packets were dropped because the NF's queue was full.
37   * The port-info stats, in contrast, record how many packets were received
38   * or transmitted on an actual NIC port.
39   */
40  struct {
41      volatile uint64_t rx;
42      volatile uint64_t rx_drop;
43      volatile uint64_t tx;
44      volatile uint64_t tx_drop;
45      volatile uint64_t tx_buffer;
46      volatile uint64_t tx_returned;
47      volatile uint64_t act_out;
48      volatile uint64_t act_tonf;
49      volatile uint64_t act_drop;
50      volatile uint64_t act_next;
51      volatile uint64_t act_buffer;
52  } stats;
53
54  struct {
55      /*
56      ↪ wakeups
57       *      sleep_state = 1 => NF sleeping (waiting on semaphore)
58       *      sleep_state = 0 => NF running (not waiting on semaphore)
59       */
60      rte_atomic16_t *sleep_state;
61      /* Mutex for NF sem_wait */
62      sem_t *nf_mutex;
63  } shared_core;
64  };

```

v19.05 API Changes:

- `int onvm_nflib_init(int argc, char *argv[], const char *nf_tag, struct onvm_nf_info **nf_info_p) -> int onvm_nflib_init(int argc, char *argv[], const char *nf_tag, struct onvm_nf_local_ctx *nf_local_ctx, struct onvm_nf_function_table *nf_function_table)`
- `int onvm_nflib_run(struct onvm_nf_info* info, pkt_handler_func pkt_handler) -> int onvm_nflib_run(struct onvm_nf_local_ctx *nf_local_ctx)`
- `int onvm_nflib_return_pkt(struct onvm_nf_info *nf_info, struct rte_mbuf* pkt) -> int onvm_nflib_return_pkt(struct onvm_nf *nf, struct rte_mbuf *pkt)`
- `int onvm_nflib_return_pkt_bulk(struct onvm_nf_info *nf_info, struct rte_mbuf** pkts, uint16_t count) -> int onvm_nflib_return_pkt_bulk(struct onvm_nf *nf, struct rte_mbuf** pkts, uint16_t count)`
- `int onvm_nflib_nf_ready(struct onvm_nf_info *info) -> int onvm_nflib_nf_ready(struct onvm_nf *nf)`

- `int onvm_nflib_handle_msg(struct onvm_nf_msg *msg, __attribute__((unused)) struct onvm_nf_info *nf_info) -> int onvm_nflib_handle_msg(struct onvm_nf_msg *msg, struct onvm_nf_local_ctx *nf_local_ctx)`
- `void onvm_nflib_stop(struct onvm_nf_info *nf_info) -> void onvm_nflib_stop(struct onvm_nf_local_ctx *nf_local_ctx)`
- `struct onvm_nf_scale_info *onvm_nflib_get_empty_scaling_config(struct onvm_nf_info *parent_info) -> struct onvm_nf_scale_info *onvm_nflib_get_empty_scaling_config(struct onvm_nf *nf)`
- `struct onvm_nf_scale_info *onvm_nflib_inherit_parent_config(struct onvm_nf_info *parent_info, void *data) -> struct onvm_nf_scale_info *onvm_nflib_inherit_parent_config(struct onvm_nf *nf, void *data)`

v19.05 API Additions:

- `struct onvm_nf_local_ctx *onvm_nflib_init_nf_local_ctx(void)`
- `struct onvm_nf_function_table *onvm_nflib_init_nf_function_table(void)`
- `int onvm_nflib_start_signal_handler(struct onvm_nf_local_ctx *nf_local_ctx, handle_signal_func signal_handler)`
- `int onvm_nflib_send_msg_to_nf(uint16_t dest_nf, void *msg_data)`
- `int onvm_nflib_request_lpm(struct lpm_request *req)`
- `struct onvm_configuration *onvm_nflib_get_onvm_config(void)`

These APIs were previously internal but are now exposed for advanced ring NFs:

- `int onvm_nflib_start_nf(struct onvm_nf_local_ctx *nf_local_ctx, struct onvm_nf_init_cfg *nf_init_cfg)`
- `struct onvm_nf_init_cfg *onvm_nflib_init_nf_init_cfg(const char *tag)`
- `struct onvm_nf_init_cfg *onvm_nflib_inherit_parent_init_cfg(struct onvm_nf *parent)`

v19.05 Removed APIs:

- `int onvm_nflib_run_callback(struct onvm_nf_info* info, pkt_handler_func pkt_handler, callback_handler_func callback_handler)`
- `struct rte_ring *onvm_nflib_get_tx_ring(struct onvm_nf_info* info)`
- `struct rte_ring *onvm_nflib_get_rx_ring(struct onvm_nf_info* info)`
- `struct onvm_nf *onvm_nflib_get_nf(uint16_t id)`
- `void onvm_nflib_set_setup_function(struct onvm_nf_info* info, setup_func setup)`

7.15.7 v19.02 (2/19): Manager Assigned NF Cores, Global Launch Script, DPDK 18.11 Update, Web Stats Overhaul, Load Generator NF, CI (Internal repo only), minor improvements and bug fixes

This release adds several new features and changes how the `onvm_mgr` and NFs start. A CloudLab template is available with the latest release here: <https://www.cloudlab.us/p/GWCloudLab/onvm>

Note: This release makes important changes in how NFs are run and assigned to cores.

Performance: We are aware of some performance irregularities with this release. For example, the first few times a Basic Monitor NF is run we achieve only ~8 Mpps on a CloudLab Wisconsin c220g2 server. After starting and stopping the NF several times, the performance rises to the expected 14.5 Mpps.

Manager Assigned NF Cores:

NFs no longer require a CORE_LIST argument to start, the manager now does core assignment based on the provided core bitmask argument.

NFs now go through the dpdk init process on a default core (currently 0) and then launch a pthread for its main loop, which using the DPDK `rte_thread_set_affinity()` function is affinized to a core obtained from the Manager.

The core info is maintained in a memzone and the Manager keeps track of what cores are used, by how many NFs, and if the cores are reserved as dedicated. The Manager always selects the core with the fewest NFs unless a flag is used when starting an NF.

Usage:

New Manager arguments:

- Hexadecimal bitmask, which tells the onvm_mgr which cores are available for NFs to run on.

The manager now must be run with a command like:

```
1 cd onvm
2 #./go.sh CORE_LIST PORT_BITMASK NF_CORE_BITMASK -s LOG_MODE
3 ./go.sh 0,1,2,3 0x3 0xF0 -s stdout
```

With this command the manager runs on cores 0-3, uses ports 1 and 2 (since `0x3` is binary `0b11`), and will start NFs on cores 4-7 (since `0xF0` is binary `0b11110000`)

New Network Functions arguments:

- `-m` manual core decision mode, NF runs on the core supplied by the `-l` argument if available. If the core is busy or not enabled then returns an error and doesn't start the NF.
- `-s` shared core mode, this will allow multiple NFs to run on the same core. Generally this should be avoided to prevent performance problems. By default, each core is dedicated to a single NF.

These arguments can be set as ONVM_ARGS as detailed below.

API Additions:

- `int onvm_threading_core_affinitize(int core)` - Affinitizes the calling thread to a new core. This is used both internally and by the advanced rings NFs to change execution cores.

Global Launch Script

The example NFs can be started using the `start_nf.sh` script. The script can run any example NF based on the first argument which is the NF name (this is based on the assumption that the name matches the NF folder and the build binary). This removes the need to maintain a separate `go.sh` script for each NF but requires some arguments to be explicitly specified.

The script has 2 modes:

- Simple

```
1 ./start_nf.sh NF_NAME SERVICE_ID (NF_ARGS)
2 ./start_nf.sh speed_tester 1 -d 1
```

- Complex

```
1 ./start_nf.sh NF_NAME DPDK_ARGS -- ONVM_ARGS -- NF_ARGS
2 ./start_nf.sh speed_tester -l 4 -- -s -r 6 -- -d 5
```

All the NF directories have a symlink to :code: `examples/go.sh` file which allows to omit the NF name argument when running the NF from its directory:

```
1 cd speed_tester && ./go.sh 1 -d 1
2 cd speed_tester && ./go.sh -l 4 -- -s -r 6 -- -d 5
```

DPDK 18.11 Update

DPDK submodule no longer points to our fork, we now point to the upstream DPDK repository. This is because mTCP requirements for DPDK have relaxed and they no longer need to have additional patches on top of it.

Also updates Pktgen to 3.6.5 to remain compatible with DPDK v18.11 The dpdk update involves: - Adds NIC ring RSS hashing functions adjustments - Adds NIC ring file descriptor size alignment

Run this to ensure the submodule is up to date:

```
1 git submodule sync
2 git submodule update --init
```

Web Stats Overhaul

Adds a new event logging system which is used for port initialization and NF starting, ready, and stopping events. In the future, this could be used for more complex logging such as service chain based events and for core mappings.

Also contains a complete rewrite of the web frontend. The existing code which primarily used jquery has been rewritten and expanded upon in React, using Flow for type checking rather than a full TypeScript implementation. This allows us to maintain application state across pages and to restore graphs to the fully updated state when returning to a graph from a different page.

Please note that **CSV download has been removed** with this update as storing this much ongoing data negatively impacts application performance. This sort of data collection would be best implemented via grepping or some similar functionality from onvm console output.

Load Generator NF

Adds a Load Generator NF, which sends packets at a specified rate and size, measures tx and rx throughput (pps) and latency. The load_generator NF continuously allocates and sends new packets of a defined size and at a defined rate using the callback_handler function. The max value for the -t pkt_rate argument for this NF will depend on the underlying architecture, for best performance increase it up until you see the NF starting to drop packets.

Example usage with a chain of load_generator <-> simple_forward:

```
1 cd examples/load_generator
2 ./go.sh 1 -d 2 -t 4000000
3
4 cd examples/simple_forward
5 ./go.sh 2 -d 1
```

Example NF output:

```
1 Time elapsed: 24.50
2
3 Tx total packets: 98001437
```

(continues on next page)

(continued from previous page)

```

4 Tx packets sent this iteration: 11
5 Tx rate (set): 40000000
6 Tx rate (average): 3999999.33
7 Tx rate (current): 3999951.01
8
9 Rx total packets: 94412314
10 Rx rate (average): 3853506.69
11 Rx rate (current): 4000021.01
12 Latency (current mean): 4.38 us

```

CI (Internal repo only)

Adds continuous integration to the internal repo. CI will automatically run when a new PR is created or when keyword @onvm is mentioned in a pr comment. CI currently reports the linter output and the Speed Tester NF performance. This will be tested internally and extended to support the public repo when ready.

To achieve this a Flask server listens to events from github, currently only the openNetVM-dev repo is setup for this. In the future we plan to expand this functionality to the public openNetVM repo.

Bug Fixes

- Fix how NF_STOPPED message is sent/processed. This fixes the double shutdown bug (observed in mTCP applications), the fast ctrl-c exit bug and the invalid arguments bug. In all of those cases memory would get corrupted, this bug fix resolves these cases.
- Add out of bounds checks for NF service ids. Before we were not handling cases when a new NF service id exceeded the MAX_SERVICES value or when launching a new NF would exceed the NF_SERVICE_COUNT_MAX value for the given service id.
- Fix the Speed Tester NF to properly exit when passed an invalid MAC addr argument.

7.15.8 v18.11 (11/18): Config files, Multithreading, Better Statistics, and bug fixes

This release adds several new features which cause breaking API changes to existing NFs. NFs must be updated to support the new API required for multithreading support. A CloudLab template is available with the latest release here: <https://www.cloudlab.us/p/GWCloudLab/onvm>

Multithreading:

NFs can now run multiple threads, each with its own set of rings for receiving and transmitting packets. NFs can either start new threads themselves or the NF Manager can send a message to an NF to cause it to scale up.

Usage:

To make an NF start another thread, run the `onvm_nflib_scale(struct onvm_nf_scale_info *scale_info)` function with a struct holding all the information required to start the new NF thread. This can be used to replicate an NF's threads for scalability (all with same service ID), or to support NFs that require several threads performing different types of processing (thus each thread has its own service ID). More info about the multithreading can be found in docs/NF_Dev.md. Example use of multithreading NF scaling can be seen in the `scaling_example` NF.

API Changes:

The prior code relied on global data structures that do not work in a multithreaded environment. As a result, many of the APIs have been refactored to take an `onvm_nf_info` structure, instead of assuming it is available as a global variable.

- `int onvm_nflib_init(int argc, char *argv[], const char *nf_tag);` -> `int onvm_nflib_init(int argc, char *argv[], const char *nf_tag, struct onvm_nf_info **nf_info_p)`
- `void onvm_nflib_stop(void)` -> `void onvm_nflib_stop(struct onvm_nf_info *nf_info)`
- `int onvm_nflib_return_pkt(struct rte_mbuf* pkt)` -> `int onvm_nflib_return_pkt(struct onvm_nf_info *nf_info, struct rte_mbuf* pkt)`
- `int pkt_handler_func(struct rte_mbuf* pkt, struct onvm_pkt_meta* action)` -> `int pkt_handler_func(struct rte_mbuf *pkt, struct onvm_pkt_meta *meta, __attribute__((unused)) struct onvm_nf_info *nf_info)`
- `int callback_handler_func(void)` -> `int callback_handler_func(__attribute__((unused)) struct onvm_nf_info *nf_info)`
- Any existing NFs will need to be modified to support this updated API. Generally this just requires adding a reference to the `onvm_nf_info` struct in the API calls.

NFs also must adjust their Makefiles to include the following libraries:

```
1 CFLAGS += -I$(ONVM)/lib
2 LDFLAGS += $(ONVM)/lib/$(RTE_TARGET)/lib/libonvmhelper.a -lm
```

API Additions:

- `int onvm_nflib_scale(struct onvm_nf_scale_info *scale_info)` launches another NF based on the provided config
- `struct onvm_nf_scale_info * onvm_nflib_get_empty_scaling_config(struct onvm_nf_info *parent_info)` for getting a basic empty scaling config
- `struct onvm_nf_scale_info * onvm_nflib_inherit_parent_config(struct onvm_nf_info *parent_info)` for getting a scaling config with the same functionality (e.g., service ID) as the parent NF
- `void onvm_nflib_set_setup_function(struct onvm_nf_info* info, setup_func setup)` sets the setup function to be automatically executed once before an NF enters the main packet loop

Stats Display

The console stats display has been improved to aggregate stats when running multiple NFs with the same service ID and to add two additional modes: verbose for all stats in human readable format and raw stats dump for easy script parsing. The NF TX stat has been updated to also include tonf traffic.

Usage:

- For normal mode no extra steps are required
- For verbose mode run the manager with `-v` flag
- For raw stats dump use the `-vv` flag

Config File Support:

ONVM now supports JSON config files, which can be loaded through the API provided in `onvm_config_common.h`. This allows various settings of either the ONVM manager or NFs to be set in a JSON config file and loaded into code, as opposed to needing to be passed in via the command line.

Usage:

- All example NFs now support passing DPDK and ONVM arguments in a config file by using the `-F config.json` flag when running an NF executable or a `go.sh` script. See `docs/examples.md` for more details.

API Changes: - `nflib.c` was not changed from an NF-developer standpoint, but it was modified to include a check for the `-F` flag, which indicates that a config file should be read to launch an NF.

API Additions: - `cJSON* onvm_config_parse_file(const char* filename)`: Reads a JSON config and stores the contents in a `cJSON` struct. For further reference on `cJSON`, see its [documentation](#). - `int onvm_config_create_nf_arg_list(cJSON* config, int* argc, char** argv[])`: Given a `cJSON` struct and pointers to the original command line arguments, generate a new `argc` and `argv` using the config file values.

Minor improvements

- **Return packets in bulk:** Adds support for returning packets in bulk instead of one by one by using `onvm_nflib_return_pkt_bulk`. Useful for functions that buffer a group of packets before returning them for processing or for NFs that create batches of packets in the fast path. *No breaking API changes.*
- **Updated corehelper.py script:** Fixed the `scripts/corehelper.py` file so that it correctly reports recommended core usage instructions. The script assumes a single CPU socket system and verifies that hyperthreading is disabled.
- **Adjusted default number of TX queues:** Previously, the ONVM manager always started `MAX_NFS` transmit queues on each NIC port. This is unnecessary and causes a problem with SR-IOV and NICs with limited queue support. Now the manager creates one queue per TX thread.
- Bug fixes were made to [prevent a crash](#) of `speed_tester` during allocation of packets when there are no free mbufs and to [fix an invalid path](#) causing an error when attempting to use `Pktgen` with the `run-pktgen.sh` script. Additionally, a few [minor documentation edits](#) were made.

7.15.9 v18.05 (5/31/18): Bug Fixes, Latency Measurements, and Docker Image

This release adds a feature to the Speed Tester example NF to support latency measurements by using the `-l` flag. Latency is calculated by writing a timestamp into the packet body and comparing this value when the packet is returned to the Speed Tester NF. A sample use case is to run 3 speed tester NFs configured to send in a chain, with the last NF sending back to the first. The first NF can use the `-l` flag to measure latency for this chain. Note that only one NF in a chain should be using the flag since otherwise timestamp information written to the packet will conflict.

It also makes minor changes to the setup scripts to work better in NSF CloudLab environments.

We now provide a docker container image that can be used to easily run NFs inside containers. See the [Docker Docs](#) for more information.

OpenNetVM support has now been integrated into the mainline [mTCP repository](#).

Finally, we are now adding issues to the GitHub Issue Tracker with the [Good First Issue](#) label to help others find ways to contribute to the project. Please take a look and contribute a pull request!

An NSF CloudLab template including OpenNetVM 18.05, mTCP, and some basic networking utilities is available here: <https://www.cloudlab.us/p/GWCloudLab/onvm-18.05>

No API changes were introduced in this release.

7.15.10 v18.03 (3/27/18): Updated DPDK and preliminary mTCP support

This release updates the DPDK submodule to use version 17.08. This DPDK update caused breaking changes to its API, so updates have been made to the OpenNetVM manager and example NFs to support this change.

In order to update to the latest version of DPDK you must run:

```
1 git submodule update --init
```

And then rebuild DPDK using the [install guide](#) or running these commands:

```
1 cd dpdk
2 make clean
3 make config T=$RTE_TARGET
4 make T=$RTE_TARGET -j 8
5 make install T=$RTE_TARGET -j 8
```

(you may need to install the `libnuma-dev` package if you get compilation errors)

This update also includes preliminary support for mTCP-based endpoint NFs. Our OpenNetVM driver has been merged into the [develop branch of mTCP](#). This allows you to run services like high performance web servers on an integrated platform with other middleboxes. See the mTCP repository for usage instructions.

Other changes include:

- Adds a new “Router NF” example which can be used to redirect packets to specific NFs based on their IP. This is currently designed for simple scenarios where a small number of IPs are matched to NFs acting as connection terminating endpoints (e.g., mTCP-based servers).
- Bug Fix in ARP NF to properly handle replies based on the ARP OP code.
- Updated pktgen submodule to 3.49 which works with DPDK 17.08.

An NSF CloudLab template including OpenNetVM 18.03, mTCP, and some basic networking utilities is available here: <https://www.cloudlab.us/p/GWCloudLab/onvm-18.03>

No API changes were introduced in this release.

7.15.11 v18.1 (1/31/18): Bug Fixes and Speed Tester improvements

This release includes several bug fixes including:

- Changed macro and inline function declarations to improve compatibility with 3rd party libraries and newer gcc versions (tested with 4.8 and 5.4)
- Solved memory leak in SDN flow table example
- Load Balancer NF now correctly updates MAC address on outgoing packets to backend servers

Improvements:

- Speed Tester NF now supports a `-c` argument indicating how many packets should be created. If combined with the PCAP replay flag, this parameter controls how many of packets in the trace will be transmitted. A larger packet count may be required when trying to use Speed Tester to saturate a chain of network functions.

No API changes were introduced in this release.

7.15.12 v17.11 (11/16/17): New TX thread architecture, realistic NF examples, better stats, messaging, and more

Since the last official release there have been substantial changes to openNetVM, including the switch to date based versioning mentioned above. Changes include:

- New TX architecture: previously NFs enqueued packets into a TX ring that was read by TX threads in the manager, which consumed significant CPU resources. By moving TX thread logic to the NF side, ONVM can run with fewer cores, improving efficiency. NFs can then directly pass packets which saves enqueueing/dequeueing to an extra ring. TX threads still send packets out the NIC, but NFs primarily do packet passing—it is suggested to run the system with at least 1 TX thread to handle outgoing packets. Despite these changes, TX threads can still perform the same work that they did before. If a user would like to run ONVM with TX threads handling all packet passing, they must set `NF_HANDLE_TX` to 0 in `onvm_common.h`
 - Our tests show this change increases NF transmit speed from 20 Mpps to 41 Mpps with the Speed Tester NF benchmark, while consuming fewer cores.
- New NFs: we have developed several new sample NFs, including:
 - `examples/ndpi_stats` uses the [nDPI library](#) for deep packet inspection to determine the protocol of each flow.
 - `examples/flow_tracker` illustrates how to use ONVM's flow table library to track the list of open connections and print information about them.
 - `examples/arp_response` can be used to assign an IP to the NICs managed by openNetVM. The NF is capable of responding to ARP requests. This facilitates NFs that act as connection endpoints, load balancers, etc.
 - `examples/load_balancer` is a layer 3, round-robin load balancer. When a packet arrives the NF checks whether it is from an already existing flow. If not, it creates a new flow entry and assigns it to a destination backend server. This NF uses ARP support to assign an accessible IP to the openNetVM host running the load balancer.
 - [Snort NF](#) provides a version of the Snort intrusion detection system ported to openNetVM.
- [PCAP replay](#): the Speed Tester NF can now load a packet trace file and use that to generate the packets that it transmits.
- [NF idle call back](#): Traditionally, NFs would wait until the ONVM manager puts packets on their Rx buffer and then calls their packet handler function to process them. This meant that NFs would sit idle until they have some packets to process. With this change, NFs can now run at any time even if there are no packets to process. NFs can provide a callback handler function to be registered with NFLib. Once this callback handler is registered with NFLib, the function will be run constantly even if there are no packets to be processed.
- [Web-based stats](#): the ONVM manager can now display statistics about the active NFs. See `onvm_web/` for more information.
- [NF-Manager Messaging Interface](#): We have expanded the interface between the manager and NFs to allow more flexible message passing.
- A multitude of other bug fixes, documentation improvements, etc!

7.15.13 v1.1.0 (1/25/17): Refactoring to library, new NFs

This release refactored the code into a proper library, making it easier to include with more advanced NFs. We also added new AES encryption and decryption NFs that operate on UDP packets.

7.15.14 v1.0.0 (8/25/16): Refactoring to improve code organization

A big set of commits to clean the structure and simplify onvm source code. We separated all functions into the main.c of the manager into modules:

- `onvm_stats` : functions displaying statistics
- `onvm_pkt` : functions related to packet processing
- `onvm_nf` : functions related to NFs management.

Each module comes with a header file with commented prototypes. And each c and h file has been “cut” into parts:

- interfaces, or functions called outside of the module
- internal functions, the functions called only inside the module and doing all the work
- helper functions, simple and short functions used many times through the module.

API Changes:

- NFs now need to call functions like `onvm_nflib_*` instead of `onvm_nf_*`. For example, `onvm_nflib_init` instead of `onvm_nf_init`. The example NFs have all been updated accordingly.
- NF Makefiles need to be updated to find the path to `onvm_nflib`.

7.15.15 4/24/16: Initial Release

Initial source code release.

7.16 Frequently Asked Questions

This page answers common questions about OpenNetVM’s Architecture.

7.16.1 Does OpenNetVM use Virtual Machines?

Not anymore! Originally, the platform was designed to run KVM-based virtual machines. However, now we use regular processes for NFs, which can be optionally deployed inside containers. The name has been kept for historical reasons.

7.16.2 What are the system’s main components?

An OpenNetVM host runs the Manager plus one or more Network Functions.

The OpenNetVM manager has a minimum of three threads: - The [RX Thread\(s\)](#) receives packets from the NIC and delivers them to the first NF in a chain. By default there is only 1 RX thread, but this can be changed with the `ONVM_NUM_RX_THREADS` macro in `onvm_init.h` - The [TX Thread\(s\)](#) retrieves packets after NFs finish processing them, and sends them out the NIC. If you have multiple NIC ports you may need to increase the number of TX threads in the manager command line arguments. - The [Master Stats Thread](#) detects when new NFs start and helps initialize them. It also periodically prints out statistics. - (Optional) The [Wakeup Thread\(s\)](#) is used if shared CPU mode is enabled

with the `-c` argument. The thread checks whether any sleeping NFs have received packets and wakes them with a semaphore. This allows NFs to share CPUs without constantly polling for packets.

Network functions are typically a single thread with two portions: - `nflib` provides the library functions to interact with the Manager. When NFs start they typically call `onvm_nflib_run` to start the main run loop. - When packets arrive for the NF, the `nflib` run loop will execute the NF's packet handler function. This should contain the application logic for the NF. - Alternatively, NFs can use the Advanced Ring mode to have direct access to their incoming packet queues, allowing for more complex execution models. - The `onvm_nflib_scale` API can be used for NFs that require multiple threads. The Manager views each scaled thread as a distinct NF with its own packet queues. - See the NF API page for more details.

7.16.3 Where are packets first received?

Packets initially arrive on a queue on a NIC port. The manager's RX and TX thread handle routing of packets between NFs and the NIC ports. The RX thread will remove a batch of packets from a NIC queue and examine them individually to determine their defined actions (drop, send to another NF, or send out of the system). The TX thread performs a similar function, but instead of reading from a NIC queue, they read from the transmit ring buffers of NFs and either send the packets out a NIC port or to a different NF service ring.

7.16.4 How do NFs process packets?

Network functions interact with the manager using the NFLib API. The `NFLib run function` then polls its receive ring for packets arriving from the manager or other NFs by calling the function `onvm_nflib_thread_main_loop`. For each packet received the NFLib executes a call back function provided by the NF developer to process the packet. An example of this process can be seen in the `bridge` NF example. When a NF runs its callback to handle an incoming packet, it is given a packet descriptor that contains the address of the packet body and metadata. The NF will then perform the desired functionality.

7.16.5 What actions can NFs perform?

There are many `examples` that show the powerful packet handling, messaging, and overall possible applications of Network Functions handled by the ONVM manager. The main NFs to jump into first are `Speed Tester` and `Basic Monitor`, because of their readability and easy command-line interfaces. Initially, NFs can be run with a great deal of mandatory/optional arguments. An example of this is `parse_app_args`, which, for NFs like Firewall, can be used to input `json configurations`. By including `#include "cJSON.h"`, an NF can parse a JSON file and instantiate a `cJSON` struct with `onvm_config_parse_file`. Doing this can help with scalable testing for many types of NFs working together. For examples of how to pass NF/Manager-specific arguments, see the NF start scripts `start_nf.sh` and `go.sh`.

ONVM provides a lot of packet parsing APIs that are crucial for handling different types of data, through either UDP or TCP connections. These `definitions` include checking the packet type, swapping mac address of the destination, parsing the IP address, and many more useful functions. NFs also have access to packet meta data defined `here` which can be tracked and updated depending on the purpose of the NF. This allows NFs to forward, drop, send packets to another NF, defining the primary packet handler, such as a Firewall, and secondaries that receive the valid forwarded packets. More information on sending packets can be seen below in **How do packets move between network functions**.

In main of each NF, a few functions should be called to set signals, data structures, and the NF context, handled by the running manager. These include `onvm_nflib_init`, `onvm_nflib_init_nf_local_ctx`, and `onvm_nflib_start_signal_handler`. There are certainly others, most of which are demonstrated in the main functions of the example NFs, such as `Speed Tester`. With these initializations, NFs set communication lines between themselves and the manager, so packets are handled according to the NFs job. For example, while Firewall's `nf_function_table->pkt_handler` is `packet_handler`, it doesn't have a `user_actions` function callback like `Basic Monitor` does. This is ok, it just shows the flexibility that NFs have to carry out the tasks they want with the packets that have been passed to them from the queue.

Finally, a recent major development to the ONVM platform has been scalable NFs running through shared cores. Previously, each NF was designated their own core to run on, but enabling shared cores in the manager will now allow multiple NFs running on the same core. As shown in the stats output, if the manager `go.sh` is run with `-c`, NFs will be put to sleep when no packets are sent to them, and awoken through signals when they have something to do. A great example to get started is running [Scaling](#), where a number of children can be passed by enabling with `-c` and set with `:code: ^-n ^`. All of these options show the different capabilities of network functions in onvm. Read through the documentation on each NF mentioned here to get a better idea of the pieces that can be extracted to combine with another NF.

7.16.6 How do packets move between network functions?

When a network function is instantiated, a `struct onvm_nf` structure is created alongside it. Along with important pieces of network function specific values, it also contains a ring buffer to store packets. The important field values involving NF to NF communication are the `rx_ring` and the `nf_tx_mgr` data structures.

If a network function wishes to send a packet to another network functions `rx_ring`, it must modify the packets metadata within the sending network functions `packet_handler`. The `onvm_pkt_meta` allows for this metadata implementation, which allows the [main network function loop](#) to understand what to do next with the packet (drop, send to another NF, or send out of the system).

An example of this process can be seen in the [simple_forward](#) network function. Here, the `packet_handler` receives a packet and proceeds to modify its corresponding meta-data. `meta->action` must be set to `ONVM_NF_TONF` along with the destination (service ID) of the intended recipient. Upon completion, the main network function loop will enqueue the packets onto the recipient network functions `rx_ring`.

7.17 CI Development

This page will serve as description and development plans for our CI system

Current CI maintainers: primary - @kevindweb, secondary - @koolzz

CI is being moved to a new repository inside `sdnfv` called `onvm-ci` you can find it [here](#).

7.17.1 Currently CI does 3 main functions:

1. Checks if the PR was submitted to the `develop` branch
2. Runs Linter and reports errors (only on lines modified by the diff)
3. Runs the Speed Tester NF and Pktgen and reports measured performance

7.17.2 Done

1. Event queue ✓

- When the CI is busy with another request it will deny all other requests and respond with a *I'm busy, retry in 10 min* message, which is clearly not ideal for a good CI system. CI should implement some type of request queue to resolve this.

2. Pktgen Tests in addition to Speed Tester

- Pktgen testing, with a 2 node setup where one node would be running Pktgen and the other would run the Basic Monitor NF. This is done both by reserving 2 worker nodes that are directly connected.

7.17.3 Planned features

1. Expanding performance test coverage

- We should also include mTCP test with epsserver and epwget
- Additionally, we need a X node chain speed tester performance

2. Linter coverage

- Currently we only lint .c, .h, .cpp files. We have a lot of python/shell scripts that we should also be linting.

3. Github Messaging

- With the new CI modes and a message editing api from github3.py, we are able to post responses as they come in.
- Linting data comes much faster than speed testing for example, so NF data will be appended to an already posted message in Github for quick responses.

4. Better resource management(long term)

- Utilizing the new cluster software we can figure out which nodes are unused and always have them ready to run CI tests. This would help speed up the CI process.