Introduction

OpenNetVM (ONVM) is a highly efficient network packet processing framework that provides simple abstraction for developing and running network functions. The ONVM platform provides load balancing, flexible packet flow management, individualized service abstractions, and basic software defined networking (SDN) capabilities. The platform already offers best-in-class performance: it has the ability to maintain 40 Gbps speeds while routing packets through dynamically created chains of network functions (NFs). OpenNetVM also lowers the barriers to deploying production network functions in software because it can run on inexpensive commodity hardware. However, this idea can be improved upon: we aim to make it possibly for multiple instances of ONVM to process packets in tandem, with seamless inter-routing, making it possible to run software-based network services at production scale.

Users & Use Cases

OpenNetVM is is intended to benefit enterprise customers. These companies run large-scale, high-performance networking applications. These network applications tend to require expensive specialized hardware that can only be purchased from one of few suppliers. For example, a large enterprise typically has an Intrusion Detection System (IDS) protecting the entry point to their network. A popular website or an entry point to the corporate VPN may saturate a 10 Gigabit connection with incoming requests. All of this traffic needs to be screened for malicious traffic. Furthermore, the enterprise may also require a firewall, DDOS detection system, and dynamic routing (like Software Defined Networking); all of these features require different expensive and specialized hardware. So, when building an enterprise-quality network, the infrastructure costs will add up and can quickly become burdensome.

OpenNetVM will allow large enterprises to run their high-performance network applications at a fraction of the cost. OpenNetVM can run on commodity hardware which vastly reduces the amount of capital required to spin up a corporate network. This platform will not sacrifice performance either; packets can be processed at 10 Gigabit speeds, the same speeds as the more expensive dedicated hardware. By utilizing the flexibility of software, OpenNetVM also integrates dynamic network topology. Packets can be routed via a “service chain,” which means a packet flow can be processed by multiple network functions (NFs). This functionality allows the functionality of what would previously require multiple dedicated appliances to be consolidated into a smaller number of commodity servers. Furthermore, as system requirements change over time, the topology of the network function interconnections can be dynamically

modified. For example, if the company's website is experiencing additional load, more instances of a load balancer can be started to accommodate the additional traffic and spread the incoming requests appropriately.

OpenNetVM is able to provide these revolutionary features by implementing Network Function Virtualization (NFV). For every "appliance" that an enterprise IT department would want on their network, they can abstract the functionality into software and run it on the OpenNetVM platform. OpenNetVM uses the high-performance DPDK networking library to interface directly with the hardware. Packets can be shared between NFs and processed without copying overhead – the platform works entirely with shared memory. Furthermore, the OpenNetVM platform will be able to function in a distributed manner: packet flows can be seamlessly routed among multiple hosts to be processed in an optimal way. This means the system can scale seamlessly and automatically recover from failed hosts without loss in service. Automatic failover and horizontal scaling provide features that enterprise customers can rely on in production operations. Traditionally, these features were very expensive to implement, as they require specialized networking hardware. OpenNetVM can provide these advantages for free on inexpensive servers by harnessing the power and flexibility of software.

System Components

Table 1: Component Overviews & Timeline

Component	Description	Start Date	End Date[1]
Distributed Datastore	The datastore will be the primary means of sharing data between multiple instances of the ONVM manager.	9/15/2016	10/16/2016
VXLAN Encapsulation/ Decapsulation Module	The encap/decap module will perform VXLAN encapsulation and decapsulation of packets routed to and from manager instances	10/15/2016	11/10/2016
Manager to NF Communication Channel	The ONVM Manager should be able to pass control messages and metadata (in addition to packets) to NFs	12/1/2016	*1/15/2017 (projected)*
NF Auto Scaling Module	The ONVM Manager will be able to start additional copies of an NF when additional capacity is needed and stop copies when there is excess capacity	1/20/2017	*3/30/2017 (projected)*

[1] Projected completion dates include ~1 week for integration and testing

Table 2: Component Requirements

Component	Functional Requirements	Non-Functional Requirements
Distributed Datastore	• Store MAC addresses for managers • Map running managers to their services • Store running NF statistics	• Not consume excess bandwidth from packet flow • Be optimized for a read-heavy workload • <500ms update latency
VXLAN Encapsulation/ Decapsulation Module	• Adhere to RFC 7348[2] standard for VXLAN • Set the destination L2 address to another ONVM instance	• Minimal latency • Fault-tolerant • Be distinguishable from non VXLAN encapsulated packets
Manager to NF Communication Channel	• Define control message format • Use a shared memory channel between manager and NF	• Small volume of important messages • Minimal interruption in packet processing critical path
NF Auto Scaling Module	• Communicate “start” and “stop” messages over the messaging channel • Use statistics stored in the datastore to make scaling decisions	• Computing when to start and stop NFs should happen off the critical packet path

[2] https://tools.ietf.org/html/rfc7348

External Dependencies

Distributed OpenNetVM depends on a two external, third-party projects. Each provides something unique to make the project better. OpenNetVM includes the source code for these libraries as git submodules and provides build/install scripts to perform the required initialization steps for each library.

First, OpenNetVM depends on Intel's Data Plane Development Kit[3] (DPDK). DPDK replaces the Linux kernel's networking stack and allows user-space applications to directly access packets coming in from the NIC. OpenNetVM leverages DPDK's Environment Abstraction Layer (EAL) and uses its standard data structures to pass packet references between the manager and NFs. Packet data is stored in a HugePage[4] region shared between the manager and each NFs. DPDK is installed and configured using a script[5], found in the OpenNetvM git repository.

OpenNetVM also depends on Apache ZooKeeper[6], a library that provides a highly reliable distributed coordination service. ZooKeeper is used as a centralized source to maintain configuration information and share it between running instances of the ONVM manager. This library is used as the backbone of the Distributed Datastore application component. ZooKeeper is used to store information about the current state of the cluster and is queried when algorithmic choices require knowledge of such global state. ZooKeeper is installed and configured using a script[7] in the OpenNetVM git repository.

Command Line Interface

There are a few different interfaces exported to the user. When installing the system, documentation is provided[8] on the steps to be run, as well as the appropriate environment variables that need to be set. Additionally, each component of the OpenNetVM system has a "go" script, which is a simple wrapper around the raw commands to be run (these commands are more complicated due to arguments for the DPDK library). These scripts support running OpenNetVM in both distributed and non-distributed modes. The user simply has to export the environment variables described in the install documentation above, and the rest will "instamagically" happen. To run the system, refer to the "Running Examples" documentation[9] page in the git repository. A short example is below, for additional reference.

[3] http://dpdk.org/
[4] https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
[5] https://github.com/phil-lopreiato/openNetVM/blob/senior-design/scripts/install.sh
[6] http://zookeeper.apache.org/
[7] https://github.com/phil-lopreiato/openNetVM/blob/senior-design/scripts/zookeeper-install.sh
[8] https://github.com/phil-lopreiato/openNetVM/blob/senior-design/docs/Install.md
[9] https://github.com/phil-lopreiato/openNetVM/blob/senior-design/docs/Examples.md

Short Example

This example will send packets between two OpenNetVM nodes. This assumes the system has already been installed and configured on two machines (see above). Any `go.sh` script can be run with no arguments to see a description of the possible arguments.

```
$ ./onvm/go.sh # For the manager
./onvm/go.sh [cpu-list] [port-bitmask]

$ ./examples/basic_monitor/go.sh  # Basic Monitor NF
./examples/basic_monitor/go.sh CPU-LIST SERVICE-ID [-p PRINT] [-n NF-ID]

$ ./examples/speed_tester/go.sh  # Speed Tester NF
./examples/speed_tester/go.sh CPU-LIST SERVICE-ID DST [-p PRINT] [-n NF-ID]
```

For reference, you can test the capabilities of the current machine using the corehelper script:

```
./scripts/corehelper.py
```

Build the manager and all example NFs on both machines:

```
cd onvm && make clean && make
cd examples && make clean && make
```

On both machines, run the manager. On the nodes in our cluster, we can run with 4 cores and a default service ID of 1:

```
./onvm/go.sh 0,1,2,3 1
```

Run a Basic Monitor NF on the first machine (with service ID 2):

```
./examples/basic_monitor/go.sh 4 2
```

Run a speed tester NF on the second machine (with service ID 3, destination service ID 2):

```
./examples/speed_tester/go.sh 4 3 2
```

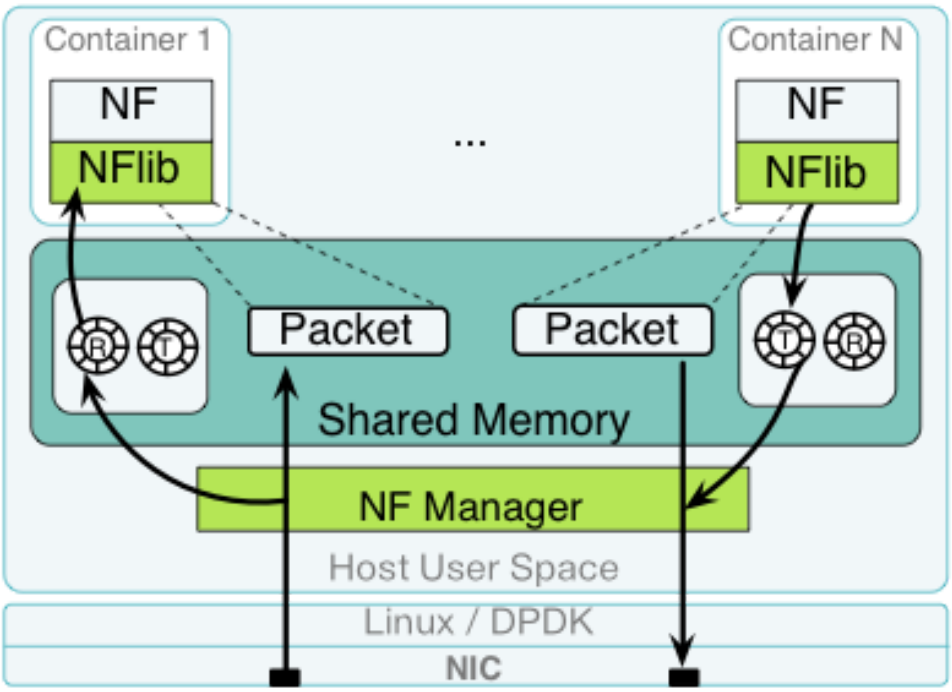
Then, on the Monitor NF, you should see the received packet count increase. This means the packets, created on the first machine, were encapsulated, sent over the network, decapsulated, and processed by the second manager. Here is an example of the output:

```
PACKETS
-----
Port : 0
Size : 2
Hash : 0
N° : 127
```

System Overview

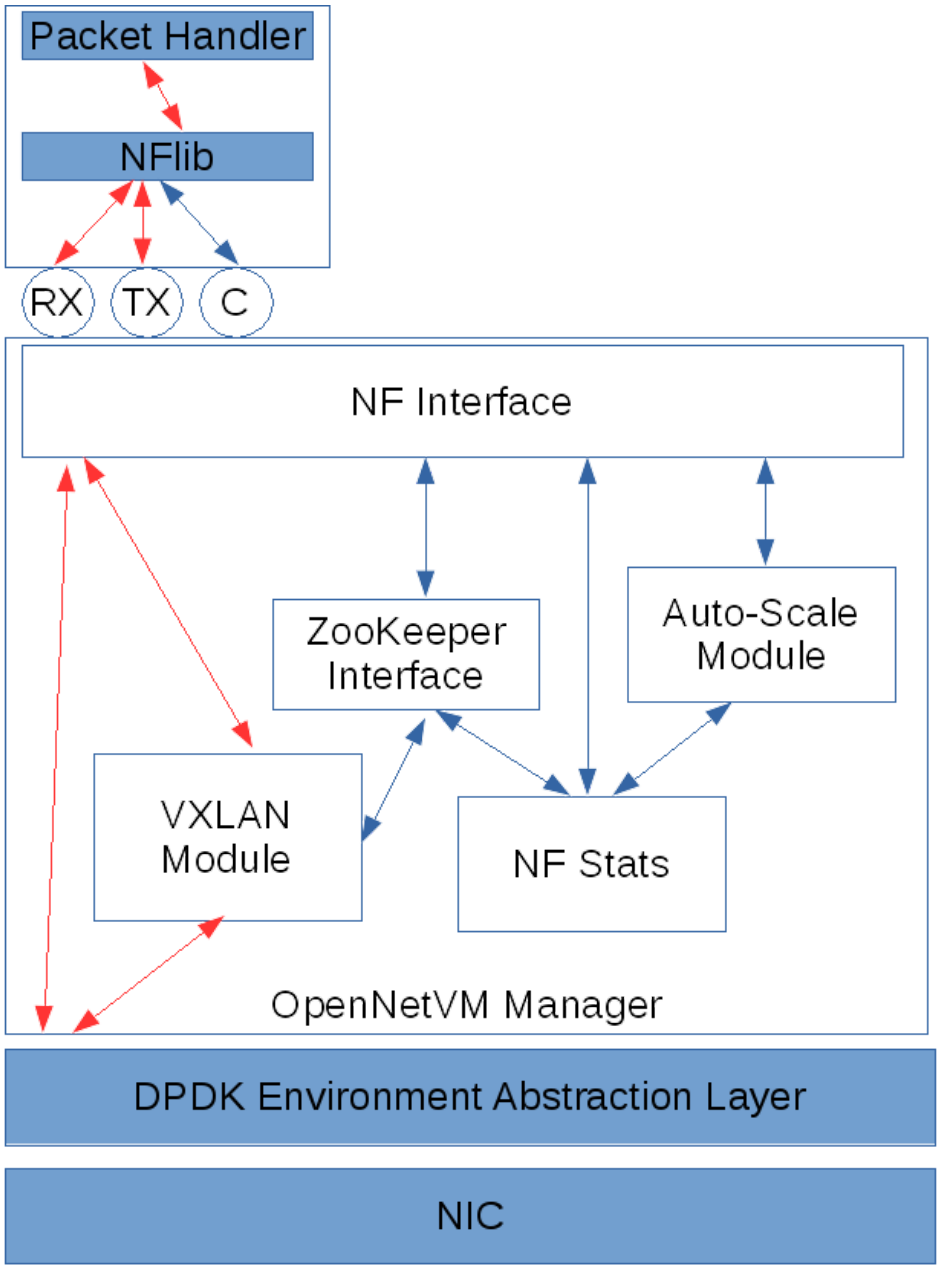
The following figures show how the different components of OpenNetVM interact.

Figure 1: Single OpenNetVM Instance



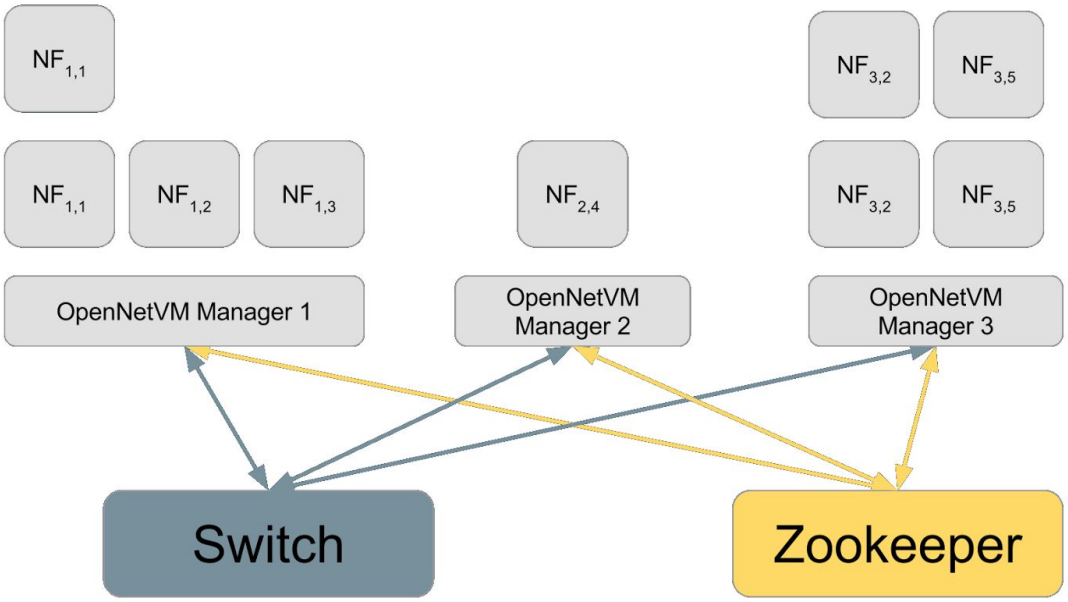
This figure shows the layout of a single instance of OpenNetVM running on a single server. The manager runs as a Linux process and leverages Intel DPDK to read packets from the NIC and store them in shared memory, for access by various NFs. Packet references are sent to NFs for processing via shared memory-backed queues. Each NF has one queue for transmitting packets and one for receiving packets (TX and RX rings).

Figure 2: OpenNetVM Manager Detail



This figure outlines the interactions between various components within OpenNetVM. The red arrows denote the packet processing critical path, for which performance is of the utmost importance. Between the Manager and NF, there are three ring buffers for communication (Packet RX, Packet TX, and Control Messages). The NF interface controls the shared memory boundary between the manager and NFs (enqueuing/dequeuing packets, handling routing, and starting/stopping NFs during auto-scaling). The ZooKeeper reads from and writes to the ZooKeeper service running locally on this machine (and is connected to the full ensemble, which supplies it with a global view of the cluster).

Figure 3: OpenNetVM Cluster



This figure shows how multiple instances of OpenNetVM communicate and coordinate. All instances in the cluster are connected to the same 10 Gigabit switch and each are connected to a ZooKeeper ensemble on a different (1G) network. This setup means that coordination traffic will not subtract from packet processing bandwidth. Each instance knows what services each other instance is running and the physical (L2) addresses of each. Then, the ZooKeeper datastore can be used to inform decisions about which instances to route packets to.