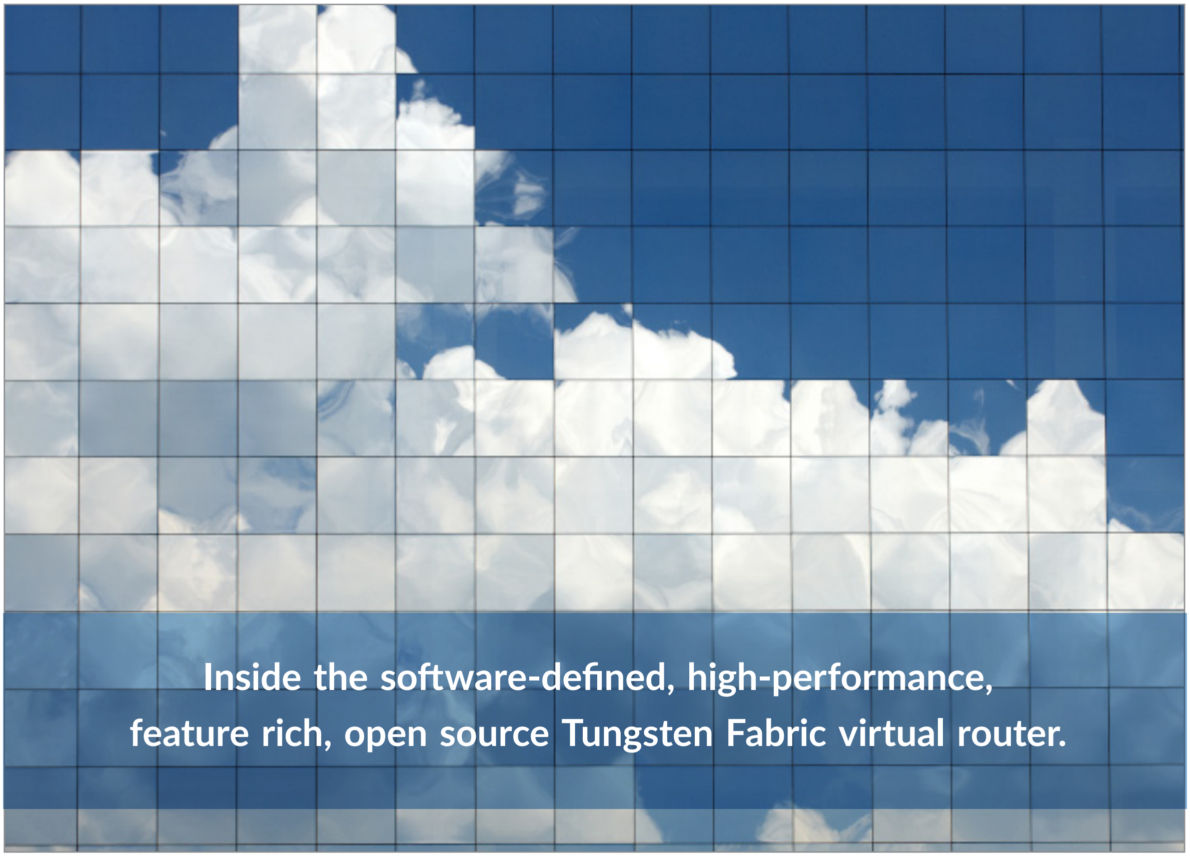
JUNIPER NETWORKS | Engineering Simplicity

DAY ONE: CONTRAIL DPDK vROUTER



Inside the software-defined, high-performance, feature rich, open source Tungsten Fabric virtual router.

By Kiran KN, Ping Song, Przemyslaw Grygiel, Laurent Durand

DAY ONE: CONTRAIL DPDK vROUTER

Let's do SDN! By using DPDK as an open source data plane for Contrail/Tungsten Fabric vRouter, you'll learn about DPDK and its related technologies (huge page, NUMA, CPU pinning, etc.) as well as the vRouter DPDK design and details on installation.

DPDK vRouter is one of the most important SDN data plane implementations and this book shows you how to increase its data plane performance, thus accelerating network applications and optimizing NFV and SDN data planes. It's all here: six chapters and over a hundred illustrations to guide you through the world of SDN and Contrail Tungsten Fabric.

"This book is a must read for network software developers. It covers in great detail how Network Applications can be accelerated with Data Plane Development Kit (DPDK). In particular, it describes libraries, tools and techniques to optimize Network Function Virtualization (NFV) and Software Defined Network (SDN) data plane performance by more than a factor of ten. It is an excellent showcase of a close knit collaboration between Intel and Juniper engineers over many years to deliver high performance and cloud scale applications for the networking industry. I am impressed with its thoroughness and wealth of practical, hands-on information. In summary, this book rocks."

Rajesh Gadiyar, Vice President and CTO, Network Platforms Group, Intel Corporation

"This is a superb book. Four Juniper engineers have combined their experience in working with DPDK and its use as an open source data plane for SDN (vRouter). Step-by-step the authors describe the innards of vRouter and show you how to configure, optimize, and troubleshoot one of the best SDN solutions in the marketplace. Congratulations to the authors, and to you the reader who are about to be impressed."

Raj Yavatkar, CTO, Juniper Networks

IT'S DAY ONE AND YOU HAVE A JOB TO DO, SO LEARN HOW TO:

- Understand SDN basics.
- Apply DPDK and network virtualization technologies.
- Identify Contrail vRouter DPDK internal architectures.
- Manage packet forwarding flows in DPDK vRouter.
- Install contrail and the traffic testing tools.
- Be familiar with the utilities available for DPDK vRouter to troubleshoot and analyze performance.

ISBN 978-1936779895
51400
9 781936 779895

Juniper Networks Books are focused on network reliability and efficiency. Peruse the complete library at www.juniper.net/books.

JUNIPER NETWORKS

DAY ONE: CONTRAIL DPDK vROUTER

Kiran, Song, Grygiel, Durand

Day One: Contrail DPDK vRouter

by Kiran K N, Ping Song, Przemyslaw Grygiel,
Laurent Durand

Juniper Networks

Published by Juniper Networks Books
Authors: Kiran K N, Ping Song, Przemyslaw Grygiel, Laurent Durand
Technical Reviewers: Vincent Zhang, Richard Roberts, T. Sridhar
Editor in Chief: Patrick Ames
Copyeditor: Nancy Koerbel
Printed in the USA by Vervante Corporation.
Version History: v1, January. 2021
2 3 4 5 6 7 8 9 10
Comments, errata: dayone@juniper.net

About the Authors
Kiran K N is a Principal Engineer in Juniper Networks, with more than 15 years of experience in the networking industry. He graduated from the Indian Institute of Technology with a Masters degree in Computer Science. His current area of interest is Software Defined Networks and datapath technology. He is an expert in DPDK and an active developer of Contrail vRouter. He has made significant contributions towards the architecture, hardening, features, and performance enhancements of vRouter.

Ping Song is a technical support engineer at Juniper Networks. As a network engineer, he currently supports customers building and maintaining their data centers with Juniper contrail networking SDN solution. Ping is also an enthusiastic Linux and Vim power user. After work, Ping enjoys gardening work and reading Chinese literature. Ping holds active double CCIE#26084 (R&S, SP) and triple JNCIE (SP#2178, ENT#775, DC#239) certifications.

Przemyslaw Grygiel is a Principal Engineer in Juniper Networks with 18 years of experience in the cloud and networking industry. He is an expert in cloud computing and SDNs and has seven years of experience with Juniper Contrail. Przemyslaw holds CCIE #15278 (R&S).

Laurent Durand is a technical consultant in Juniper Networks He started as C/C++ developer 25 years ago. In early 2000, he worked as a Network and System Engineer. Later, while working as a Network Architect, he designed country wide IP MPLS networks (Mobile and Fix), and VoIP solutions for some European Telcos. For the last few years he has worked as Cloud solutions Architect; he has also been working on SDN infrastructures and teaches Network Virtualization in some Paris engineering schools

Authors' Acknowledgments
We'd all like to thank Patrick Ames for his encouragement and support during the time of writing this book. And thank you to Nancy Koerbel for the expert editing and proofing.

Kiran: Writing the book would not have been possible without the help and support from my family members and my managers at Juniper. I would like to thank my parents - Mr. Prasad K N and Mrs. Gowri Prasad K N for their constant support. I would also like to thank Juniper CTO Raj Yavatkar, Juniper VPs - Rakesh Manocha, and T. Sridhar for their help and guidance in writing this book. Last but not least, I want to thank my teammates Ping, Przemyslaw, and Laurent for a great and fun-filled collaboration.

Ping: This book was written during the most special year - 2020. Needless to say, it has been tough and full of uncertainties for everyone, but I am positive we will get through this soon. I would like to thank Laurent, Kiran, and Przemysław, my partners in this book, for their deep knowledge and helpful technical discussions during the past few months. Thanks to my manager Siew Ng, for being supportive of the contrail book project, and for allowing me to focus more on the book during the last few weeks. In that regard, I'd like to also thank my CFTS SDN team-mates, who offloaded parts of my routine work during the book writing process. Lastly, I'd like to thank my wife, Sandy, for her support on my work during the pandemic, and my lovely kids Xixi and Jeremy for all the joy they brought. Thank you all!

Przemysław: I would like to thank my family and my manager for their support during writing of this book.

Laurent: I'd like to thank all my teammates for their support on Contrail DPDK and for their deep understanding and troubleshooting.

Welcome to Day One

This book is part of the *Day One* library, produced and published by Juniper Networks Books. *Day One* books cover the Junos OS and Juniper Networks network-administration with straightforward explanations, step-by-step instructions, and practical examples that are easy to follow.

- Download a free PDF edition at http://www.juniper.net/dayone
- PDF books are available on the Juniper app: Junos Genius
- Purchase the paper edition at Vervante Corporation (www.vervante.com).

Terminology

Throughout this book, the authors use the terms Contrail, OpenContrail, Tungsten Fabric, and TF interchangeably.

Key DPDK Resources

This book is not a substitute for the excellent documentation that exists in the Juniper TechLibrary. The authors of this book assume you are familiar with Contrail documentation.

- The latest Tungsten Fabric Architecture can be found at: https://tungstenfabric.github.io/website/Tungsten-Fabric-Architecture.html.
- The authors keep a GitHub website at: https://github.com/pinggit/dpdk-contrail-book, where you can find the book's content, figures, technical discussions, and more. Add comments, suggestions, or questions regarding the book, too.

What You Need Before Reading This Book:

- You need a basic understanding of IP networking.
- You need a basic understanding of Linux.
- This book assumes that you have some basic knowledge about SDN and Contrail architecture.
- You will need to have two or more Intel Xeon servers with Linux OS and DPDK-compatible NIC cards (For example Intel x710, Intel 82599).

After Reading This Book...

- You will understand SDN basics.
- You will be able to apply DPDK and network virtualization technologies.
- You will be able to identify Contrail vRouter DPDK internal architectures.
- You will be able to manage packet forwarding flows in DPDK vRouter.
- You will know how to install contrail and the traffic testing tools.
- You will be familiar with the utilities available for DPDK vRouter to troubleshoot and analyze DPDK performance.

How This Book Is Set Up

This book is organized into six chapters:

- Chapter 1 provides an SDN overview.
- Chapter 2 introduces virtualization concepts.
- Chapter 3 elaborates DPDK vRouter architecture in detail.
- Chapter 4 covers DPDK vRouter fine tuning.
- Chapter 5 reviews key Contrail and testing tool installations.
- Chapter 6 details DPDK vRouter tools and some lab studies.

NOTE This book replaces the term “slave” with the term “client.”

Chapter 1

SDN Review

Network Device Evolution

Since the early 1990s, network device manufacturers have worked on multiple innovation initiatives to increase switch and routing performance. They started from a router node in which everything was computed by the central CPU, and evolved into a situation where the central CPU is used less and less due to a distributed architecture in which high performance processing is handled by line cards, as can be seen in Figure 1.1.

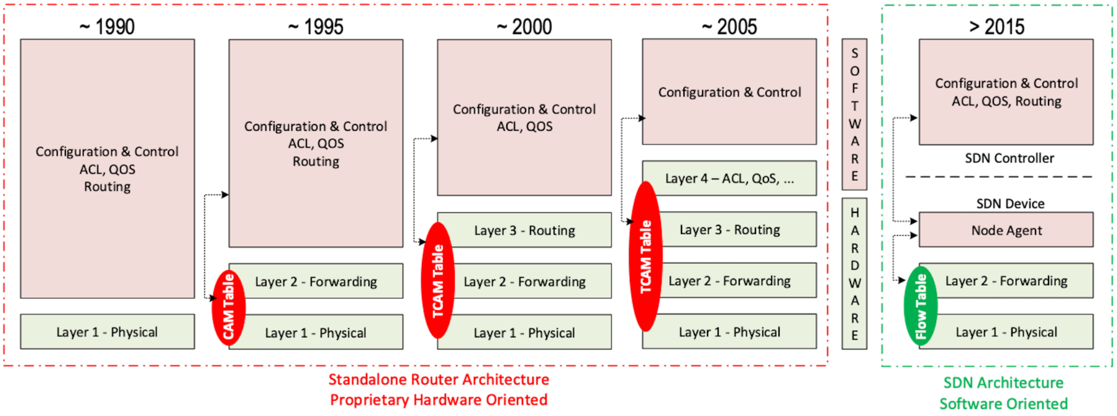


Figure 1.1 *Network Device Evolution*

This progress was possible thanks to the use of proprietary TCAM (Ternary Content-Addressable Memory) and ASICs (Application-Specific Integrated Circuits), which are designed to perform table look up and data packet forwarding at extremely high speeds.

In early 2000, the virtualization of x86 systems led to several innovations in systems domains. And evolution in compute virtualization and high-speed network devices has enabled network cloud creation.

Since it isn't convenient to manage several isolated network devices, each of which may have its own configuration language, the following needs have emerged:

- Single point of configuration
- Configuration protocol standardization
- Network feature support on x86 servers
- Extensibility and ability to scale
- Good performance

All of which calls for more cloud and SDN technology development.

The Early Age of SDN

The Stanford Clean Slate Project was initiated at Stanford University (US - CA) in 2006 in order to think about how to improve the internet's network architecture. The Ethane project was part of this program. Its purpose was to design networks where connectivity is governed by high-level, global policy. This project is generally known as the first implementation of SDN.

In 2008, a white paper was proposed by ACM (Association for Computing Machinery) to design a new protocol (OpenFlow) that can program network devices from a network controller. In 2011, the ONF (Open Networking Foundation) was created to promote SDN Architecture and OpenFlow protocols.

SDN Startups Acquired by Major Networks or Virtualization Vendors

The first companies focusing on SDN were founded around 2010. (Most of them have now been acquired by main networks or virtualization solution vendors.) In 2007, Martin Casado, who was working on the Ethane project at Stanford, founded Nicira to provide solutions for network virtualization with SDN concepts. Nicira was acquired by VMware in 2012 and became a key part of the VMWare NSX product. In 2016, VMware also bought PLUMGrid, a SDN startup founded in 2013. In 2010, Big Switch Networks, which was proposing an SDN solution, was founded. In early 2020, Big Switch was acquired by Arista Networks. In 2012, Cisco created Insieme Networks, a spin-off start-up company working on SDN. In 2013, Cisco took back control of Insieme in order to develop its own SDN solution called ACI (Application Centric Infrastructure). In early 2012, Contrail Systems was created and acquired at the end of the year by Juniper Networks. In 2013, Alcatel Lucent created Nuage Networks, a spin-off start-up company working on SDN, which is now an affiliate of Nokia.

The history of SDN development is not straightforward and is more nuanced than a single storyline suggests. It's far more complex than can be described in this short section. Figure 1.2 from [sdn-history] shows developments in programmable networking over the past 20 years, and their chronological relationship to advances in network virtualization.

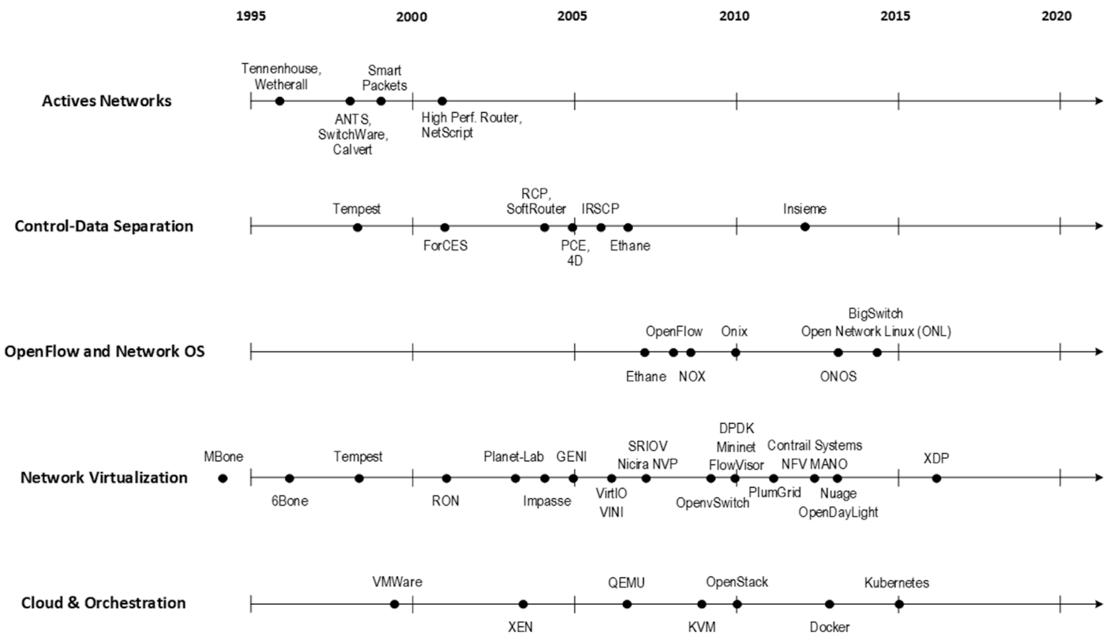


Figure 1.2 SDN History

So What is SDN?

Both the concept of SDN, and the term itself, are broad and often confusing. There is no truly accurate definition of SDN, and vendors usually explain it very differently. Initially it was used to describe Stanford's OpenFlow project, but today that definition has been extended to include a much wider swath of technologies. Discussion about each vendor's exact SDN definition is beyond the scope of this book, but in general, an SDN solution provides anywhere from one to several of the following characteristics:

- A network control and configuration plane split from the network data plane
- A centralized configuration and control plane (SDN controller)
- A simplified network node

- Network programmability to provide network automation
- Automatic provisioning ZTP (zero touch provisioning) of network nodes
- Virtualization support and openness

According to [onf-sdn-definition], software-defined networking (SDN) is the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices.

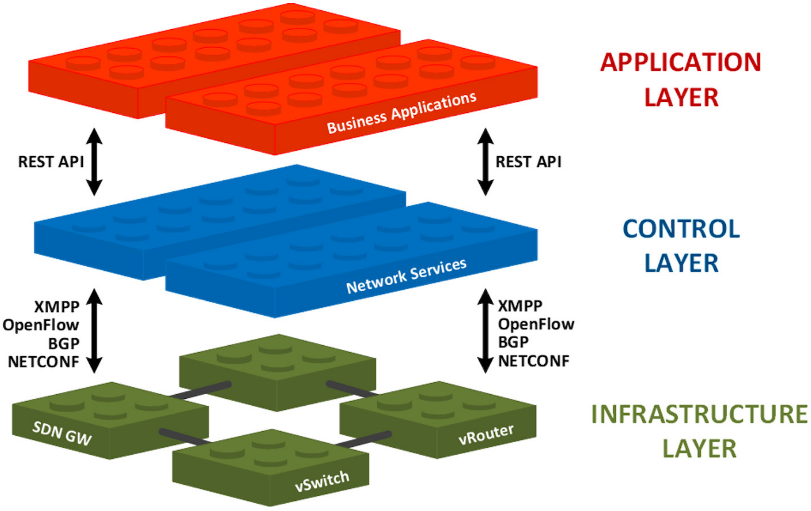


Figure 1.3 What is SDN?

You can see in Figure 1.3 that SDN allows simple high-level policies in the application layer to modify the network, because the device level dependency is eliminated to some extent. The network administrator can operate the different vendor-specific devices in the infrastructure layer from a single software console – the control layer. The controller in the control layer is designed in such a way that it can globally view the whole network. This controller design helps to introduce functionalities or programs, since the applications just need to talk to the centralized controller without needing to know all the details communicating with each individual device. These details are hidden by the controller from the applications.

Several traits fit this new model:

- Openness: Communication between controller and network device uses standardized protocols like REST, OpenFlow, XMPP, NetConf, gRPC, etc. This eliminates traditional vendor lock-in, giving you freedom of choice in networking.
- Cost reduction: Due to the open model, users can pick any low-cost vendor for their infrastructure (hardware).

- Automation: The controller layer has a global view of whole network. With the APIs exposed by the control layer, automation of applications becomes much easier.

In Figure 1.3, OpenFlow is labeled as the protocol between the control and infrastructure layers. This is just an example showing the use of standard communication protocols. Today more choices of communication protocols are available, and are the standard in the SDN industry, which is covered later in this chapter.

Traditional Network Planes and SDN Layer

Traditionally, a typical network device (for example, a router) has the following planes, as shown in Figure 1.4.

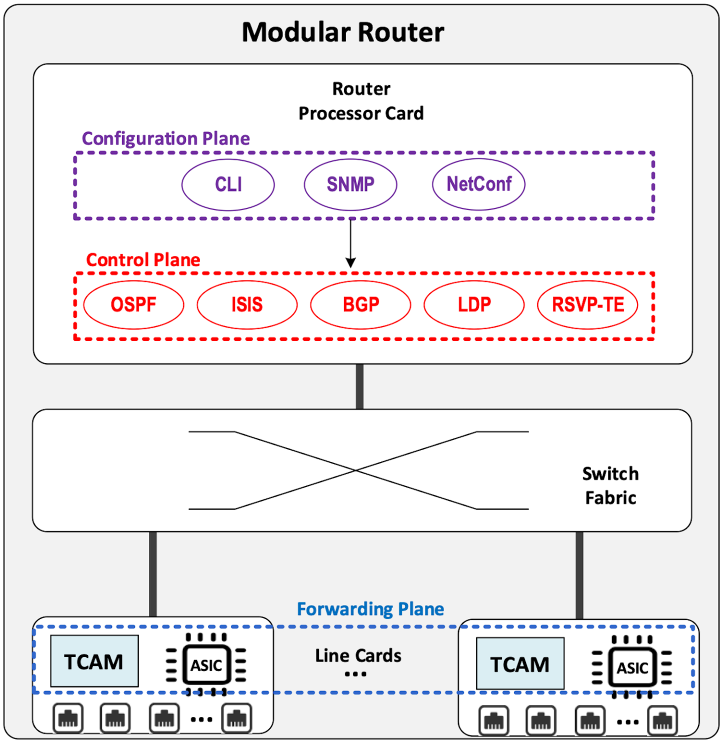


Figure 1.4 A Typical Modular Router

Figure 1.4 illustrates the following:

- Configuration (and management) plane: This is used for network node configuration and supervision. Examples of widely used schemes are CLI (command line interface), SNMP (Simple Network Management Protocol), and Netconf.

- Control plane: This is used by network nodes to make packet forwarding decisions. In traditional networks there has been a wide range of different network control protocols: OSPF, ISIS, BGP, LDP, RSVP-TE, etc.
- Forwarding (or data) plane: This plane is responsible for performing data packet processing and forwarding. This forwarding plane is made of proprietary protocols and is specific to each network equipment vendor.

Configuration and Control planes are located in the device's main processor card, often called the routing engine (RE) or routing switching engine. The forwarding plane is located in the device's packet forwarding card, often called the line card.

SDN architecture typically has three layers, as shown in Figure 1.5.

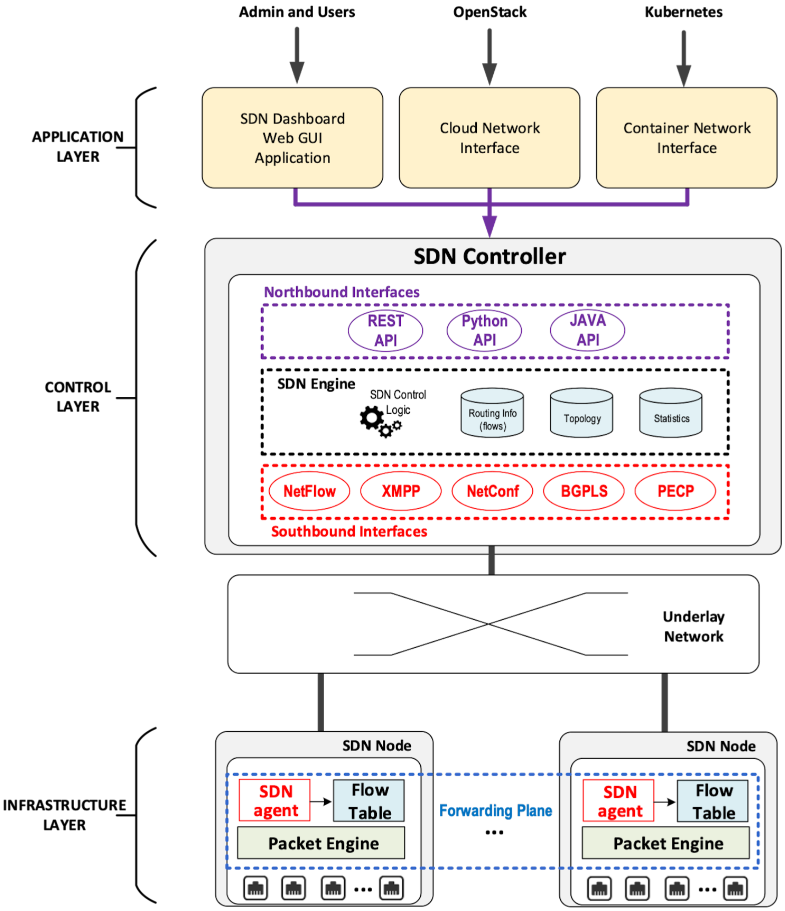


Figure 1.5 SDN Architecture

Figure 1.5 illustrates the following SDN architecture layers:

- Application Layer: This layer contains all the applications provided by the SDN solution. Generally, a Web GUI dashboard is the first application provided to SDN users. Other common applications are network infrastructure interconnection interfaces that allow the SDN solution to be plugged into a cloud infrastructure or a container orchestrator.
- Control Layer: The layer containing the SDN controller. This is the most intelligent part of a SDN solution and has a global view of the whole network. The SDN controller is made up of:

  - The SDN engine, which contains SDN control logic and databases to store the state and configuration of the network.

  - Southbound interfaces, which are used to communicate with the SDN network nodes. Some of the most commonly used southbound interface protocols are OpenFlow, XMPP, and OVSDB.

  - Northbound interfaces, which are used to expose services provided by the infrastructure layer 'upwards' to the SDN applications. The most commonly used northbound interface protocol is HTTP/REST.
- Infrastructure Layer: This layer consists of the SDN network nodes. This is the workhorse of a SDN solution, and SDN network nodes can be either physical or virtual. Typically, each SDN node is composed of:

  - A SDN agent that handles the communication between each SDN network node and the SDN controller

  - A flow/routing table built by the SDN Agent.

  - A forwarding plane engine

Primary Changes Between SDN and Traditional Networking

In a traditional infrastructure, the route calculation is made on each individual router. Each router needs to run one or several routing protocols through which it exchanges routes with the rest routers in the network, and eventually, based on the route information learned, each router builds a routing information base (RIB) in its routing engine and assumes it gains enough knowledge about the network in order to make the forwarding decision – most often in the form of a forwarding information base (FIB) that will be downloaded into the forwarding plane. From the network perspective, the control plane is distributed in each individual router, and the end-to-end routing path is the result of all decisions made by the control plane located on each router.

The control plane on a router might look like Figure 1.6.

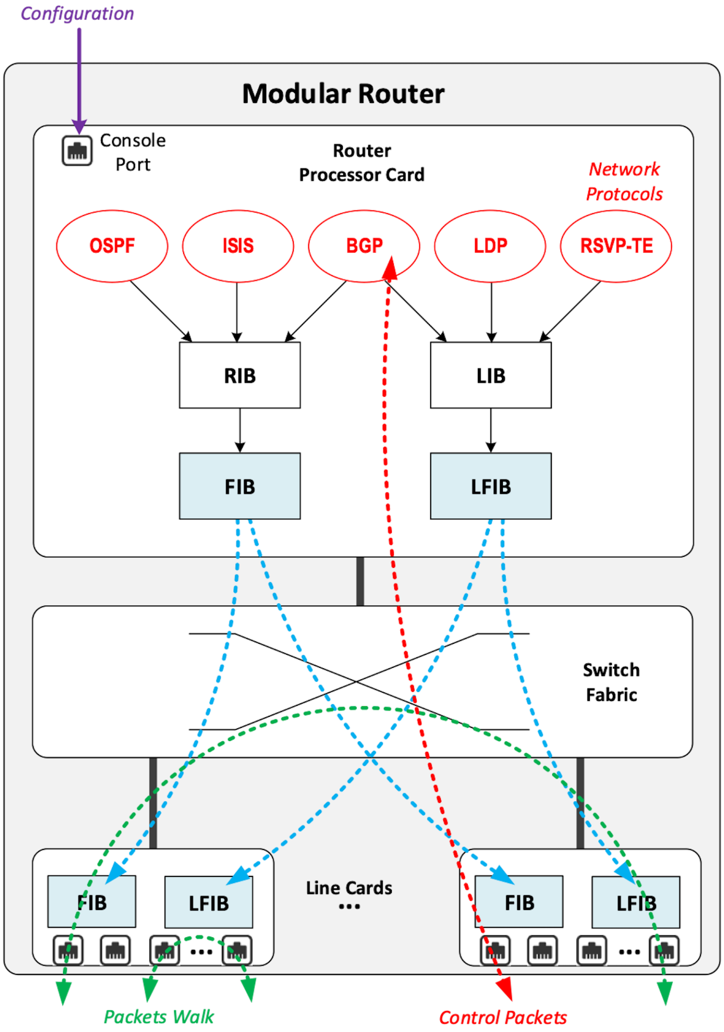


Figure 1.6 Components in a Traditional Router

NOTE In practice, routing and forwarding databases in a router are much more complicated. For example, whenever MPLS is involved there will also be a Label Information Base (LIB) and a label forwarding information base (LFIB), but we won't cover these details in this book.

For example, a simplified Juniper MX Series control plane typically looks like the one illustrated in Figure 1.7.

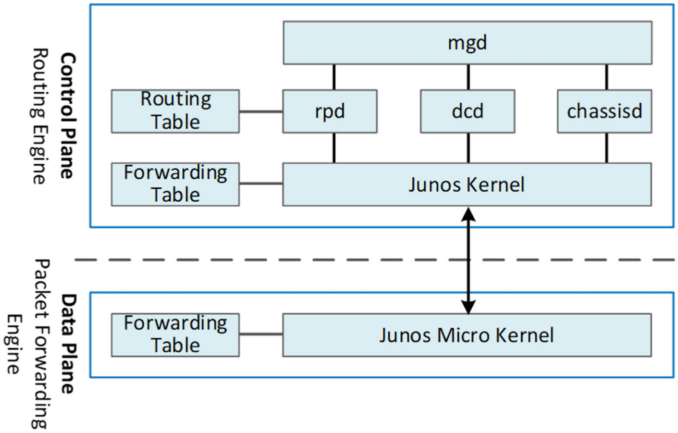


Figure 1.7 *Typical Control Plane in a Sample MX Series Router*

Running a control plane on each router is very hard to manage, because each individual network device needs to be carefully configured. Extensive, vendor-specific experiences and skills are required to configure each device. The high number of configuration points can make it challenging to build a robust network. Flexibility is also a recurring hurdle for traditional networks since most routers run proprietary hardware and software. In contrast, in SDN networking control and configuration functions are gathered into a SDN controller, which controls network devices. The new architecture is intended to provide a completely new way of configuring the network. This new cloud infrastructure brings:

- simplified routers, without complex control planes in each router.
- a centralized control plane, which is a single configuration point.

Let's compare the two architectures as shown in Figure 1.8.

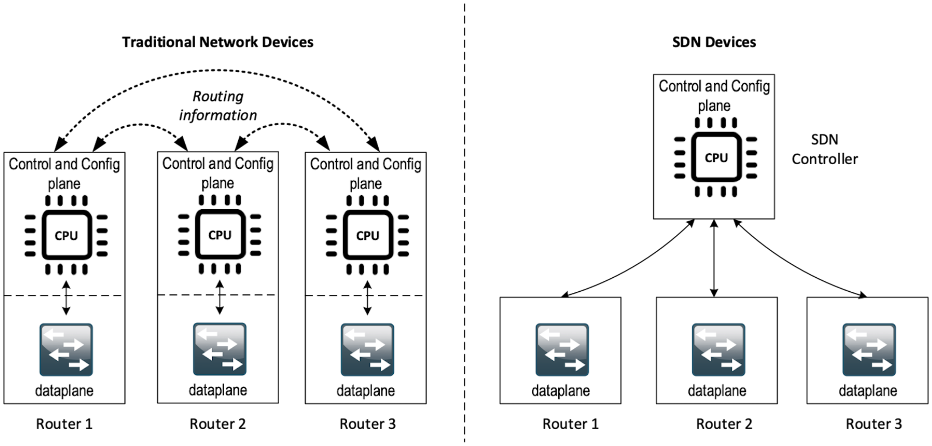


Figure 1.8 *Comparison Between Traditional Network Devices and SDN Devices*

You can see that the SDN infrastructure uses a centralized configuration and control point. Route calculation is done centrally in the controller and distributed into each SDN network node. While the idea looks simple, two fundamental protocols and infrastructures must be implemented before the model can work:

- A southbound network protocol: This is needed to allow routing information being exchanged between the SDN controller and each controlled element.
- A underlay network: This is a network infrastructure that allows the communication between SDN controller and SDN network nodes, and also the communication between SDN nodes themselves.

The underlay network infrastructure plays the same role as the local switch fabric inside a standalone router between the control processor card and lines cards. An overlay network based on it can be built by the controller, which basically hides underlay network infrastructure details from the applications so they focus on the high level service implementations. We'll discuss underlay and overlay more in the next section.

This model also makes the controller the weakest point. Think of what will happen if this SDN controller, serving as the brain, stops working. Everything freezes and nothing works as expected, or even worse, some part of the infrastructure continues to run but in an unexpected way, which will very likely trigger bigger issues in other parts of the network.

Each SDN solution supplier has put forth a lot of effort to solve this weakness. Using a clustered architecture to build a highly resilient controller cluster is a common and efficient practice. For example, three SDN controllers can load balance and/or backup each other. If one or two fails, the other one can still make the whole cluster survive, giving the operator longer maintenance windows to fix the problem.

Underlay Versus Overlay

In SDN architecture, each network node is connected to a physical network infrastructure. This physical network providing basic connectivity between network nodes is called the underlay network infrastructure. Sometimes it is also called fabric, and typically it's a plain L3 IP network.

Very often, the underlay needs to separate between different administrative domains (called tenants), switch within the same L2 broadcast domain, route between L2 broadcast domains, provide IP separation via VRFs, and more. This is implemented in the form of an overlay network. The overlay network is a logical network that runs on top of the underlay network. The overlay is formed with tunnels to carry the traffic across the L3 fabric.

Why are Overlay Networks Needed?

Today the industry is moving towards building L3 data centers and L3 infrastructures, mostly due to the rich features coming from L3 technologies such as ECMP load balancing, flooding control, etc. But the L2 traffic does not disappear and most likely it never will. There is always the desire that a group of network users will need to reside in the same L2 network – typically a VLAN. However, in today's virtualization environment a user's VM can be spawned in any compute located anywhere in the L3 cluster. Even if two VMs are spawned in the same server, there is often a need to move them around between different servers without changing their networking attributes. These requirements that a VM must always belong to the same VLAN call for an overlay model over the L3 network. In other words, you need a new mechanism to allow you to tunnel L2 Ethernet domains with different encapsulations over an L3 network.

For example, let's assume that in a SDN node, node1, you're running VM11 and VM12; both serve the same sales department and so are located in the same VLAN. Because of some administrative requirement, VM12 needs to be moved to another physical SDN, node2, which may be physically located in another rack that is a few route hops away. Now we need to ensure not only that the data packets from VM11 in SDN node1 can reach VM12 in SDN node2, but also that they are talking to each other as if they are still in the same VLAN, exactly the same way as before, and as if VM12 has never moved. This ability to make the local (in same VLAN) traffic traverse transparently across underlay network infrastructure calls for packet encapsulation, or tunneling mechanisms in SDN networks (see Figure 1.9).

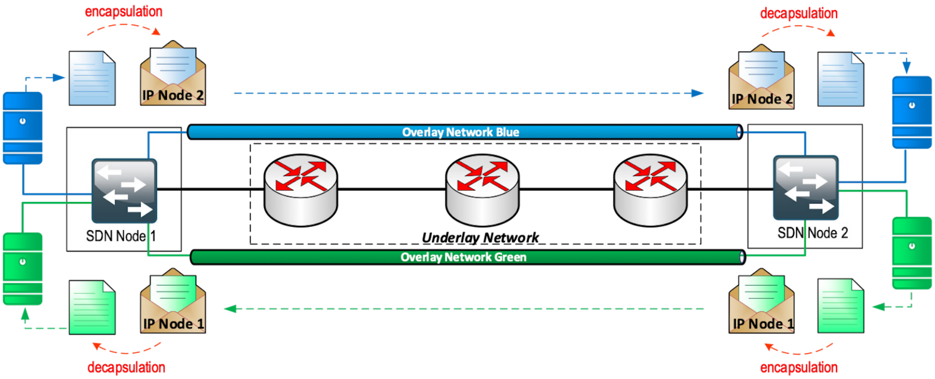


Figure 1.9 *Overlay Tunnels and Encapsulations*

Indeed, without such encapsulation mechanisms, traditional segmentation solutions (VLAN, VRF) would have to be provided by the physical infrastructure and implemented up to each SDN node in order to provide an isolated transportation channel for each customer network connected to the SDN infrastructure.

Tunneling protocols used in SDN networks have to provide at least the following capabilities:

- The ability to build connectivity for several different networks between two SDN network nodes. This is called network segmentation.
- The ability to transparently carry Ethernet frames and IP packets.
- The ability to be carried over IP connectivity.

Today, several tunneling protocols are used in SDN networks:

- VxLAN
- MPLS over GRE
- MPLS over UDP
- NVGRE
- Geneve

The tunneling protocols shown in Figure 1.10 provide Overlay connectivity, which is required between customer workloads connected to the SDN infrastructure.

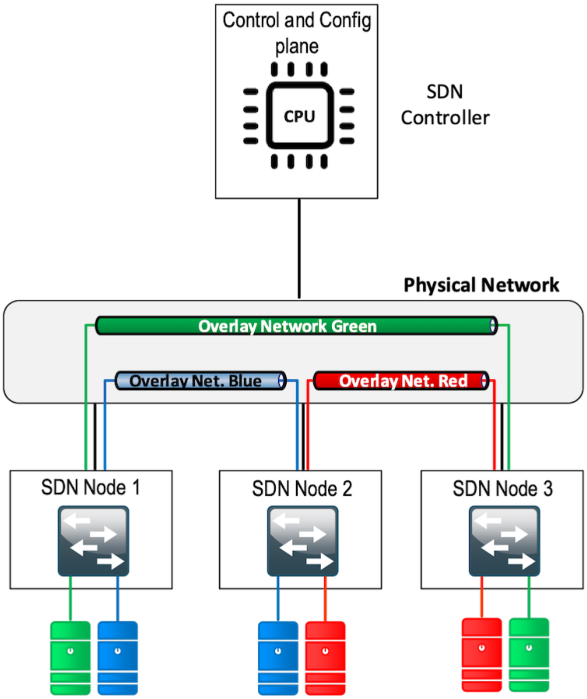


Figure 1.10 Overlay Networks Implemented by Tunneling Protocols

NOTE There are also other protocols not included in this list. They are either non standardized or not actively used in the industry. One such example is STT - Stateless Transport Tunneling. (https://www.ietf.org/archive/id/draft-davie-stt-08.txt).

TIP In VxLAN, specifically, each SDN node is called a VTEP (virtual tunnel end point) as it is starting and terminating the overlay tunnels.

Interfaces Between Layers

Let's elaborate on the concepts of southbound and northbound interface and some choices available in today's industry.

Southbound Interface

The southbound interface resides between the controller in the control layer and network devices in the infrastructure layer and provides a means of communication between the two layers. Based on demands and needs, a SDN Controller will dynamically change the configuration or routing information of network devices. For example, a new VM will advertise a new subnet or host routes when it is spawned in a server, and this advertisement will be delivered to an SDN controller via a southbound protocol. Accordingly, the SDN controller collects all the routing updates from the whole SDN cluster through the southbound interfaces, decides the most current and best route entries, and then it reflects this information to all the other network devices or VMs. This ensures all devices have the most up-to-date routing information in real time. Examples of the most well-known southbound interfaces in the industry are, among others, OpenFlow, OVSDB, gRPC and XMPP. OpenFlow and OVSDB are perhaps the most well-known southbound interfaces. We'll briefly introduce them.

OpenFlow

OpenFlow is a protocol that sends flow information into the virtual switch so the switch can forward the packets between the different ports. Flows are defined based on different criteria such as traffic between a source MAC address and a destination MAC address, source and destination IP addresses, TCP ports, VLANs, tunnels, and so on.

OpenFlow is one of the most widely deployed southbound standards from the open source community. Martin Casado at Stanford University first introduced it in 2008. The appearance of OpenFlow was one of the main factors that gave birth to SDN.

OpenFlow provides various information for the controller. It generates the event-based messages in case of port or link changes. The protocol generates a flow-based statistic for the forwarding network device and passes it to the controller.

OpenFlow also provides a rich set of protocol specifications for effective communication at the controller and switching element side. OpenFlow provides an open source platform for the research community.

Every physical or virtual OpenFlow-enabled network (data plane) device in the SDN domain needs to first register with the OpenFlow controller. The registration process is completed via an OpenFlow HELLO packet originating from the OpenFlow device to the SDN controller.

OVSDB

Unlike OpenFlow, Open vSwitch Database (OVSDB) is a southbound API designed to provide additional management or configuration capabilities like networking functions. With OVSDB you can create the virtual switch instances, set the interfaces and connect them to the switches. You can also provide the QoS policy for the interfaces. OVSDB sends and receives commands via JSON (JavaScript Object Notation) RPCs.

Northbound Interface

The northbound interface provides connectivity between the controller and the network applications running in the management plane. As already discussed, the southbound interface has different available protocols, while the northbound interface lacks such type of protocol standards. With the advancement of technology, however, we now have a wide range of northbound API support like ad-hoc APIs, RESTful APIs, etc. The selection of a northbound interface usually depends on the programming language used in application development.

More Alphabet Soup of Terms

With the development of virtualization, SDN technologies, and their ecology in recent years, more and more terms and changes in these terms, are emerging. A lot of confusion arises due to the context in which these terms are used. Sometimes the latest term the industry uses is a particular technology such as VNF or a concept such as NFV. Terms rise and fall out of favor as the industry evolves. In recent years, terms such as OpenStack, NVF and VNF have become the industry’s favorite buzzwords. This raises the question – just what are OpenStack, NVF, VNF, and what is the relationship of these things with SDN?

NFV: Networking Function Virtualization

NFV/VNF sounds like a new buzzword but it's been here for years, see Figure 1.11.

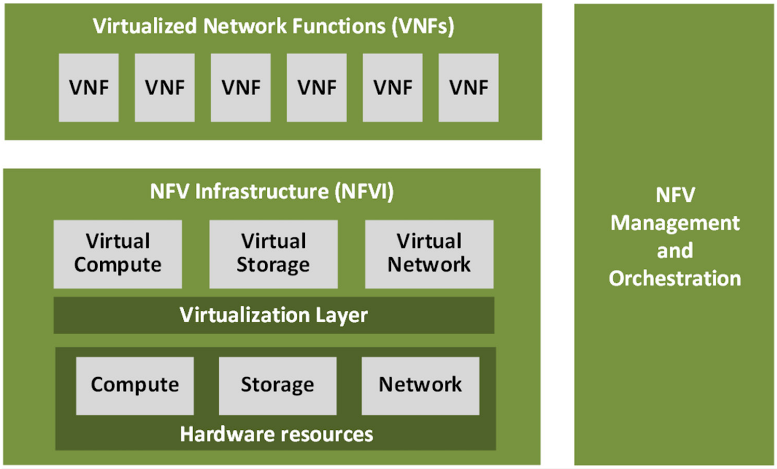


Figure 1.11 VNF/VNFI (Contrail/NFX) vs NFV (vSRX) vs NMO (CSO)

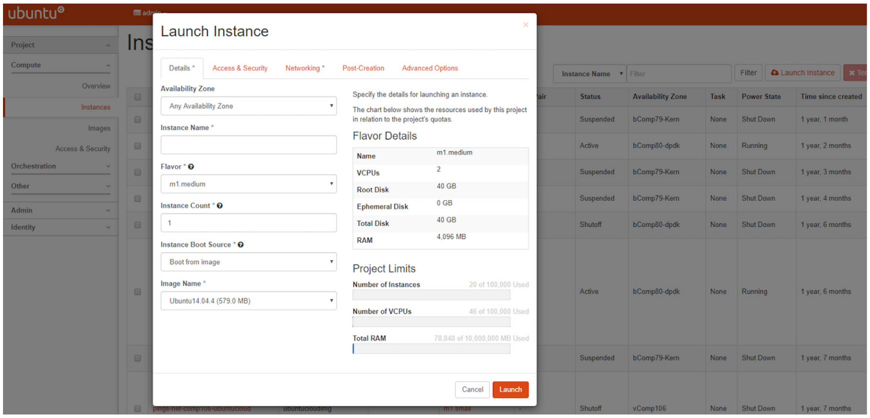
NFV stands for Network Function Virtualization, an operation framework for orchestrating and automating VNFs. VNF stands for virtual network function, such as virtualized routers, firewalls, load balancers, traffic optimizers, IDS or IPS, web application protectors, and so on.

In a nutshell, you can think of NFV as a concept or framework to virtualize certain network functions, while VNF is the implementation of each individual network function. Firewalls and load balancers are the two most common VNFs in the industry, especially for deployments inside data centers. When you read today's documents about virtualization technology, you will see the terms in a pattern like vXXX (e.g., vSRX, vMX), or cXXX (e.g., cSRX), where the letter "v" indicates it is a virtualized product, while the letter "c" means containerized or its container version.

OpenStack

Jointly launched by NASA and Rackspace in 2010, OpenStack has rapidly gained popularity in many enterprise data centers. It is one of the most used open source cloud computing platforms to support software development and big data analytics. OpenStack comprises a set of software modules such as compute, storage, and networking modules, which work together to provide an open source choice for building private and public cloud environments. As an IaaS (Infrastructure as a Service) open source implementation, it provides a wide range of services, from basic services like computing, storage, networking, and more, to advanced services like database, container orchestration, and others.

You can think of OpenStack as an abstraction layer providing a cloud environment on your promise. With OpenStack installed on your servers, you can spawn a VM, then consume and recycle it when you are done, all in seconds (see Figure 1.12). Under that abstraction layer, OpenStack hides most complexities of automation and orchestration of diverse underlying resources like compute, storage, and networking. You could choose servers, storage, and networking devices from your favorite vendors to build the underlying infrastructure, and OpenStack will consume all of them and expose them to the user as a pool of common resources like number of CPUs, RAMs, hard disk spaces, IP addresses, etc. The user does not (need to) care about vendor and brand details.



Launch Instance

Details * | Access & Security | Networking * | Post-Creation | Advanced Options

Availability Zone: Any Availability Zone

Instance Name *:

Flavor *: m1.medium

Instance Count *: 1

Instance Boot Source *: Boot from image

Image Name *: Ubuntu14.04.4 (579.0 MB)

Specify the details for launching an instance.

The chart below shows the resources used by this project in relation to the project's quotas.

Flavor Details

Name	m1.medium
VCPUs	2
Root Disk	40 GB
Ephemeral Disk	0 GB
Total Disk	40 GB
RAM	4,096 MB

Project Limits

Number of Instances 20 of 100,000 Used

Number of VCPUs 46 of 100,000 Used

Total RAM 78,848 of 10,000,000 MB Used

Cancel Launch

Status	Availability Zone	Task	Power State	Time since created
Suspended	bComp79-Kern	None	Shut Down	1 year, 1 month
Active	bComp80-dpdk	None	Running	1 year, 2 months
Suspended	bComp79-Kern	None	Shut Down	1 year, 3 months
Suspended	bComp79-Kern	None	Shut Down	1 year, 4 months
Shutoff	bComp80-dpdk	None	Shut Down	1 year, 6 months
Active	bComp80-dpdk	None	Running	1 year, 6 months
Suspended	bComp79-Kern	None	Shut Down	1 year, 7 months
Shutoff	vComp106	None	Shut Down	1 year, 7 months

Figure 1.12 *OpenStack Launching a New Instance*

If you compare OpenStack with SDN, it's easy to see that the two models share some common features. Both provide a certain level of abstraction, hiding the low-level hardware details and exposing upper level user applications. The differences are somewhat subtle to describe in just a few words. First off, although there are various distributions from different vendors, they share common core components managed by the OpenStack Foundation. SDN is more of a framework or an approach to manage the network dynamically, which can be implemented with totally different software techniques.

Second, from the perspective of technical ecological coverage, the ecological aspects of OpenStack are much wider because networking, along with various other plugins, is just one of the services implemented by its Neutron component. In contrast, SDN and its ecology focus mainly on the networking. There are also differences in the way that Neutron works compared to how a typical SDN controller works. OpenStack Neutron focuses on providing network services for virtual machines, containers, physical servers, etc. and provides a unified northbound REST API to users. SDN focuses on configuration and management of forwarding control toward the underlaying network device. It not only provides user-oriented northbound API, it also provides standard southbound APIs for communicating with various hardware devices.

The comparison between OpenStack and SDN here is conceptual. In reality these two models can, and in fact are, often coupled with each other in some way, loosely or tightly. One example is Tungsten Fabric(TF), which we'll talk about later in this chapter.

SDN Solutions Overview

Controllers

As mentioned previously, SDN is a networking solution that changes the traditional network architecture by bringing all control functionalities to a single location and making centralized decisions. In this solution, SDN controllers are the brain, which performs the control decision tasks while routing the packets. Centralized decision capability for routing enhances the network performance. As a result, an SDN controller is the core component of any SDN solutions.

While working with SDN architecture, one of the major points of concern is which controller and solution should be selected for deployment. There are quite a few SDN controller and solution implementations from various vendors, and every solution has its own pros and cons, along with its working domain. In this section we'll review some of the popular SDN controllers in the market, and the corresponding SDN solutions.

OpenDaylight (ODL)

OpenDaylight, often abbreviated as ODL, is a Java-based open source project started in 2013. It was originally led by IBM and Cisco but later hosted under the Linux Foundation. It was the first open source controller that could support non-OpenFlow southbound protocols, which made it much easier to integrate with multiple vendors.

ODL is a modular platform for SDN. It is not a single piece of software. It is a modular platform for integrating multiple plugins and modules under one umbrella. There are many plugins and modules built for OpenDaylight. Some are in production, while some are still under development.

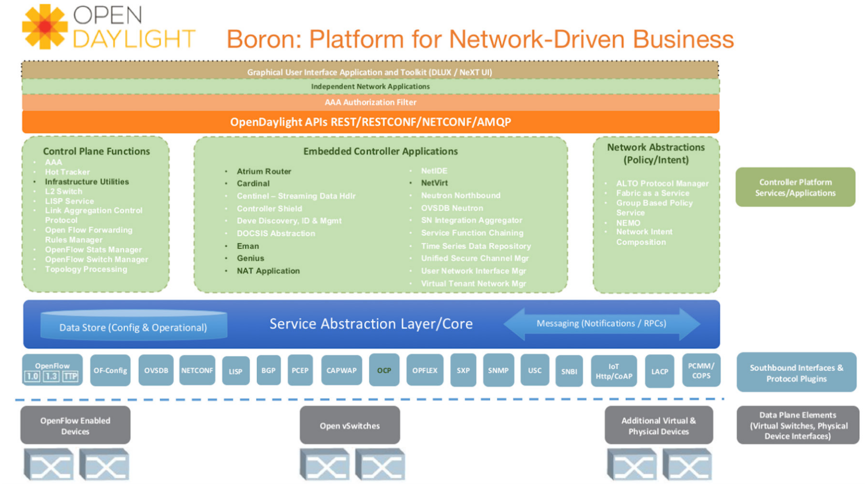

OPEN DAYLIGHT
Boron: Platform for Network-Driven Business
Graphical User Interface Application and Toolkit (DLUX / NeXT UI)
Independent Network Applications
AAA Authorization Filter
OpenDaylight APIs REST/RESTCONF/NETCONF/AMQP
Control Plane Functions
AAA
Hot Tracker
Infrastructure Utilities
L2 Switch
LISP Service
Link Aggregation Control Protocol
Open Flow Forwarding Rules Manager
OpenFlow Stats Manager
OpenFlow Switch Manager
Topology Processing
Embedded Controller Applications
Atrium Router
Cardinal
Centinel – Streaming Data Hdlr
Controller Shield
Deve Discovery, ID & Mgmt
DOCSIS Abstraction
Eman
Genius
NAT Application
NetIDE
NetVirt
Neutron Northbound
OVSDB Neutron
SN Integration Aggregator
Service Function Chaining
Time Series Data Repository
Unified Secure Channel Mgr
User Network Interface Mgr
Virtual Tenant Network Mgr
Network Abstractions (Policy/Intent)
ALTO Protocol Manager
Fabric as a Service
Group Based Policy Service
NEMO
Network Intent Composition
Controller Platform Services/Applications
Data Store (Config & Operational)
Service Abstraction Layer/Core
Messaging (Notifications / RPCs)
OpenFlow 1.0 1.3 TTP
OF-Config
OVSDB
NETCONF
LISP
BGP
PCEP
CAPWAP
OCP
OPFLEX
SXP
SNMP
USC
SNBI
IoT Http/CoAP
LACP
PCMM/ COPS
Southbound Interfaces & Protocol Plugins
OpenFlow Enabled Devices
Open vSwitches
Additional Virtual & Physical Devices
Data Plane Elements (Virtual Switches, Physical Device Interfaces)

Figure 1.13 *OpenDaylight Boron*

Some of the initial SDN controllers had their southbound APIs tightly bound to OpenFlow, but as you can see from Figure 1.13, besides OpenFlow, many other southbound protocols available in today's market are also supported. Examples are NETCONF, OVSDB, SNMP, BGP, and more. Support for these protocols is done in a modular method in the form of different plugins, which are linked dynamically to a central component named the Service Abstraction Layer (SAL). SAL does translations between the SDN application and the underlaying network equipment. For instance, when it receives a service request from an SDN application, typically via high level API calls (northbound), it understands the API call and translates the request to a language that the underlying network equipment can also understand. That language is one of the southbound protocols.

While this translation is transparent to the SDN application, ODL itself needs to know all the details about how to talk to each one of the network devices it supports, their features, capabilities, etc. A topology manager module in ODL manages this type of information. It collects topology related information from various modules and protocols, such as ARP, host tracker, device manager, switch manager, OpenFlow, etc., and based on this information, it visualizes the network topology by dynamically drawing a diagram showing all the managed devices and how they are connected together (see Figures 1.14 and 1.15).

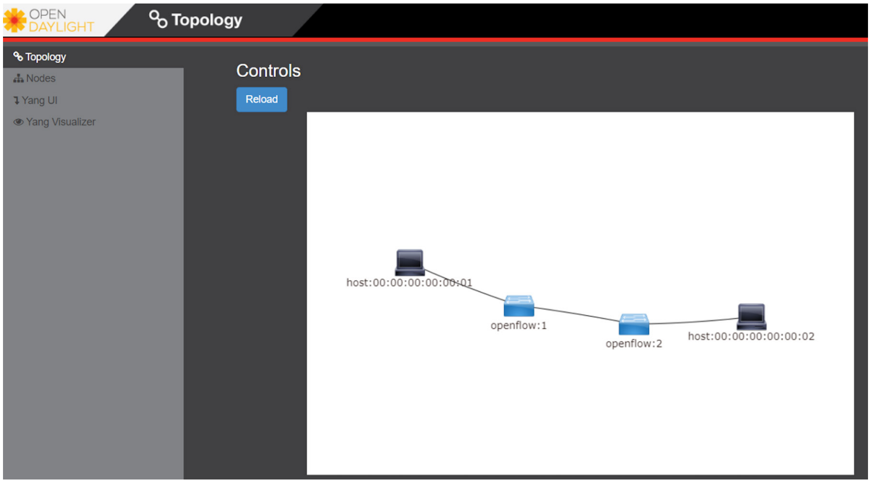


Figure 1.14 ODL Topology

Any topology changes, such as adding new devices, will be updated in the database and reflected immediately in the diagram.

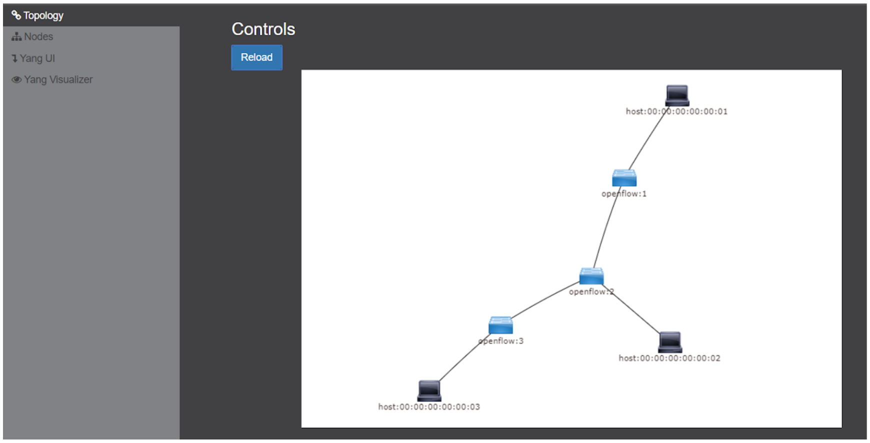


Figure 1.15 ODL Topology Update

As an SDN controller, ODL has a global view of the whole network, therefore it has all the necessary visibility and knowledge of the network that can be used to draw the network diagram in real time.

Open vSwitch (OVS)

OVS is one of the most popular and production quality open source implementations of a multilayer virtual switch. It was initiated in 2009 by Nicira, which was acquired by VMware in 2012. It is licensed under the Apache 2.0 license and provided by Linux Foundation. The virtual switch completes most of the jobs you'd expect a physical switch to, but via a software method. OVS is typically run with Linux hypervisors like KVM and can be loaded on a Linux kernel. OVS supports most features supported in traditional physical switches, such as:

- 802.1Q and VLAN
- BFD
- NetFlow/sFlow
- port mirroring
- LACP
- VXLAN
- GENEVE GRE Overlays
- STP
- IPv6

Besides the functions of traditional switches, the bigger advantage of OVS is that it also has native support to the SDN solution via OVSDB and OpenFlow protocols. That means any SDN controller can integrate OVS via these two open standard protocols. Therefore OVS can work either as a standalone L2 switch within a hypervisor host, or it can be managed and programmed via an SDN controller, such as ODL. That is why it is used in so many open source and commercial virtualization projects.

Calico

Here is a quote from the official Calico website:

Calico is an open source networking and network security solution for containers, virtual machines, and native host-based workloads. Calico supports a broad range of platforms including Kubernetes, OpenShift, Docker EE, OpenStack, and bare metal services.

Calico has been an open-source project from day one. It was originally designed for today's modern cloud-native world and runs on both public and private clouds. Its reputation derives from its deployment in Kubernetes and its ecosystem environments. Today Calico has become one of the most used Kubernetes Container Network Interfaces (CNI) and many enterprises are using it at scale.

Compared with other overlay network SDN solutions, Calico is special in the sense that it does not use any overlay networking design or tunneling protocols, nor does it require NAT. Instead it uses a plain IP networking fabric to enable host-to-host and pod-to-pod networking. The basic idea is to provide Layer 3 networking capabilities and associate a virtual router with each node, so that each node behaves like a traditional router or a virtual router. We know that a typical internet router relies on routing protocols like OSPF or BGP to learn and advertise the routing information, and that is the way a node in calico networking works. It chooses BGP as its routing protocol because of its simplicity, the industry's current best practice, and the only protocol that sufficiently scales.

Calico uses a policy engine to deliver high-level network policy management.

VCP (Nuage)

The SDN platform offered by Nuage Networks (now Nokia) is the Virtualized Cloud Platform (VCP). It provides a policy-based SDN platform that has a data plane built on top of the open source OVS, and a closed source SDN controller.

The Nuage platform uses overlays to provide policy-based networking between different clouding environment (Kubernetes pods or non-Kubernetes environments such as VMs and bare metal servers). It also has a real-time analytics engine to monitor Kubernetes applications.

All components can be installed in containers. There are no special hardware requirements.

Overview of Tungsten Fabric (TF)

Tungsten Fabric (TF) is an open-standard based, proactive overlay SDN solution. It works with existing physical network devices and helps address networking challenges for self-service, automated, and vertically integrated cloud architecture. It also improves scalability through a proactive overlay virtual network technique.

The TF controller integrates with most popular cloud management systems such as OpenStack, VMWare, and Kubernetes. TF provides networking connectivity and functionalities and enforces user-defined network and security policies to the various workloads based on different platforms and orchestrators.

One other major advantage of TF is that it is multi-cloud and multi-stack. It is made up of open standards for easier interoperability with other networking hardware like routers or switches. Today it supports:

- Multiple compute types - BareMetal, VMs and containers
- Multiple cloud stack types - VMware, OpenStack, Kubernetes (via CNI), OpenShift
- Multiple performance modes - Kernel native, DPDK accelerated, and several SmartNICs from different vendors
- Multiple overlay models - VxLAN, MPLSoUDP, MPLSoGRE tunnels or direct, non-overlay mode (no tunneling)

TF fits seamlessly into Linux Foundation Networking's (LFN) mission to foster open source innovation in the networking space. The TF system is implemented as a set of nodes running on general-purpose x86 servers. Each node can be implemented as a separate physical server or VM.

Open Source Version

Initially Contrail was a product of the startup company Contrail Systems, which was acquired by Juniper Networks in December 2012. It was open sourced in 2013 with a new name OpenContrail under the Apache 2.0 license, which means anyone can use and modify the code of the OpenContrail system without any obligation to publish or release the modifications. In early 2018, it was rebranded to Tungsten Fabric (TF) as it transitioned into a fully-fledged Linux Foundation project. Currently TF is still managed by the Linux Foundation.

Commercial Version

Juniper also maintains a commercial version of Contrail, and provides commercial support to licensed users. Both the open source and commercial versions of Contrail provide the same full functionalities, features, and performances.

NOTE As a reminder, we use these terms Contrail, OpenContrail, Tungsten Fabric, and TF interchangeably throughout this book.

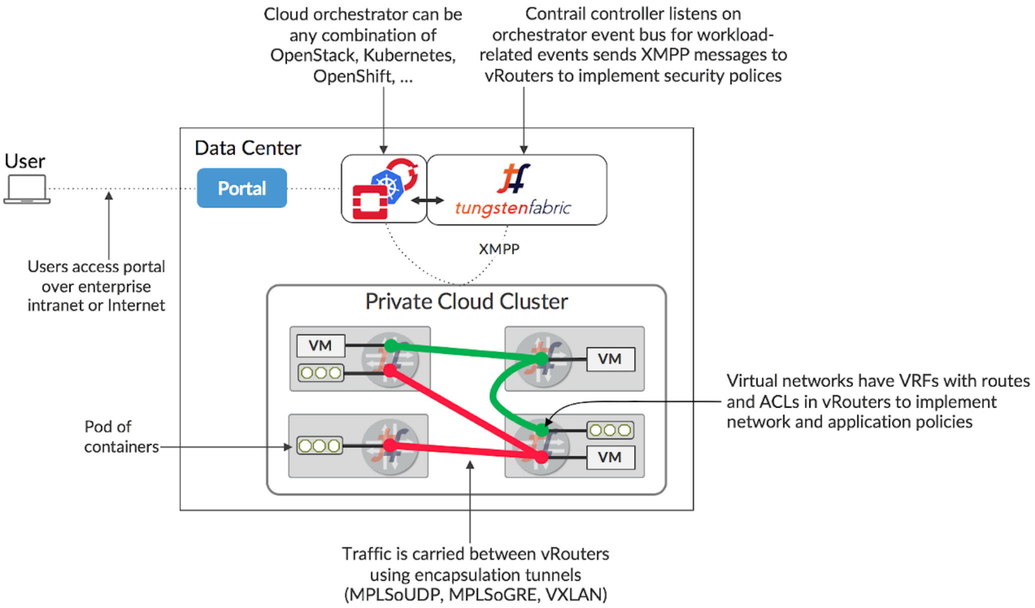
TF Architecture

TF consists of two main components:

- Tungsten Fabric Controller: This is the SDN controller in the SDN architecture.
- Tungsten Fabric vRouter: This is the forwarding plane that runs in each compute node performing packet forwarding and enforcing network and security policies.

The communication between the controller and vRouters is via XMPP, a widely used messaging protocol.

A high-level Tungsten Fabric architecture is shown in Figure 1.16.



Cloud orchestrator can be any combination of OpenStack, Kubernetes, OpenShift, ...

Contrail controller listens on orchestrator event bus for workload-related events sends XMPP messages to vRouters to implement security polices

Users access portal over enterprise intranet or Internet

Pod of containers

Virtual networks have VRFs with routes and ACLs in vRouters to implement network and application policies

Traffic is carried between vRouters using encapsulation tunnels (MPLSoUDP, MPLSoGRE, VXLAN)

Figure 1.16 Tungsten Fabric Architecture

The TF SDN Controller Node

The TF SDN controller integrates with an orchestrator's networking module in the form of a plugin, for instance:

- In OpenStack environment, TF interfaces with the Neutron server as a Neutron plugin.
- In Kubernetes environment, TF interfaces with k8s API server as a kube-network-manager process and a CNI plugin that is watching the events from the k8s API.

TF SDN Controller is described as a logically centralized but physically distributed SDN controller. It is physically distributed because the same exact controllers can be running in multiple (typically three) nodes in a cluster for high availability (HA) purpose. However, all controllers work together to behave as a single logical unit that is responsible for providing the management, control, and analytics functions of the whole cluster.

This physically-distributed nature of the Contrail SDN Controller is a distinguishing feature because there can be multiple redundant instances of the controller operating in an active/active mode (as opposed to an active-standby mode). When everything works, two controllers can share the workload and load balance the control tasks. When a node becomes overloaded, additional instances of that node type can be instantiated, after which the load is au tomatically redistributed. On the failure of any active node, the system as a whole can continue to operate without any interruption. This prevents any single node from becoming a bottleneck and allows the system to manage a very large scale system. In production, a typical high-availability deployment is to run three controller nodes in an active-active mode, as single point failure is eliminated.

As with any SDN controller, the TF controller has a global view of all routes in the cluster. It implements this by collecting the route information from all computes (where the TF vRouters reside) and distributes this information throughout the cluster.

TF vRouter: Compute Node

Compute nodes are general-purpose virtualized servers that host VMs. These VMs can be tenants running general applications, or service VMs running network services such as a virtual load balancer or virtual firewall. Each compute node contains a TF vRouter that implements the forwarding plane.

The TF vRouter is conceptually similar to other existing virtualized switches such as the Open vSwitch (OVS), but it also provides routing and higher layer services. It replaces traditional Linux bridge and IP tables, or Open vSwitch networking on the compute hosts. Configured by the TF controller, the TF vRouter implements the desired networking and security policies. While workloads in same network can communicate with each other by default, an explicit network policy is required to communicate with VMs in different networks.

As with other overlay SDN solutions, TF vRouter extends the network from the physical routers and switches in a data center into a virtual overlay network hosted in the virtualized servers. Overlay tunnels are established between all computes, and communication between VMs on different nodes is carried out in these tunnels and behaves as if they are on the same compute. Currently VxLAN, MPLSoUDP, and MPLSoGRE tunnels are supported.

TF Controller Components

In each TF SDN Controller there are three main components, as shown in Figure 1.17:

- Configuration nodes - These nodes keep a persistent copy of the intended configuration states and store them in a Cassandra database. They are also responsible for translating the high-level data model into a lower-level form suitable for interacting with control nodes.

- Control nodes - These nodes are responsible for propagating the low-level state data it received from configuration node to the network devices and peer systems in an eventually consistent way. They implement a logically centralized control plane that is responsible for maintaining the network state. Control nodes run XMPP with network devices, and run BGP with each other.

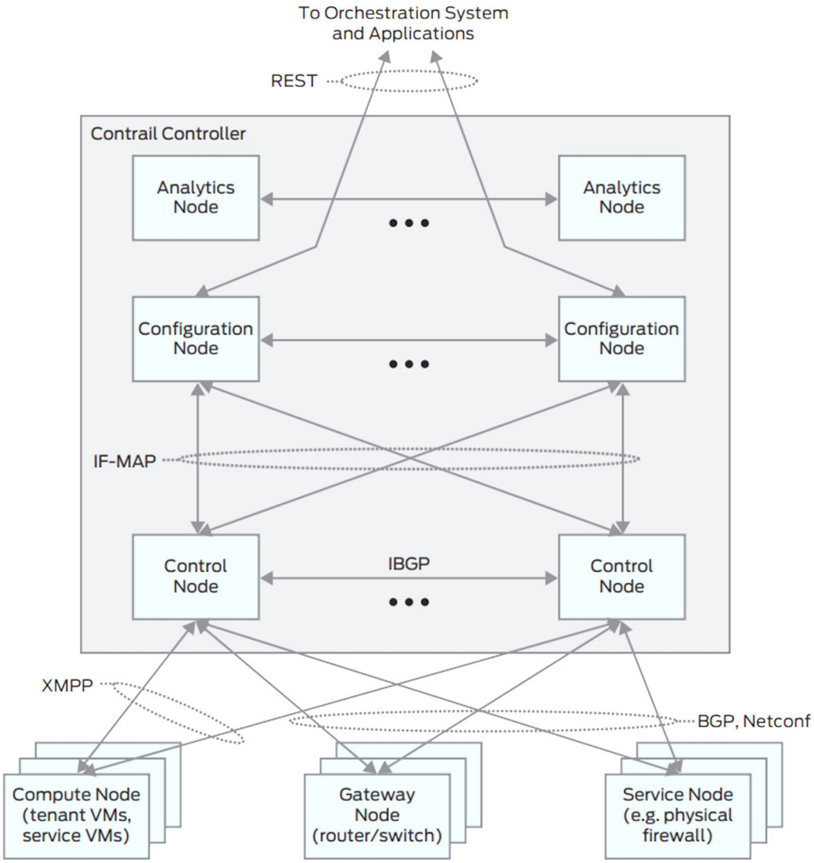


Figure 1.17 Tungsten Fabric Controller Components

- Analytics nodes - These nodes are mostly about statistics and logging. They are responsible for capturing real-time data from network elements, abstracting it, and presenting it in a form suitable for applications to consume. It collects, stores, correlates, and analyzes information from network elements.

TF vRouter Components

The TF vRouter is the main forwarding module running in each compute node. The compute node is a general-purpose x86 server that hosts tenant VMs running customer applications.

The TF vRouter consists of two components:

- The vRouter agent, which is the local control plane
- The vRouter forwarding plane

In a typical configuration, Linux is the host OS and KVM is the hypervisor. The Contrail vRouter forwarding plane can sit either in the Linux kernel space or in the user space while running on DPDK mode. More details about this will be covered in later chapters in this book.

The vRouter agent is a user space process running inside Linux. It acts as the local, lightweight control plane in the compute, in a way similar to what a routing engine does in a physical router (see Figure 1.18). For example, vRouter agents establish XMPP neighborships with two controller nodes, then exchanges the routing information with them. The vRouter agent also dynamically generates flow entries and injects them into the vRouter forwarding plane. This gives instructions to the vRouter about how to forward packets.

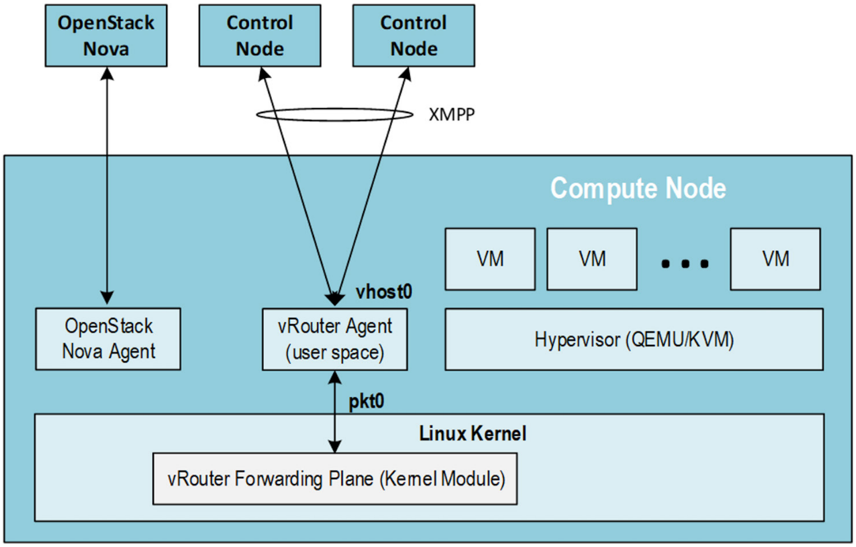


Figure 1.18 vRouter Agent

The vRouter forwarding plane works like a line card of a traditional router (see Figure 1.19). It looks up its local FIB and determines the next hop of a packet. It also encapsulates packets properly before sending them to the underlay network and decapsulates packets to be received from the underlay network.

We'll cover more details of TF vRouter in later chapters.

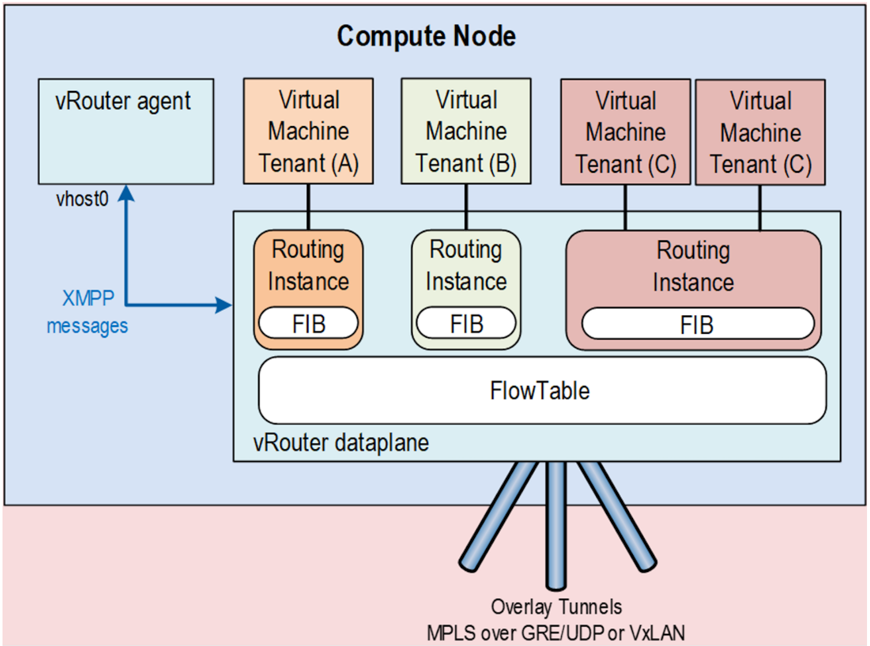


Figure 1.19 vRouter Forwarding Plane

SDN References

This has been a whirlwind tour of SDN so here are some additional references you may find useful:

- https://www.cs.princeton.edu/courses/archive/fall13/cos597E/papers/sdnhistory.pdf
- https://www.opennetworking.org/sdn-definition/
- https://www.openvswitch.org/

Chapter 2

Virtualization Concepts

Server Virtualization

Kernel-based virtual machine (KVM) is an open source virtualization technology built into Linux. It provides hardware assistance to the virtualization software, using built-in CPU virtualization technology to reduce virtualization overheads (cache, I/O, memory) and improve security.

QEMU is a hosted virtual machine emulator that provides a set of different hardware and device models for the guest machine. For the host, QEMU appears as a regular process with its own process memory scheduled by the standard Linux scheduler. In the process, QEMU allocates a memory region that the guest sees as physical and executes the virtual machine's CPU instructions.

With KVM, QEMU can create a virtual machine that the processor is aware of and that runs native-speed instructions with just virtual CPUs (vCPUs). When a special instruction—like the ones that interact with the devices or special memory regions—is reached by KVM, vCPU pauses and informs QEMU of the cause of pause, allowing the hypervisor to react to that event.

Libvirt is an open-source toolkit that allows you to manage virtual machines and other virtualization functionality, such as storage and network interface management (see Figure 2.1). It proposes to define virtual components in XML-formatted configurations that are able to be translated into the QEMU command line.

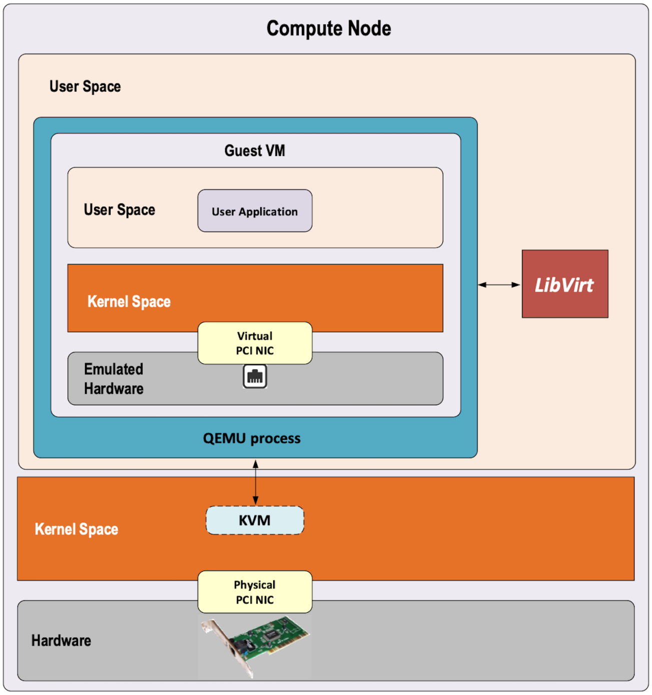


Figure 2.1 LibVert Compute Node

Interprocess Communication

Interprocess communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions. The communication between these processes can be considered as a method of cooperation between them.

IPC is used in network virtualization in order to exchange data between different distributed processes of a same application (for example, Virtio frontend and backend, Contrail vRouter agent and data plane, etc.) or between processes of distinct applications (e.g., contrail vRouter and QEMU Virtio, Virtio and VFIO, and so on).

Two different modes of communication are used for IPC:

- Shared Memory: processes read and write information into a shared memory region.
- Message Passing: processes establish a communication link that will be used to exchange messages.

Shared Memory

The following scenario is used when shared memory is used for IPC:

- First, a shared memory area is defined (shmget) with a key identifier known by processes involved in the communication.
- Second, processes attach (shmat) to the shared memory and retrieve a memory pointer.
- Then, processes read or write information in the shared memory using the shared memory pointer (read/write operation).
- Next, processes detach from the shared memory (shmdt)
- Last, the shared memory area is freed (shmctl)

The following system calls are used in shared memory IPC:

- shmget: create the shared memory segment or use an already created shared memory segment.
- shmat: attach the process to the already created shared memory segment.
- shmdt: detach the process from the already attached shared memory segment.
- shmctl: control operations on the shared memory segment (set permissions, collect information).

Message Passing

Several message passing methods are available to exchange data information between processes:

- eventfd: is a system call that creates an eventfd object (64-bit integer). It can be used as an event wait/notify mechanism by user-space applications, and by the kernel to notify user-space applications of events.
- pipe (and named pipe) is a unidirectional data channel. Data written to the write-end of the pipe is buffered by the operating system until it is read from the read-end of the pipe.
- Unix Domain Socket: domain sockets use the file system as their address space. Processes reference a domain socket as an inode, and multiple processes can communicate using a same socket. The server of the communication binds a UNIX socket to a path in the file system, so a client can connect to it using that path.

There are some other mechanisms that can be used by processes to exchange messages (shared file, message queues, network sockets, and signals system calls) but they are not described in this book.

Network Device Architecture and Concepts

Control and Data Paths

Two different flows are used by a network application using a NIC device (Figure 2.2):

- Control: manages configuration changes (activation/deactivation) and capability negotiation (speed, duplex, buffer size) between the NIC and network application for establishing and terminating the data path on which data packets will be transferred.
- Data: performs data packets transfer between NIC and network application. Packets are transferred from NIC internal buffer to a host memory area that is reachable by the network application.

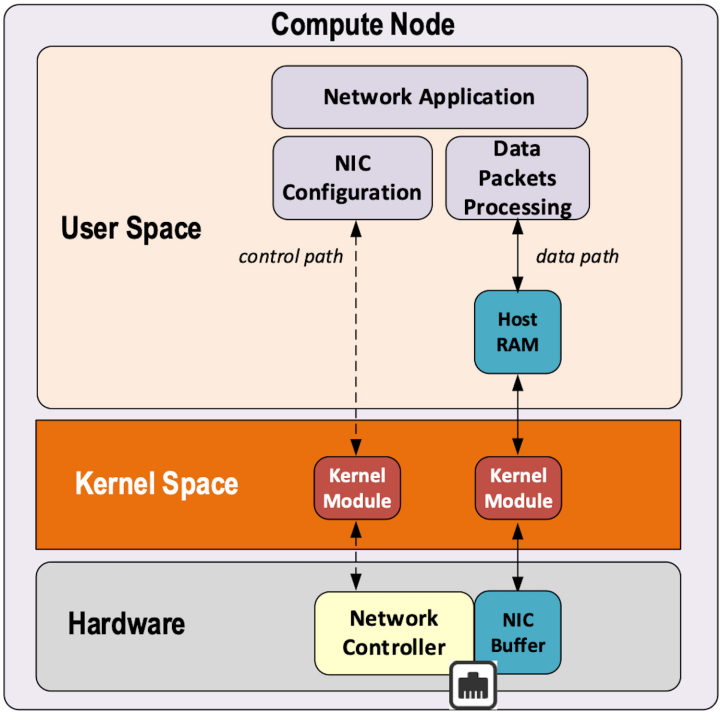


Figure 2.2 *Two Different Flows Using a NIC Device*

Each flow is using a well-defined path: a control path and a data path.

Event Versus Polling-based Packet Processing

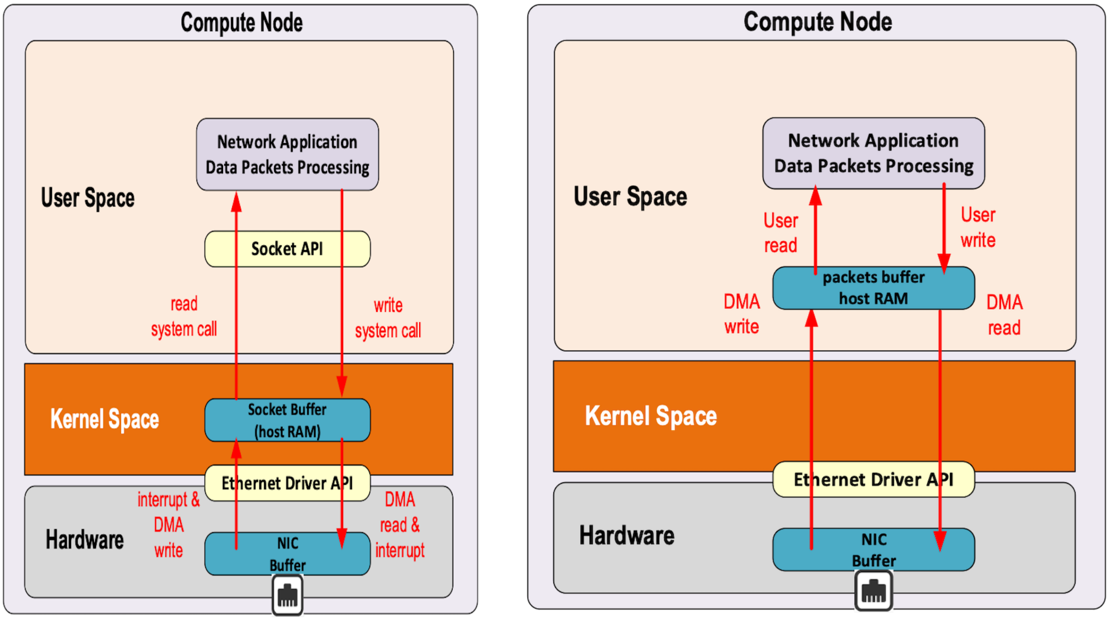
The Linux network stack (Figure 2.3) uses an event-based packet processing method. In such a method every incoming packet hitting the NIC:

- is copied in host memory via DMA,
- then the NIC generates an interrupt,
- then a Kernel module places the packet into a socket buffer,
- and the application runs a read system call.

For every egress packet generated by the network application:

- The application performs a write call on the socket in order to copy the generated packet from the application user space to a socket buffer.
- The kernel device driver invokes the NIC DMA engine to transmit the frame onto the wire.
- Once the transmission is complete, the NIC raises an interrupt to signal transmit completion in order to get socket buffer memory freed.

This method is not efficient when packets are hitting the NIC at a high packet rate. Lots of interrupts are generated, creating lots of context switching (kernel to user and vice-versa).



Event-based Packet Processing

Polling-based Packet Processing

Figure 2.3 *Event-based Versus Polling-based*

Polling based packet processing is an alternate method (it is used by DPDK). All incoming packets are copied transparently (without generating any interrupt) by the NIC into a specific host memory area region (predefined by the application). At a regular pacing, the network application is reading (polling) packets stored into this memory area.

In the opposing direction, the network application is writing packets into the shared memory area region. A DMA transfer is triggered to copy the packet from the host memory to the NIC card buffers.

No interrupt is used with this method, but it requires the network application to check at a regular interval whether a new packet has hit the NIC. This method is well suited for high-rate packet processing: if packets are arriving at a slow rate this algorithm is less efficient than the event-based method.

Network Devices Virtualization

Like CPU virtualization, two kinds of methods are used to virtualize network devices:

- software-based emulation
- hardware-assisted emulation

Software-based emulation is widely supported but can suffer from poor performance. Hardware-assisted emulation provides good performance thanks to its hardware acceleration but requires hardware that supports some specific features.

Software-based Emulation

Two solutions are proposed for device virtualization with software:

- Traditional Device Emulation (binary translation): the guest device drivers are not aware of the virtualization environment. During runtime, the Virtual Machine Manager (VMM), usually QEMU/KVM, will trap all the IO and Memory-mapped I/O (MMIO) accesses and emulate the device behavior (trap and emulate mechanism).The Virtual Machine Manager (VMM) emulates the I/O device to ensure compatibility and then processes I/O operations before passing them on to the physical device (which may be different). Lots of VMEXIT (context switching) is generated with this method. It provides poor performance.
- Paravirtualized Device Emulation (virtio): the guest device drivers are aware of the virtualization environment. This solution uses a front-end driver in the guest that works in concert with a back-end driver in the VMM. These drivers are optimized for sharing and have the benefit of not needing to emulate an entire device. The back-end driver communicates with the physical device. Performance is much better than with traditional device emulation.

Software-emulated devices can be completely virtual with no physical counterpart exposing a compatible interface.

Hardware-assisted Emulation

A physical device is directly assigned to a single virtual machine. TTwo solutions are proposed for device virtualization assisted with hardware direct assignment, allowing a VM to directly access a network device. Thus the guest device drivers can directly access the device configuration space to launch a DMA operation in a safe manner, via IOMMU, for example.

The drawbacks are:

- Direct assignment has limited scalability. A physical device can only be assigned to a single VM.
- IOMMU must be supported by the host CPU (Intel VT-d or AMD-Vi feature).

SR-IOV: with SR-IOV (Single Root I/O Virtualization), each physical device (physical function) can appear as multiple virtual ones (aka virtual function). Each virtual function can be directly assigned to one VM, and this direct assignment uses the VT-d/IOMMU feature.

The drawbacks are:

- IOMMU must be supported by the host CPU (Intel VT-d or AMD-Vi feature).
- SR-IOV must be supported by the NIC device (but also by the BIOS, the host OS and the guest VM).

Emulated Network Devices

The following two emulated network devices are provided with QEMU/KVM:

- e1000 device: emulates an Intel E1000 network adapter (Intel 82540EM, 82573L, 82544GC).
- e1000e device: emulates a newer Intel network adapter (Intel 82574)
- rtl8139 device: emulates a Realtek 8139 network adapter.

Paravirtualized Network Device

Virtio is an open specification for virtual machines' data I/O communication, offering a straightforward, efficient, standard, and extensible mechanism for virtual devices, rather than a vendor-specific, per-environment or per-OS mechanisms. Virtio uses the fact that guests can share memory with the host for I/O to implement it.

Virtio was developed as a standardized open interface for virtual machines to access simplified devices such as block devices and network adaptors.

Virtio Frontend and Backend

Virtio interface is composed of a backend component and a frontend component as shown in Figure 2.4:

- The frontend component is the guest side of the virtio interface
- The backend component is the host side of the virtio interface

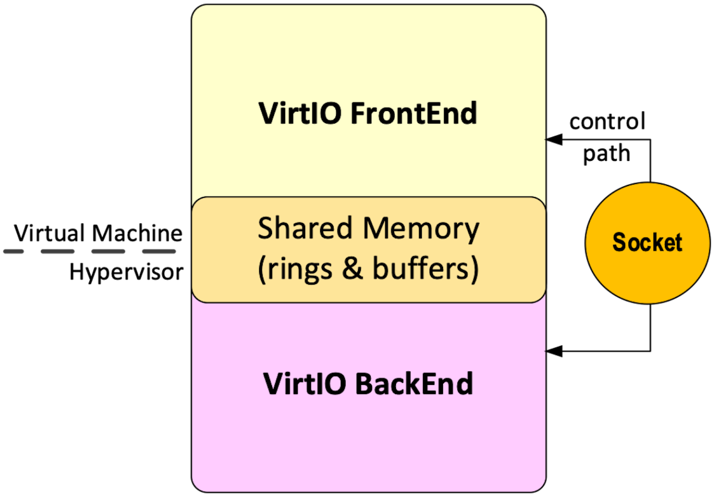


Figure 2.4 Virtio Frontend and Backend

Virtio Transport Protocol

The virtio network driver is the virtio frontend component exposed into the guest VM. The virtio network device is the virtio backend component exposed by the hypervisor into the host operating system. The virtual network frontend and backend are interconnected with a transport protocol (usually PCI/PCIe). See Figure 2.5.

The virtio drivers must be able to allocate memory regions that both the hypervisor and the devices can access for reading and writing via memory sharing. Two different domains have to be considered for a network device:

- Virtio device initialization, activation or shutdown (control plane)
- Network packets transfer through the virtio device (data plane)

The control plane is used for capability exchange negotiation between the host and guest, both for establishing and terminating the data plane. The data plane is used for transferring the actual packets between the host and guest.

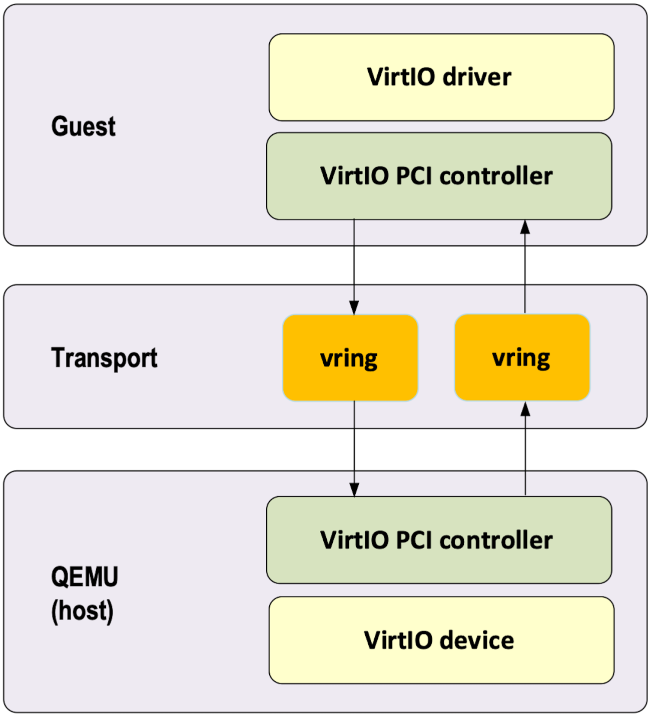


Figure 2.5 *Virtio Transport*

Virtqueues are the mechanism for bulk data transport on virtio devices. They are composed of:

- guest-allocated buffers that the host interacts with (read/write packets)
- descriptor rings

Virtqueues are controlled with I/O Registers notification messages:

- Available Buffer Notification: Virtio driver notifies there are buffers that are ready to be processed by the device.
- Used Buffer Notification: Virtio device notifies it has finished processing some buffers.

Virtio Device Network Backend

The network backend interacts with the emulated NIC and is exposed on the host side. Usually the network backend is a tap device but other backends are proposed with Virtio (SLIRP, VDE (Virtual Distributed Ethernet), Socket).

Tap devices are virtual point-to-point network devices that the user's applications can use to exchange L2 packets. Tap devices require the tun kernel module to be loaded. Tun kernel modules create a kind of device in the /dev/net system directory tree (/dev/net/tun). Each new tap device has a name in the /dev/net/tree filesystem.

Virtio Net Backend Drawbacks

The usual transport backend used by the Virtio net device presents some inefficiencies:

- syscall and data copy are required for each packet to send or receive through the tap interface (no bulk transfer mode).
- When there is a packet available for the virtio device (backend) the virtio driver (front end) creates a notice with an interrupt message (IOCTL).
- Each interrupt message stops vCPU execution and generates a context switch (vmexit). Then the host processes the available packet and resumes (vmexit) the VM execution using a syscall.

Each time a packet is sent, the VM stops to get the available packet processed. Hypervisor is involved in both virtio control plane and data plane.

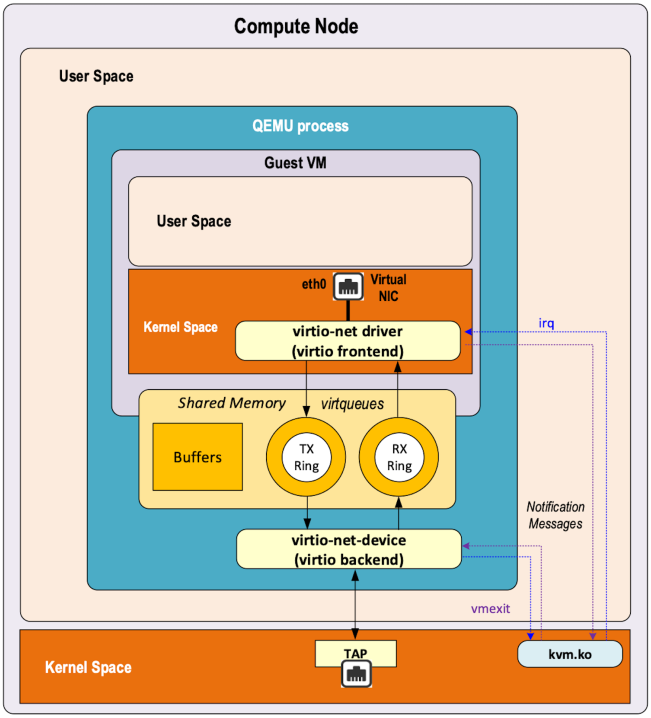


Figure 2.6 Virtio Transport

vhost protocol

The vhost protocol was designed to address the virtio device transport backend limitations. It's a message-based protocol that allows the hypervisor to offload the data plane to a handler. The handler is a component that manages virtio data forwarding. The host hypervisor no longer processes packets.

The data plane is fully offloaded to the handler that reads or writes packets to/from the virtqueues. The vhost handler directly accesses the virtqueues memory region in addition to sending and receiving notification messages.

The vhost handler is made up of two parts:

vhost-net:

- a kernel driver
- exposes a character device on /dev/vhost-net
- uses ioctls to exchange vhost messages (vhost protocol control plane),
- uses irqfd and ioeventfd file descriptor to exchange notifications with the guest
- spawns a vhost worker thread

vhost worker:

- a Linux thread named vhost-<pid> (<pid> is the hypervisor process ID)
- handles the I/O events (generated by virtio driver or tap device)
- forwards packets (copy operations)

A tap device is still used to communicate the guest instance with the host, but the virtio data plane is managed by the vhost handler and is no longer processed by the hypervisor (see Figure 2.7). Guest instances are no longer stopped (context switch with a VMEXIT) at each VirtIO packet transfer. New virtio vhost-net packet processing backend is completely transparent to the guest who still uses the standard virtio interface.

Physical Network Device Direct I/O Assignment

KVM guests usually have access to software-based emulated NIC devices (either para-virtualized devices with virtio or traditional emulated devices). On host machines that have Intel VT-d or AMD IOMMU hardware support, another option is possible. PCI devices may be assigned directly to the guest, allowing the device to be used with minimal performance overhead. Assigned devices are physical devices that are exposed to the VM. This method is also known as passthrough.

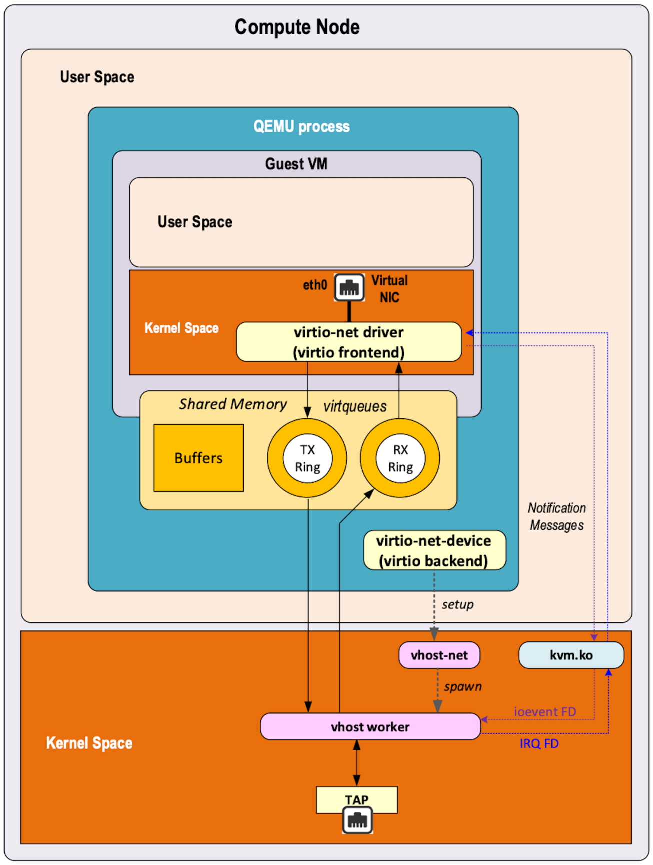


Figure 2.7 Virtio Vhost Handler

The VT-d or AMD IOMMU extensions must be enabled in BIOS in order to conduct device direct assignment. Two methods are supported:

- PCI passthrough: PCI devices on the host system are directly attached to virtual machines, providing guests with exclusive access to PCI devices for a range of tasks. This enables PCI devices to appear and behave as if they were physically attached to the guest virtual machine.
- VFIO device assignment: VFIO improves on previous PCI device assignment architecture by moving device assignment out of the KVM hypervisor and enforcing device isolation at the kernel level.

With VFIO the physical device is exposed to the host user space memory and is made visible from the guest VM it has been assigned to. See Figure 2.8.

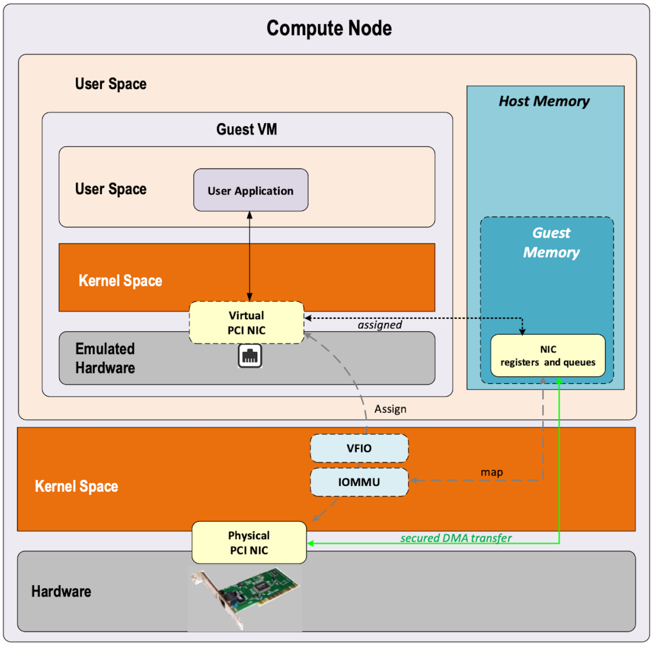


Figure 2.8 Physical Network Device Direct I/O Assignment

SR-IOV

The Single Root I/O Virtualization (SR-IOV) specification is defined by the PCI-SIG (PCI Special Interest Group). This is a PCI Express (PCI-e) that extends a single physical PCI function to share its PCI resources as separate virtual functions (VFs).

The physical function contains the SR-IOV capability structure and manages the SR-IOV functionality (it can be used to configure and control a PCIe device).

A single physical port (root port) presents multiple, separate virtual devices as unique PCI device functions (up to 256 virtual functions – depending on device capabilities).

Each virtual device may have its own unique PCI configuration space, memory-mapped registers, and individual MSI-based interrupts (MSI: Message Signalled Interrupts). Unlike a physical function, a virtual function can only configure its own behavior. Each virtual function can be directly connected to a VM via PCI device assignment (passthrough mode).

SR-IOV improves network device performance for each virtual machine as it can share a single physical device between several virtual machines using the device direct I/O assignment method (Figure 2.9).

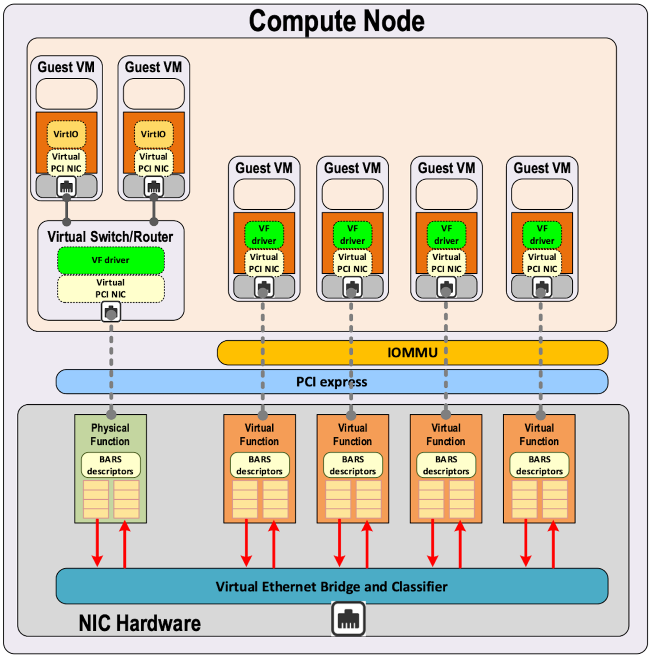


Figure 2.9 Single Root I/O Virtualization

With SR-IOV, each VM has direct access to the physical network using the assigned virtual function interface allocated to it. They can communicate altogether using the Virtual Ethernet Bridge provided by the NIC card. A virtual switch can also use SRIOV to get access to the physical network. A VM using a SR-IOV assigned virtual function device has direct access to the physical network and is not connected to any intermediate virtual network switch or router (see Figure 2.10).

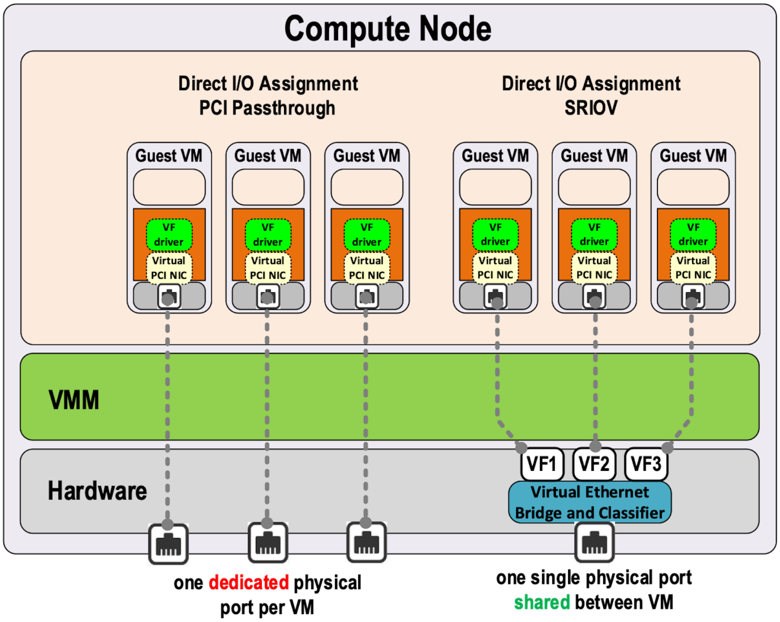


Figure 2.10 *Single Root I/O Virtualization*

The following command can be used to check whether SR-IOV is supported or not on a physical NIC card:

```
$ lspci -s <NIC_BDF> -vvv | grep -i "Single Root I/O Virtualization"
```

Virtio, SR-IOV, and SDN

Virtio provides a lot of flexibility by offering a standardized driver which is fully independent of the hardware used on the physical platform hosting VM instances.

When Virtio connectivity is used a VM can be easily migrated from one host to another using a live migration feature. When SR-IOV is used, this live migration is not an easy task.

Indeed, network drivers used by a VM depend on used hardware on the bare metal nodes that are hosting them. In order to make the VM migration from one bare metal node to another, both nodes must at least use the same hardware NIC model. But when SRIOV is used, VM connectivity has about the same performance as a real physical NIC, whereas with Virtio, performance could be poor.

Also, by providing direct access to the physical NIC, SR-IOV is making host virtual network nodes (virtual router/switch) used by the SDN solution totally blind about the VM using such connectivity. Local traffic switching between the VM connected on the same SR-IOV physical card is achieved by the virtual Ethernet bridge proposed by SR-IOV. Communication between VMs connected onto the distinct SRIOV physical ports must rely on the physical network.

SDN vSwitch/vRouter usage is very limited with SR-IOV. Indeed, packet switching between VMs that are using VFs from the same SR-IOV physical port are using the physical virtual Ethernet bridge hosted in the physical NIC. See Figure 2.11.

Only a few use cases are relevant. The first provides internal connectivity between VMs using distinct SR-IOV physical ports connected to a Virtual Switch/Router (it avoids sending the traffic out of the server to be processed by the physical network).

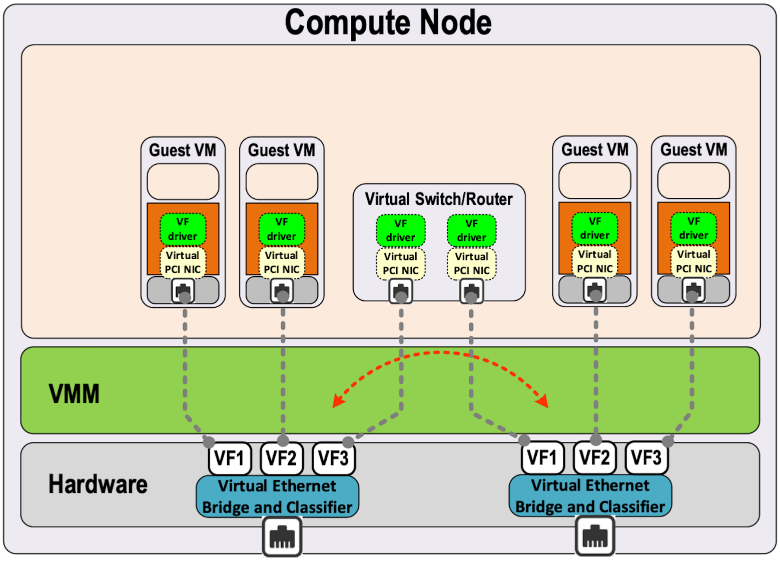


Figure 2.11 Connectivity Across Distinct Physical SRIOV Ports Using a Dual Homed Virtual Switch/Router

A second use case is building hybrid mode solutions with multi-NIC VM. Network traffic (management traffic, for instance) not requiring high performance uses emulated NIC. Network connectivity requiring high performance is processed by the SR-IOV assigned NIC (for instance, video data traffic). See Figure 2.12.

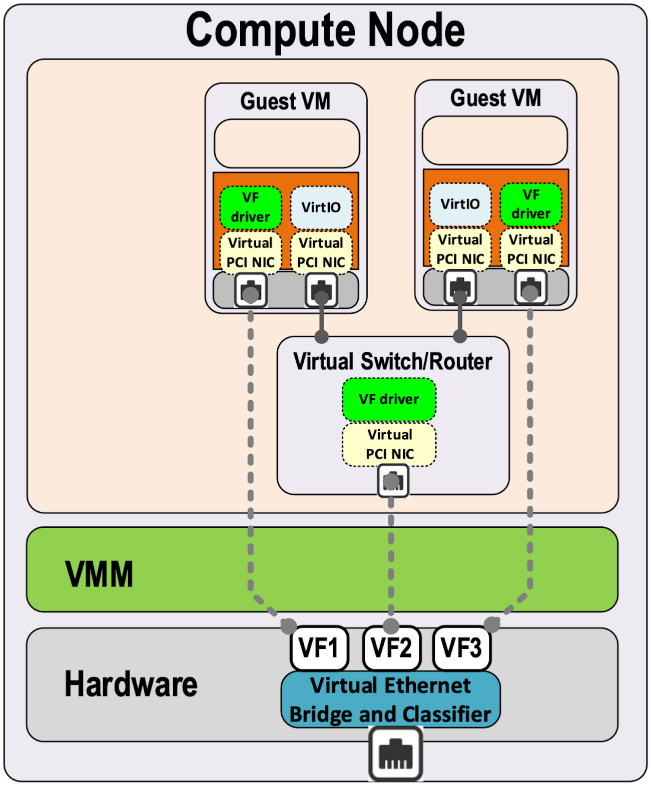


Figure 2.12 Building Hybrid Mode Solutions with Multi-NIC VM

With SR-IOV you get high performance but with poor flexibility and no network virtualization features. With VirtIO you get a high level of network virtualization suitable for SDN, which is very flexible with poor performances.

For SDN use cases, you need network virtualization features and performance. DPDK will bring both.

Network Packet Processing Performance Requirements

The Ethernet minimum frame size is 64 Bytes. See Figure 2.13. When Ethernet frames are sent onto the wire, Inter Frame Gap and Preamble bits are added. The minimum size of Ethernet frames on the physical layer is 84 Bytes (672 bits).

For a 10 Gbit/s interface, the number of frames per seconds can reach up to 14.88 Mpps for traffic using the smallest Ethernet frame size. It means a new frame will have to be forwarded every 67 ns. A CPU running at 2Ghz has a 0.5 ns cycle. Such a CPU has a budget of only 134 cycles per packet to process a flow of 10 Gb/s.

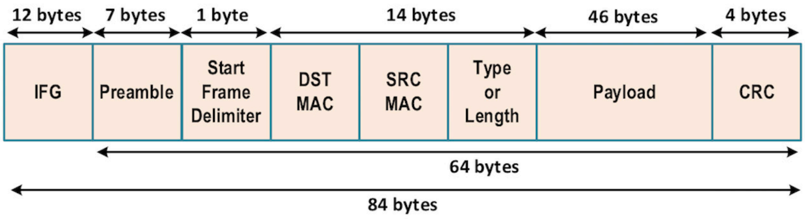


Figure 2.13 *64 bytes Ethernet Packet Frame Header*

Generic Linux Ethernet drivers are not powerful enough to process such a 10Gb/s packet flow. Indeed, with regular Linux NIC drivers, a lot of time is required to perform packet processing in Linux Kernel using interrupt mechanism, and transfer application data from host memory to a NIC.

DPDK is one of the best solutions available as it allows you to build a network application using high-speed NICs and work at wire speed. Therefore, Contrail is proposing DPDK as one of the solutions to be used for the physical compute connectivity.

DPDK and Network Applications

DPDK Application Working Principle

The Data Plane Development Kit (DPDK) dedicates one (or more) CPUs to one (or more) threads that are continuously polling one (or more) DPDK NIC RX queue. CPUs on which a DPDK polling thread is started at 100% whether some packets have to be processed or not, as no interrupt mechanism is used in DPDK to warn the DPDK application that a packet has been received. (See Figure 2.14.)

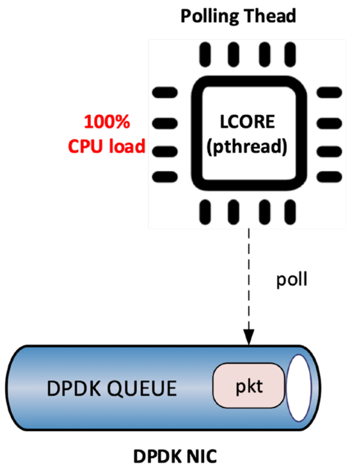


Figure 2.14 *DPDK Working Principle*

Using the DPDK library API, physical NIC packets are made available in user space memory in which the DPDK application is running. So, when DPDK is used, there is no user space to kernel space context switching, which saves a lot of CPU cycles. Also, the host memory is using a large continuous memory area. The huge pages allow large data transfers and avoid high data fragmentation in memory, which would require a higher memory management effort at the application level. Such a fragmentation would also cost some precious CPU cycles.

Hence, most of the CPU cycles of the DPDK pinned CPU are used for polling and processing packets delivered by the physical NIC in DPDK queues. As a result, the packet forwarding task can be processed at a very high speed. If one CPU is not powerful enough to manage incoming packets that are hitting the physical NIC at a very high rate, you can allocate an additional one to the DPDK application in order to increase its packet processing capacity.

A DPDK application is a multi-thread program that uses the DPDK library to process network data. In order to scale, you can start several packet polling and processing threads (each one pinned on a dedicated CPU) that are running in parallel.

Three main components are involved in a DPDK application (see Figure 2.15):

- Physical NIC
    - buffering packets in physical queues
    - using DMA to transfer packets in host memory
- DPDK NIC abstraction with its queue representation in huge pages host memory:
    - descriptor rings
    - mbuf (to store packets)
- Linux pThread to poll and process packets received in DPDK NIC queues.

DPDK Overview

DPDK is a set of data plane libraries and network interface controller drivers for fast packet processing, currently managed as an open-source project under the Linux Foundation. The main goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications.

The framework creates a set of libraries for specific environments through the creation of an Environment Abstraction Layer (EAL), which may be specific to a mode of the Intel architecture (32-bit or 64-bit), Linux user space compilers, or a specific platform. These environments are created through the use of make files and configuration files. Once the EAL library is created, the user may link with the library to create their own applications.

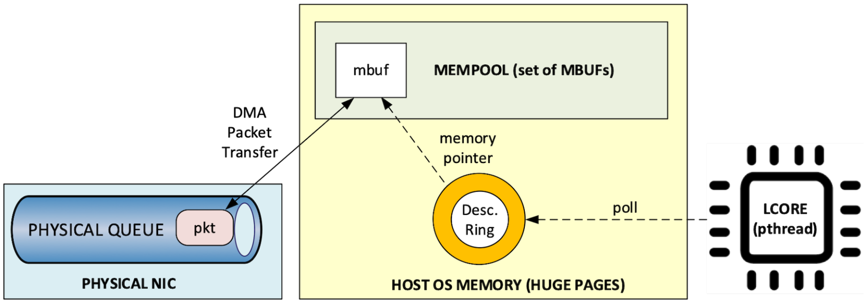


Figure 2.15 Three Components of DPDK

The DPDK implements a "run to completion model" for packet processing, where all resources must be allocated prior to calling data plane applications, running as execution units on logical processing cores.

The model does not support a scheduler and all devices are accessed by polling. The primary reason for not using interrupts is the performance overhead imposed by interrupt processing.

For more information please refer to dpdk.org documents: http://dpdk.org/doc/guides/prog_guide/index.html

DPDK Software Architecture

DPDK is a set of programing libraries that can be used to create an application that needs to process network packets at a high speed. DPDK has the following functions:

- A queue manager that implements lockless queues.
- A buffer manager that pre-allocates fixed size buffers.
- A memory manager that allocates pools of objects in memory and uses a ring to store free objects.
- Poll mode drivers (PMD) that are designed to work without asynchronous notifications, reducing overhead.
- A packet framework made up of a set of libraries that are helpers to develop packet processing.

In order to reduce the Linux user to kernel space context switching, all of these functions are made available by DPDK into the user space where applications are running. User applications using DPDK libraries have direct access to the NIC cards, without passing through a NIC Kernel driver as is required when DPDK is not used.

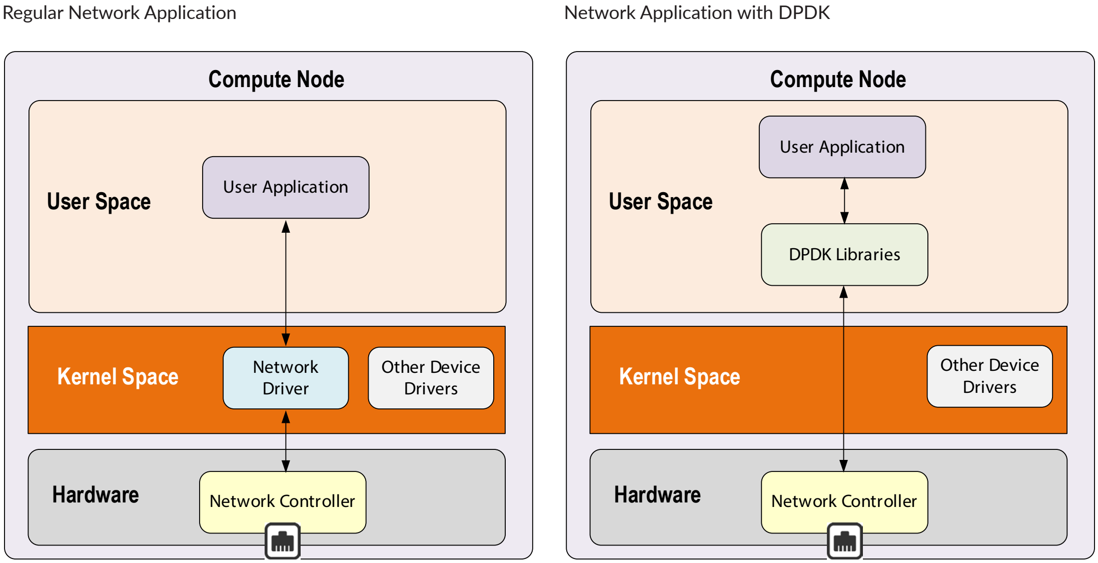


Figure 2.16 Regular versus DPDK Network Applications

DPDK allows you to build user-space multi-thread network applications using the POSIX thread (pthread) library. DPDK is a framework made of several libraries:

- Environment Abstraction Layer (EAL)
- Ethernet Devices Abstraction (ethdev)
- Queue Management (rte_ring)
- Memory Pool Management (rte_mempool)
- Buffer Management (rte_mbuf)
- Timer Manager (librte_timer)
- Ethernet Poll Mode Driver (PMD)
- Packet Forwarding Algorithm made up of Hash (librte_hash) and Longest Prefix Match (LPM,librte_lpm) libraries
- IP protocol functions (librte_net)

The ethdev library exposes APIs to use the networking functions of DPDK NIC devices. The bottom half of ethdev is implemented by NIC PMD drivers. Thus, some features may not be implemented.

Poll Mode Ethernet Drivers (PMDs) are a key component for DPDK. These PMDs bypass the kernel and provide direct access to the Network Interface Cards (NIC) used with DPDK.

Linux user space device enablers (UIO or VFIO) are provided by the Linux Kernel and are required to run DPDK. They allow PCI devices to discover and expose information and address space through the `/sys` directory tree.

DPDK libraries (See Figure 2.17) allow kernel-bypass application development:

- probing for PCI devices (attached via a Linux user space device enabler),
- huge-page memory allocation
- and data structures geared toward polled-mode message-passing applications:
  - such as lockless rings
  - memory buffer pools with per-core caches

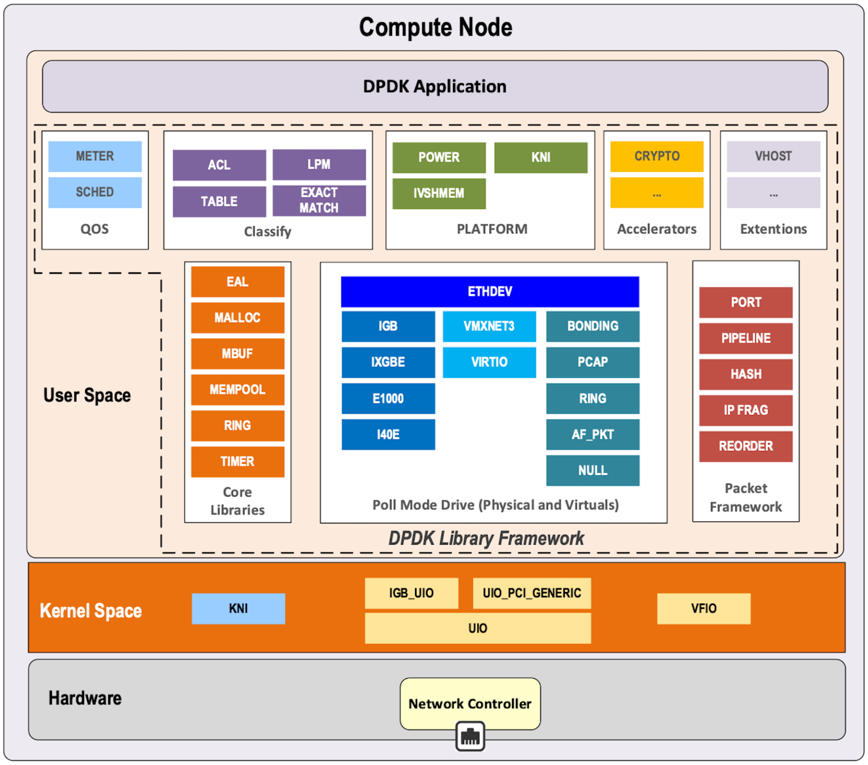


Figure 2.17 Overview of DPDK Libraries

Only a few libraries have been described in this diagram. The set of libraries is enriched with each new DPDK release: see https://www.dpdk.org/).

DPDK Environment Abstraction Layer

The EAL is responsible for providing access to low-level resources such as hardware and memory space (see Figure 2.18). It provides a generic interface that hides the specifics of the environment from the applications and libraries. The EAL performs physical memory allocation using mmap() in hugetlbfs (using the huge page sizes to increase performance). Services provided by EAL are:

- DPDK loading and launching
- Support for multi-process and multi-thread execution types
- Core affinity/assignment procedures
- System memory allocation/de-allocation
- Atomic/lock operations
- Time reference
- PCI bus access
- Trace and debug functions
- CPU feature identification
- Interrupt handling
- Alarm operations
- Memory management (malloc)

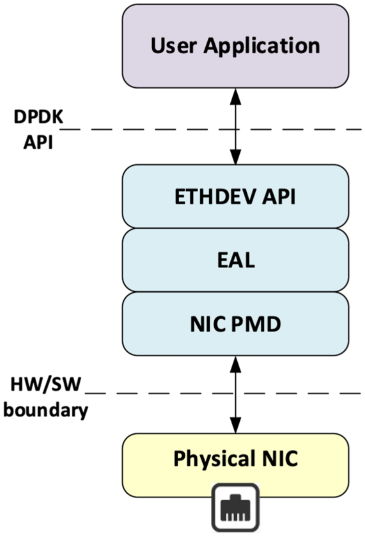


Figure 2.18 Environment Abstraction Layer (EAL)

DPDK Memory Management

DPDK Optimized Memory Management for Speed

DPDK has a highly optimized memory manager that works on a group of fixed size objects called a mempool. Every one of them are pre-allocated. DPDK does not encourage dynamic allocations because it consumes a lot of CPU cycles and it is a speed killer.

DPDK stores incoming packets into mbufs (memory buffers). DPDK pre-allocates a set of mbufs and keeps it in a pool called mempool. DPDK makes use of mempools each time it needs to allocate a mbuf where packets are stored. Instead of allocating a single mbuf, DPDK does bulk allocation, or bulk free once packets are consumed. By doing this, packets to be processed (mbufs) are already in cache memory. Therefore, DPDK is very cache friendly.

Mempool has further optimizations. It is also very cache friendly. Everything is aligned to the cache and has a some mbufs allocated for each DPDK thread or lcore. Each mempool is bound with rings, which are referencing mbufs containing packets stored in mempool.

Each ring is a highly optimized lockless ring. It can be used by several lcores in a multi-producer/multi-consumer kind of scenario without locks. By avoiding locks, DPDK gets large performance gains, as data structures locking is also a speed killer.

mbufs and mempools

Network data is stored in compute central memory (in hugepage area). DPDK uses message buffers known as `mbufs` to store packet data into the host memory. These `mbufs` are stored in memory pools known as `mempools`. See Figure 2.19.

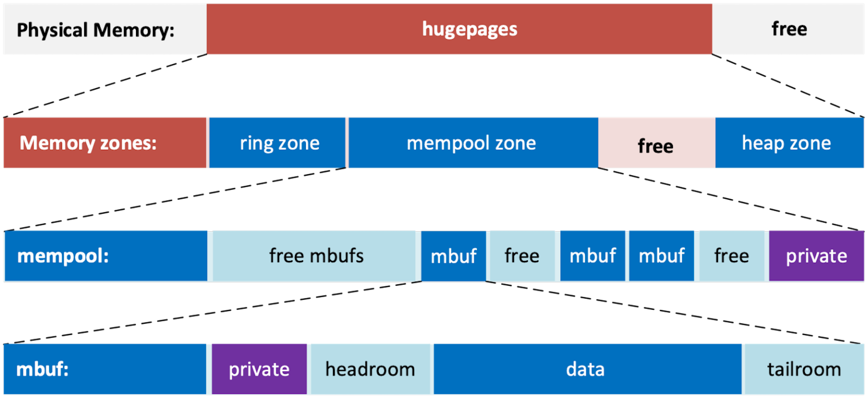


Figure 2.19 mbufs and mempools

The mbufs are storing DPDK NIC incoming and outgoing packets which have to be processed by the DPDK application.

Packet Descriptors

DPDK queues are not storing the packets but a descriptor points to the real packet, as seen in Figure 2.20. It avoids performing a data transfer that would be needed when packets have to be forwarded from one DPDK NIC to another.

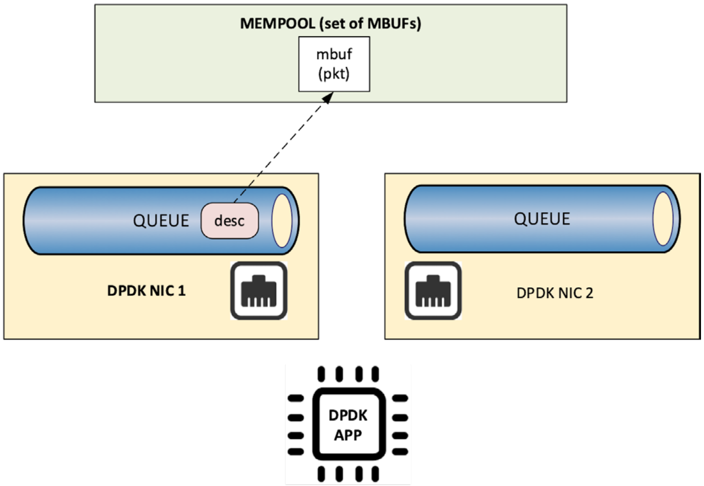


Figure 2.20 Packet Descriptors

Packets are not moved from one queue to another, these are the descriptors (pointers) that are moving from one queue to another, as seen in Figure 2.21.

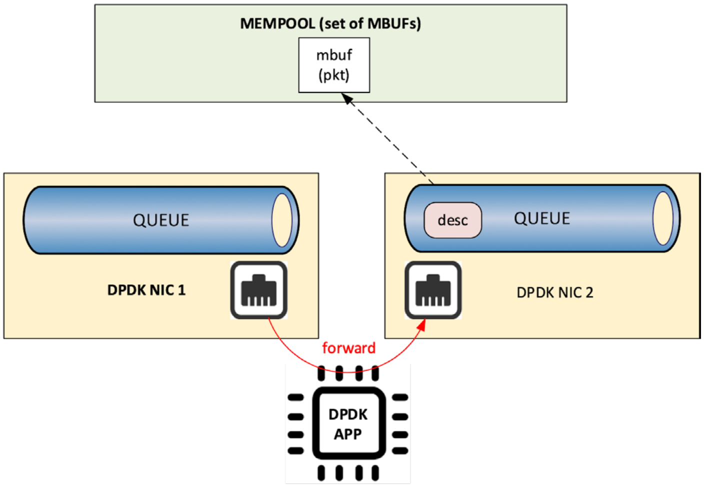


Figure 2.21 Packet Descriptors MOVING

DPDK Rings

Descriptors are set up as a ring, or a circular array of descriptors. Each ring describes a single direction DPDK NIC queue. Each DPDK NIC queue is made up of two rings (one per direction: one RX ring, one TX ring). See Figure 2.22.

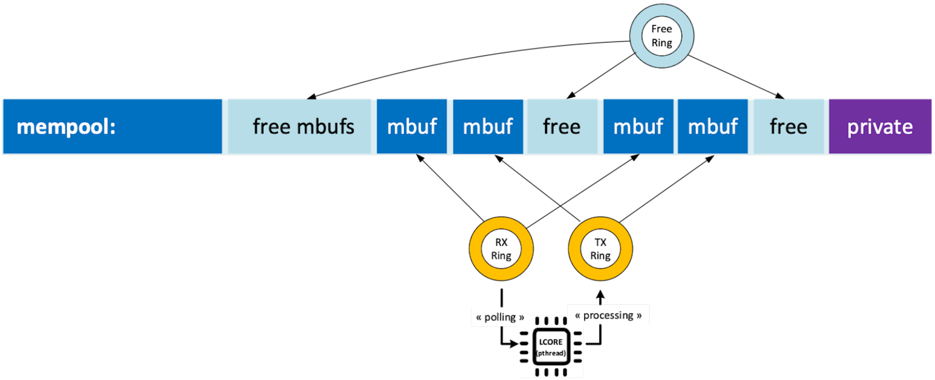


Figure 2.22 DPDK Rings

Each descriptor points onto a packet that has been received (RX ring) or that is going to be transmitted (TX ring). The more descriptors RX/TX rings contain, the more memory required to store data in each mempool (number of mbufs).

Data Transfer Between Host NIC and Memory

The DPDK application is only processing packets that are exposed in user space host OS memory. DPDK rings are an abstraction of the real NIC queues: DPDK is using DMA to keep synchronized at any time between the NIC hardware queues and its DPDK representation in the host memory.

Physical NIC Incoming Packets

When an incoming packet reaches the physical NIC interface, it is stored in NIC physical queue memory. The RX ring manages packets that have to be processed by a DPDK application. Synchronization between the host OS and the NIC happens through two registers, whose content is interpreted as an index in the RX ring (see Figure 2.23):

- Receive Descriptor Head (RDH): indicates the first descriptor prepared by the OS that can be used by the NIC to store the next incoming packet.
- Receive Descriptor Tail (RDT): indicates the position to stop reception, i.e. the first descriptor that is not ready to be used by the NIC.

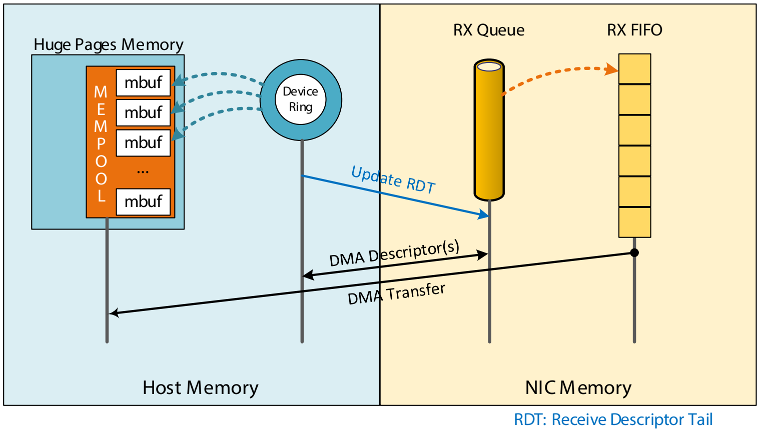


Figure 2.23 Receive Descriptor Tail (RDT)

DMA transfer transparently copies packets from physical NIC memory to the host central memory. DMA uses the RDT descriptor as a destination memory address for the data to be transferred. Once packets have been transferred into host memory both RX rings and RDT are updated.

Physical NIC Outgoing Packets

When a packet has to be sent from host memory to the physical NIC interface, it is referenced in NIC TX ring by the DPDK application, as seen in Figure 2.24. TX ring manages packets that have to be transferred onto a NIC card.

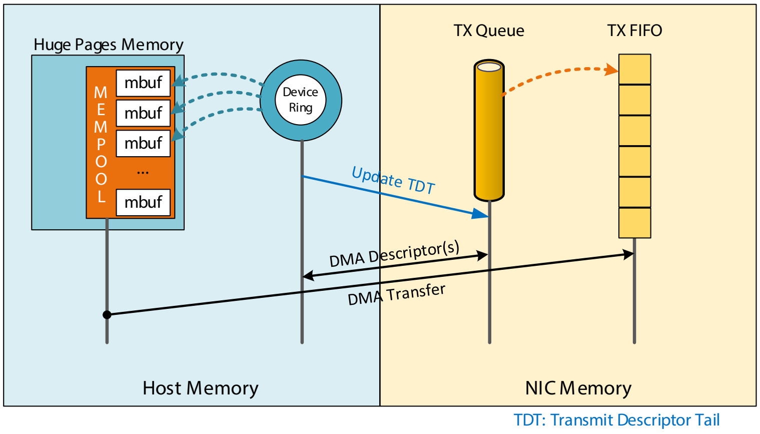


Figure 2.24 Transmit Descriptor Tail (TDT)

Synchronization between the host OS and the NIC happens through two registers, whose content is interpreted as an index in the TX ring:

- Transmit Descriptor Head (TDH): indicates the first descriptor that has been prepared by the OS and has to be transmitted on the wire.
- Transmit Descriptor Tail (TDT): indicates the position to stop transmission, i.e. the first descriptor that is not ready to be transmitted, and that will be the next to be prepared.

DPDK and Packet Processing

Linux pthreads

Multithreading is the ability of a CPU (single-core in a multi-core processor architecture) to provide multiple threads of execution concurrently. In a multithreaded application, the threads share some CPU resources memory:

- CPU caches
- translation lookaside buffer (TLB).

A single Linux process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (local variables).

Linux pthreads (POSIX threads) is a C library containing set functions that allow managing threads into an application. DPDK uses the Linux pthreads library.

DPDK lcores

DPDK uses threads that are designed as “lcore.” A lcore refers to an EAL thread, which is really a Linux pthread, which is running onto a single processor execution unit:

- first lcore: that executes the main() function and that launches other lcores is named master lcore.
- any lcore: that is not the master lcore is a slave lcore.

Lcores are not sharing CPU units. Nevertheless, if the host processor supports hyperthreading, a core may include several lcores or threads. Lcores are used to run DPDK application packet processing threads. Several packet processing models are proposed by DPDK. The simplest one is the run-to-completion model, shown in Figure 2.25.

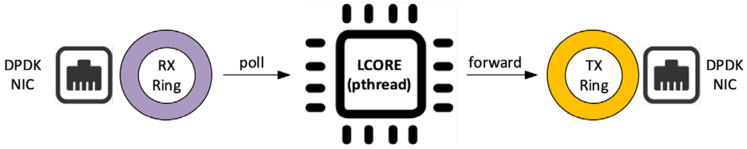


Figure 2.25 Run-To-Completion

Run to completion uses a single thread (lcore) for end-to-end packet processing (packet polling, processing and forwarding).

Multicore Scaling - Pipeline Model

A complex application is typically split across multiple cores, with cores communicating through software queues. Packet framework facilitates the creation of pipelines. Each pipeline thread is assigned to a CPU and uses software queues like output or/and input ports, as shown in Figure 2.26.

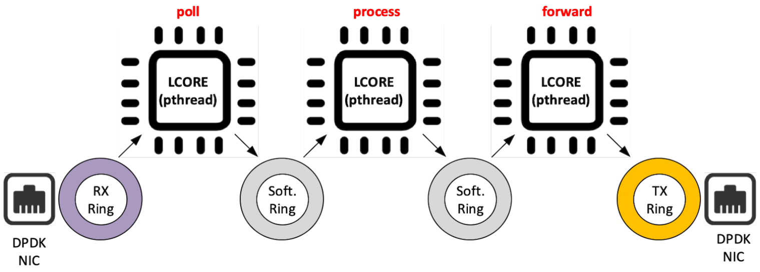


Figure 2.26 Pipeline Model

For instance, Contrail DPDK vRouter is using such a model for GRE encapsulated packet processing.

Control Threads

It is possible to create control threads which can be used for management/infrastructure tasks and are used internally by DPDK for multi process support and interrupt handling.

Service Core

DPDK service cores enable a dynamic way of performing work on DPDK lcores. Service core support is built into the EAL and an API is provided to optionally allow applications to control how the service cores are used at runtime.

DPDK and Poll Mode Drivers (PMD)

When DPDK is used, network interfaces are no longer managed in kernel space. The regular Linux NIC driver, which is usually used to manage the NIC, has to be replaced by a new driver that is able to run into user space. This new driver, called Poll Mode Driver (PMD), is used to manage the network interface into user space with the DPDK library.

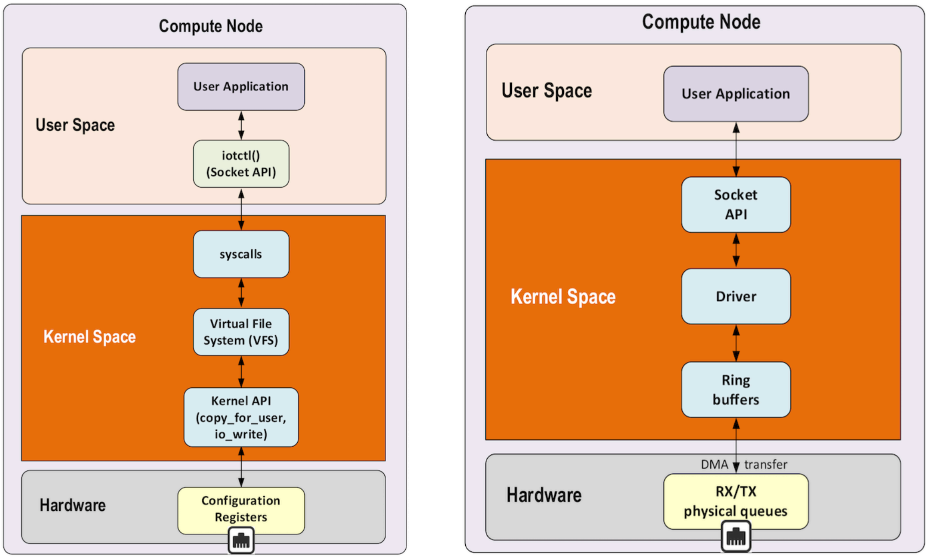
Physical NIC and Base Address Registers (BAR)

PCI devices have a set of registers referred to as configuration space for devices. These configuration space registers are mapped to host memory locations.

When a PCI device is enabled, the system’s device drivers (by writing configuration commands to the PCI controller) programs the BAR to inform the PCI device of its address mapping. Next, the host operating system is able to address this PCI device.

Linux NIC Drivers

With usual Linux NIC kernels, both NIC configuration and packet processing is done in kernel space. User applications, which have to establish a TCP connection or send a UDP packet, use the sockets API exposed by libc library. See Figure 2.27.



NIC Configuration **NIC Packet Processing**

Figure 2.27 NIC Configuration and Packet Processing APIs

Linux packet processing with sockets API requires the following operations, which can be costly:

- Kernel Linux system calls
- Multitask context switching on blocking I/O
- Data copying from kernel (ring buffers) to user space
- Interrupt handling in kernel

With usual Linux drivers most operations occur in kernel modes and require lots of user space to kernel space context switching and interruption mechanisms. The heavy context switching usage costs lots of CPU cycles and limits the numbers of packets that a CPU is able to process. Such drivers are not able to perform packet processing at expected high speeds, especially when 10/40/100G Ethernet generation cards are used on a Linux System.

Poll Mode Drivers

PMD consists of APIs running in user space to configure the devices and their respective queues. In addition, a PMD accesses the RX and TX descriptors directly without any interrupts (with the exception of Link status change interrupts) to quickly receive, process, and deliver packets in the user's application.

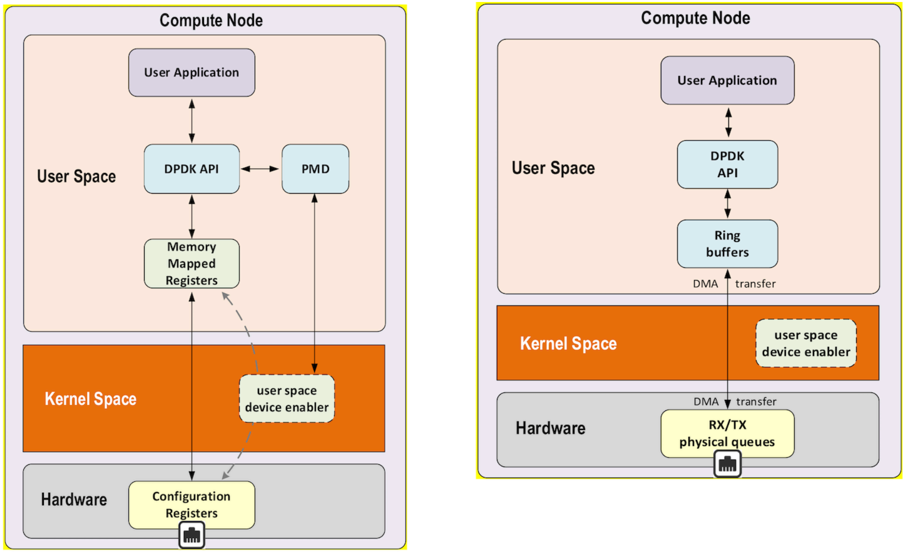
PMDs are involved in NIC configuration. They expose NIC configuration registers into a host memory area that is directly reachable from user space. See Figure 2.28.

In short, PMDs are user space pthreads that:

- call specific EAL functions
- have a per NIC implementation
- have direct access to RX/TX descriptors
- use Linux user space device enablers (UIO or VFIO) driver for specific control changes (interrupts configuration)

Hence user applications can directly configure the NIC cards they are using from Linux user space where they are running.

A first configuration phase is using PMDs and the DPDK library to configure DPDK rings buffers into Linux user space. Next, incoming packets will be automatically transferred with DMA (direct memory access) mechanism from the NIC physical RX queues in NIC memory to DPDK RX rings buffered in host memory. DMA is also used to transfer outgoing packets from the DPDK TX rings buffer in host memory to the NIC physical TX queues in NIC memory. DMA offloads expensive memory operations, such as large copies or scatter-gather operations, from the CPU.



NIC Configuration

NIC Packet Processing

Figure 2.28 NIC Configuration and Packet Processing DPDK APIs

Direct Memory Access

DMA allows PCI devices to read (write) data from (to) memory without CPU intervention. This is a fundamental requirement for high-performance devices.

DMA is a mechanism that uses a specific hardware controller to manage, read, and write operations into the main system memory (RAM). This mechanism is totally independent of the CPU and does not consume any CPU resource. A DMA transfer is used to manage data transfer. DMA transfer is triggered by the CPU and works in the background using the specific hardware resource (a DMA controller).

DPDK rings and NIC buffers are synchronized with DMA. Thanks to this synchronization mechanism, DPDK applications can access transparently to NIC packets in user space reading or writing data in DPDK rings.

IOMMU

Input/output memory management unit (IOMMU) is a memory management unit (MMU) that connects a DMA capable I/O bus to the main memory. See Figure 2.29.

In virtualization, an IOMMU re-maps the addresses accessed by the hardware into a similar translation table that is used to map guest virtual machine address memory to host-physical addresses memory.

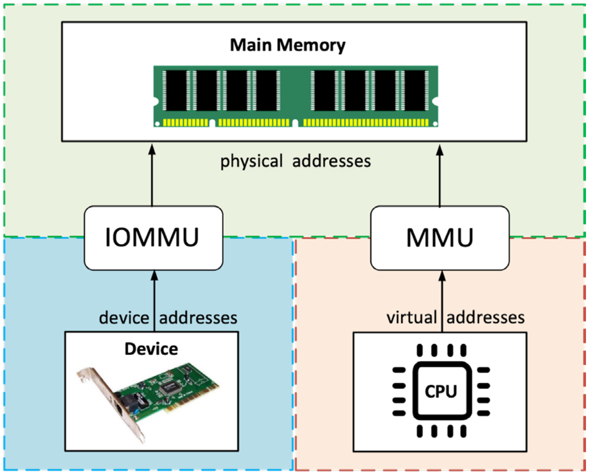


Figure 2.29 *IOMMU*

IOMMU provides a short path for devices to get access only to a well-scoped physical device memory area which corresponds to a given guest virtual machine memory. IOMMU helps to prevent DMA attacks that could originate via malicious devices. IOMMU provides DMA and interrupt remapping facilities to ensure I/O devices behave within the boundaries they've been allotted.

Intel has published a specification for IOMMU technology: *Virtualization Technology for Directed I/O* abbreviated as VT-d.

In order to get IOMMU enabled:

- both kernel and BIOS must support and be configured to use IO virtualization (such as VT-d).
- IOMMU must be enabled into Linux Kernel parameters in /etc/default/grub and run the `update-grub` command.

GRUB configuration example with IOMMU Passthrough enabled:

```
GRUB_CMDLINE_LINUX_DEFAULT="iommu=pt intel_iommu=on"
```

DPDK Supported NICs

The DPDK library includes PMDs for physical and emulated Ethernet controllers which are designed to work without asynchronous, interrupt-based signaling mechanisms.

- Available DPDK PMD for physical NIC:
  - I40e PMD for Intel X710/XL710/X722 10/40 Gbps family of adapters http://dpdk.org/doc/guides/nics/i40e.html

  - IXGBE PMD http://dpdk.org/doc/guides/nics/ixgbe.html
  - Linux bonding PMD http://dpdk.org/doc/guides/prog_guide/link_bonding_poll_mode_drv_lib.html
- Available DPDK PMD for Emulated NIC:
  - DPDK EM PMD supports emulated Intel 82540EM Gigabit Ethernet Controller (qemu e1000 device): http://doc.dpdk.org/guides/nics/e1000em.html
  - Virtio PMD for emulated VirtIO NIC: http://dpdk.org/doc/guides/nics/virtio.html
  - VMXNET3 NIC when VMware hypervisors are used: http://doc.dpdk.org/guides/nics/vmxnet3.html

MORE? Lots of other NICs are supported by DPDK: (cf: http://doc.dpdk.org/guides/nics/overview.html.

Different PMDs may require different kernel drivers in order to work properly (cf Linux user space device enablers). Depending on the PMD being used, a corresponding kernel driver should be loaded and bound to the network ports.

It is also preferable that each NIC has been flashed with the latest version of NVM (Non-Volatile Memory)/firmware.

Linux User Space Device Enablers

Most of PMD uses generic user space device enablers to expose physical NIC registers in user space into the host memory. Two space device enablers are widely used by the DPDK PMD and they are UIO and VFIO.

UIO - User Space IO

Linux kernel version 2.6, introduced the User Space IO (UIO) loadable module. UIO is a kernel-bypass mechanism that provides an API that enables user space handling of legacy interrupts (INTx).

UIO has some limitations:

- UIO does not manage message-signaled interrupts (MSI or MSI-X).
- UIO also does not support DMA isolation through IOMMU.

UIO only supports legacy interrupts so it is not usable with SR-IOV and virtual hosts that require MSI/MSI-X interrupts.

Despite these limitations, UIO is well suited for use in VMs, where direct IOMMU access is not available. In such a situation, a guest instance user space process is not isolated from other processes in the same instance. But the hypervisor can isolate any guest instance from others or hypervisor host processes using IOMMU.

Currently, two UIO modules are supported by DPDK:

- Linux Generic (uio_pci_generic), which is the standard proposed UIO module included in the Linux kernel.
- DPDK specific (igb_uio), which must be compiled with the same kernel as the one running on the target.

DPDK specific UIO Kernel module is loaded with insmod command after UIO module has been loaded:

```
$ sudo modprobe uio
$ sudo insmod kmod/igb_uio.ko
```

While a single command is needed to load Linux generic UIO module:

```
$ sudo modprobe uio_pci_generic
```

DPDK specific UIO module could be preferred in some situation to Linux Generic UIO module (cf: https://doc.dpdk.org/guides/linux_gsg/linux_drivers.html).

VFIO – Virtual Function I/O

Virtual Function I/O (VFIO) kernel infrastructure was introduced in Linux version 3.6. VFIO provides a user space driver development framework allowing user space applications to interact directly with hardware devices by mapping the I/O space directly to the application's memory. VFIO is a framework for building user space drivers that provides:

- Mapping of device's configuration and I/O memory regions to user memory.
- DMA and interrupt remapping and isolation based on IOMMU groups.
- Eventfd and irqfd based signaling mechanism to support events and interrupts from and to the user space application.
- VFIO exposes APIs that allow:
- create character devices (in /dev/vfio/),
- support ioctl calls,
- and support mechanisms for describing and registering interrupt notification.

VFIO driver is an IOMMU/device agnostic framework for exposing direct device access to user space, in a secure, IOMMU protected environment. For bare-metal environments, VFIO is the preferred framework for Linux kernel-bypass. It operates with the Linux kernel's IO. See Figure 2.30.

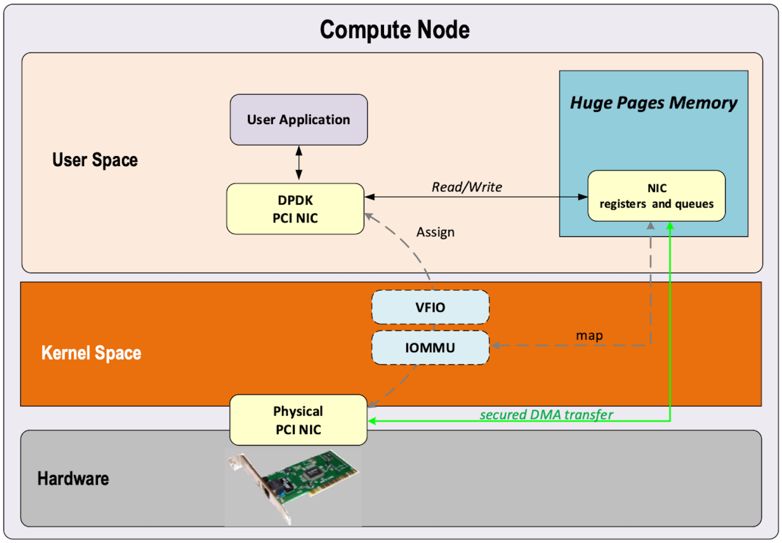


Figure 2.30 VFIO Driver

MMU subsystem is used to place devices into IOMMU groups. User space processes can open these IOMMU groups and register memory with the IOMMU for DMA access using VFIO ioctl calls. VFIO also provides the ability to allocate and manage message-signaled interrupt vectors.

A single command is needed to load VFIO module:

```
$ sudo modprobe vfio_pci
```

Despite the fact that VFIO was created to work with IOMMU, VFIO can be also be used without it (this is just as unsafe as using UIO).

Linux User Space Device Enablers to Be Used

VFIO is generally the preferred Linux user space device enabler to be used because it supports IOMMU to protect host memory. When a real hardware PCI device is attached to the host system and IOMMU is used with VFIO, all the reads/writes of that device done in user space by the DPDK application will be protected by the host IOMMU.

But there are a few exceptions. Figure 2.31 is an Intel recommendation for the choice of the Kernel driver to be used with DPDK: (https://software.intel.com/content/www/us/en/develop/articles/memory-in-dpdk-part-2-deep-dive-into-iova.html).

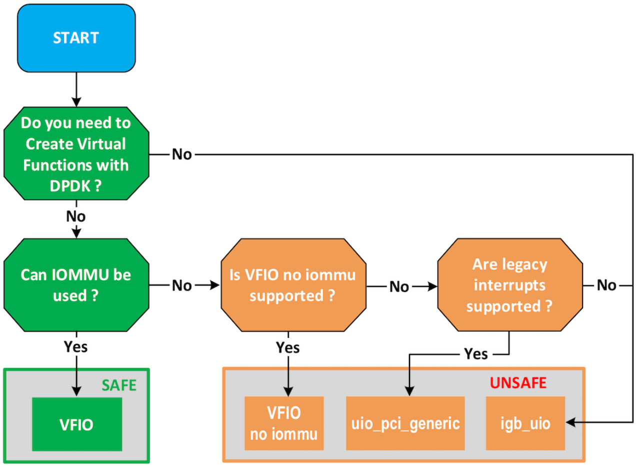

START
Do you need to Create Virtual Functions with DPDK ?
No
No
Can IOMMU be used ?
No
Is VFIO no iommu supported ?
No
Are legacy interrupts supported ?
No
Yes
Yes
Yes
SAFE
VFIO
UNSAFE
VFIO no iommu
uio_pci_generic
igb_uio

Figure 2.31 Choice of the Kernel Driver to Be Used With DPDK

DPDK and Host Hardware Architecture

NUMA

NUMA stands for Non-Uniform Memory Access systems and consists of a traditional server that has a single CPU, a single RAM, and a single RAM controller. RAM can be made of several DIMM banks in several sockets, all of which are associated to the CPU. When the CPU needs access to data in RAM, it requests it from its RAM controller. For example, in a server with two CPUs, each one can be a separate NUMA: NUMA0 and NUMA1, as shown in Figure 2.32.

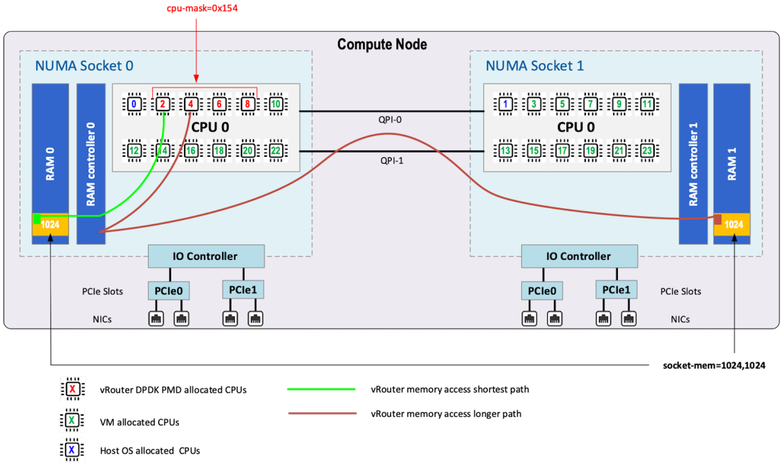


Figure 2.32 NUMA Overview

NUMA nodes architecture in Figure 2.32 shows:

- In green: CPU core accessing a memory item located in its own NUMA's RAM controller, showing minimum latency.
- In red: CPU core accessing a memory item located in the other NUMA through the QPI (Quick Path Interconnect) path and the remote RAM controller, showing a higher latency.

When CPU0 needs to access data located in RAM0, it will go through its local RAM controller 0. The same thing happens for CPU1.

When CPU0 needs to access data located in the other RAM1, the first (local) controller 0 has to go through the second (or remote) RAM controller 1, which will access the (remote) data in RAM 1. Data will use an internal connection between the two CPUs called QPI, or Quick Path Interconnect, which is typically of a high enough capacity to avoid being a bottleneck, typically one or two times 25GBps (400 Gbps). For example, the Intel Xeon E5 has two CPUs with two QPI links between them; Intel Xeon E7 has four CPUs, with a single QPI between pairs of CPUs.

The fastest RAM that the CPU has access to is the register, which is inside the CPU and reserved for it. Beyond the register, the CPU has access to cached memory, which is a special memory based on higher performance hardware (see Figure 2.33).

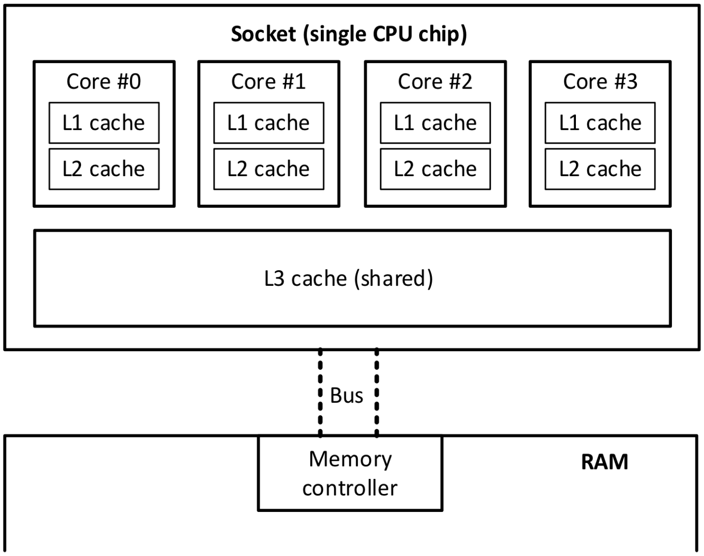


Figure 2.33 Numa Nodes Overview

Cached memories are shared between the cores of a single CPU. Typical characteristics of memory cache are:

- Accessing a Level 1 cache takes 7 CPU cycles (with a size of 64KB or 128KB).
- Accessing a Level 2 cache takes 11 CPU cycles (with a size of 1MB).
- Accessing a Level 3 cache takes 30 CPU cycles (with a larger size).

If the CPU needs to access data that is in the main RAM, it has to use its RAM controller.

Access to RAM typically takes 170 CPU cycles (the green line in Figure 2.32). Access to the remote RAM through the remote RAM controller typically adds 200 cycles (the red line in Figure 2.32), meaning RAM latency is roughly doubled.

When data needed by the CPU is located in both the local and the remote RAM with no particular structure, latency to access data can be unpredictable and unstable.

Hyper-threading (HT)

A single physical CPU core with hyper-threading appears as two logical CPUs to an operating system. While the operating system sees two CPUs for each core, the actual CPU hardware has only a single set of execution resources for each core. Hyper-threading allows the two logical CPU cores to share physical execution resources.

The sharing of resources allows two logical processors to work with each other more efficiently and allows a logical processor to borrow resources from a stalled logical core (assuming both logical cores are associated with the same physical core). Hyper-threading can help speed up processing, but it's nowhere near as good as having actual additional cores.

Huge Pages

Memory is managed in blocks known as pages. On most systems, a page is 4KB, and 1MB of memory is equal to 256 pages; 1GB of memory is 256,000 pages, etc. (See Figure 2.34.) CPUs have a built-in memory management unit that manages a list of these pages in hardware.

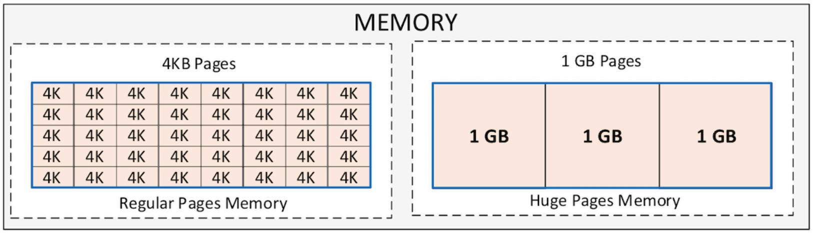


Figure 2.34 Huge Pages Memory Allocation

The Translation Lookaside Buffer (TLB) is a small hardware cache of virtual-to-physical page mappings. If the virtual address passed in a hardware instruction can be found in the TLB, the mapping can be determined quickly. If not, a *TLB miss* occurs, and the system falls back to slower, software-based address translation. This results in performance issues. Since the size of the TLB is fixed, the only way to reduce the chance of a TLB miss is to increase the page size.

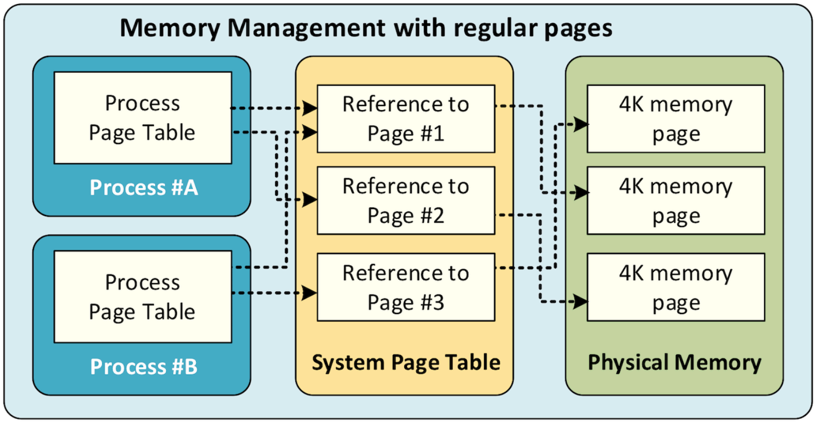


Figure 2.35 *Huge Pages Memory Management*

Virtual memory address lookup slows down when the number of entries increases.

A huge page is a memory page that is larger than 4KB. (See Figure 2.34.) In x86_64 architecture, in addition to standard 4KB memory page size, two larger page sizes are available: 2MB and 1GB. Contrail DPDK vRouter can use both or only one huge page size.

CPU Isolation and Pining

An operating system uses a scheduler to place each single process and/or threads it has to run onto one of the CPUs offered by a host. There are two kinds of scheduling, *cooperative* and *preemptive*. By default, Linux scheduler uses a cooperative mode.

In order to get a CPU booked for a subset of tasks, you have to inform the OS scheduler not to use these CPUs for all the tasks it has to run. These CPUs are "*isolated*" because they are no longer used by the OS to process all the tasks. In order to get a CPU isolated, several mechanisms can be used:

- remove the CPU from the common CPU list used to process all tasks,
- change the scheduling algorithm (cooperative to pre-emptive), or

- participate or do not interrupt processing.

Isolation and pinning are two complementary mechanisms that are proposed by Linux OS:

- CPU isolation restricts the set of CPUs that are available for operating system scheduler level. When a CPU is isolated, no task will be scheduled on it by the OS. An explicit task assignment must be completed.
- CPU pinning is also called processor affinity and it enables the binding and unbinding of a process or a thread onto the CPU. On the opposite side, CPU pinning is a mechanism that consists in defining a limited set of CPUs that are allowed to be used by:
  - The OS scheduler. OS CPU affinity is managed through system.
  - A specific process: using CPU pinning rules (taskset command for instance).

Tasks to be run by an operating system must be spread across available CPUs. These tasks are in a multi-threading environment often made of several processes which are also made of several threads.

CPU Isolation Mechanisms

Isolcpus

Isolcpus is a kernel scheduler option. When a CPUs is specified in isolcpus list, it is removed from the general kernel SMP balancing and scheduler algorithms. The only way to move a process onto or off of an isolated CPU is via the CPU affinity syscalls (or to use the taskset command).

This isolation mechanism:

- removes isolated CPUs from the common CPU list used to process all tasks
- changes the scheduling algorithm from cooperative to pre-emptive
- performs CPU isolation at the system boot

Isolcpus is suffering lots of drawbacks:

- it requires manual placement of processes on isolated CPUs.
- it is not possible to rearrange the CPU isolation rules after the system startup.
- the only way to change an isolated CPU list is by rebooting with a different isolcpus value in the boot loader configuration (GRUB for instance).
- isolcpus disables the scheduler load balancer for isolated CPUs. This also means the kernel will not balance those tasks equally among all the CPUs sharing the same isolated CPUs (having the same affinity mask).

CPU Shield

The cgroups subsystem proposes a mechanism to dedicate some CPUs to one or several user processes. It consists of defining a user shield group which protects a subset of CPU system tasks.

The definition of three cpusets consists of:

- root: present in all configurations and contains all CPUs (unshielded)
- system: containing CPUs used for system tasks - the ones which need to run but aren't important (unshielded)
- user: containing CPUs used for tasks that you want to assign a set of CPUs for their exclusive use (shielded)

The CPU shield can be manipulated with the `cset shield` command.

Tuned

Tuned is a system tuning service for Linux. Tuned uses Tuned profiles to describe Linux OS performance tuning configuration.

The cpu-partitioning profile partitions the system CPUs into isolated and housekeeping CPUs. This profile is intended to be used for latency-sensitive workloads.

NOTE Currently, Tuned is only supported on Linux Red Hat OS family. See : https://tuned-project.org/.

Linux systemd - System Task CPU Affinity

A thread's CPU affinity mask determines the set of CPUs on which it is eligible to run. Linux systemd is a software suite that provides an array of system components for Linux operating systems. Its primary component is an init system used to bootstrap user space and manage user processes. The CPU affinity parameter restricts all processes spawned by systemd to the list of cores defined by the affinity mask.

Default CPU Affinity

When run as a system instance, systemd interprets the configuration file /etc/systemd/system.conf. In this configuration file, CPU affinity variable configures the CPU affinity for the service manager as well as the default CPU affinity for all forked off processes.

Per Service Specific CPU Affinity

Individual services may override the CPU affinity for their processes with the CPU affinity setting in unit files:

```
# vi /etc/systemd/system/<my service>.service
...
[Service]
CPUAffinity=<CPU mask>
```

If a specific CPU affinity has been defined for a given service, it has to be restarted in order for the new configuration file to be taken into consideration.

CPU Assignment for User Processes (taskset)

Taskset is used to set or retrieve the CPU affinity of a running process given its PID, or to launch a new command with a given CPU affinity. You can retrieve the CPU affinity of an existing task by:

```
# taskset -p pid
```

Or set it:

```
# taskset -p mask pid
```

Bind a Virtual NIC to DPDK

DPDK requires a direct NIC access into user space. Virtio vhost-user backend exposes the Virtio network device in user space. The vhost-user is a library that implements the vhost protocol in user space. The vhost-user library allows you to expose a Virtio backend interface into user space.

The vhost-user library defines the structure of messages that are sent over a UNIX socket to communicate with the Virtio net device backend (vhost-net kernel driver is using ioctls instead)(see Figure 2.36).

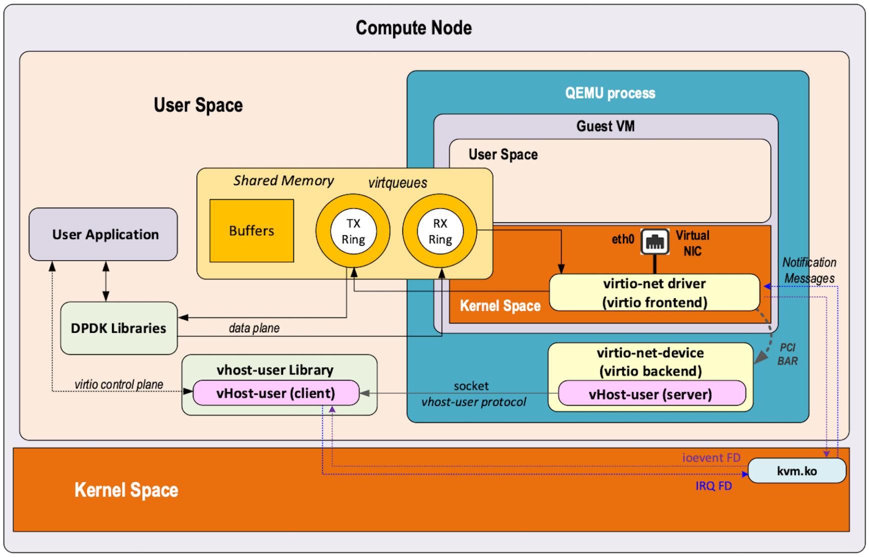


Figure 2.36 Vhost-user Library

You can see the kernel-mode VM connected to a DPDK compute application. The user application is using both:

- the vhost user library: for emulated PCI NIC control plane
- the DPDK libraries: for emulated PCI NIC data plane

Support for user space vhost has also been provided with QEMU 2.1.

Run DPDK in a Guest VM

Virtual IOMMU

Virtual IOMMU (vIOMMU) allows emulation of the IOMMU for guest VMs.

vIOMMU has the following characteristics:

- Translates guest VM I/O Virtual Addresses (IOVA) to Guest Physical Addresses (GPA).
- Guests VM Physical Addresses (GPA) are translated to Host Virtual Addresses (HVA) through the hypervisor memory management system.
- Performs device isolation.
- Implements a I/O TLB (Translation Lookaside Buffer) API which exposes memory mappings.

In order to get a virtual device working with a virtual IOMMU you have to create the needed IOVA mappings into the vIOMMU and configure the device's DMA with the IOVA.

The following mechanisms can be used to create vIOMMU memory mappings:

- Linux Kernel's DMA API for kernel drivers
- VFIO for user space drivers (see Figure 2.37)

The integration between the virtual IOMMU and any user space network application like DPDK is usually done through the VFIO driver. This driver performs device isolation and automatically adds the memory (IOVA to GPA) mappings to the virtual IOMMU.

The use of huge pages memory in DPDK contributes to optimize TLB lookups, since fewer memory pages can cover the same amount of memory. Consequently, the number of device TLB synchronization messages drops dramatically. Hence, the performance penalty from TLB lookups is lowered, see: https://www.redhat.com/en/blog/journey-vhost-users-realm, and https://wiki.qemu.org/Features/VT-d.

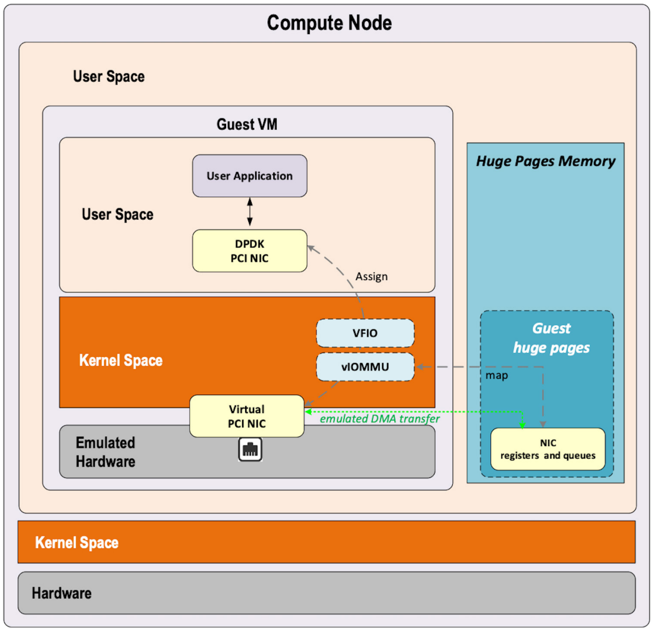


Figure 2.37 VFIO for User Space Drivers

Virtio Poll Mode Driver

Virtio-pmd driver, is a DPDK driver, built on the PMD abstraction, that implements the virtio protocol, see Figure 2.38.

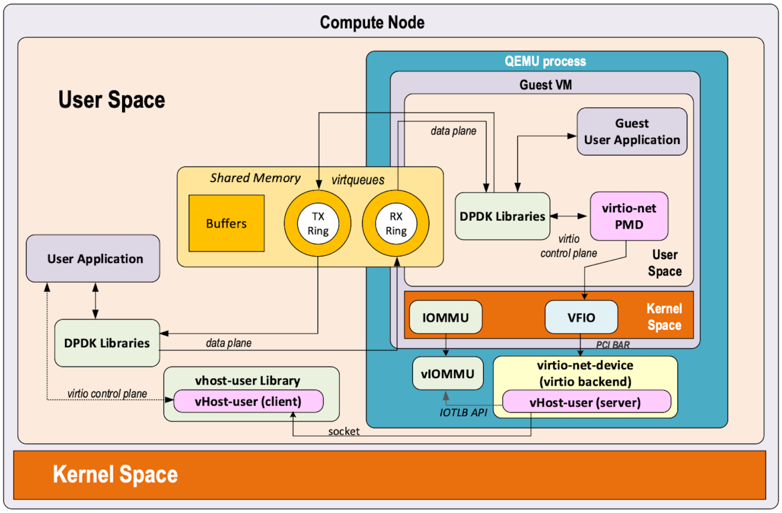


Figure 2.38 VFIO for User Space Drivers

Vhost user protocol moves the virtio ring from kernel all the way to user space. The ring is shared between the guest and DPDK application. QEMU sets up this ring as a control plane using UNIX sockets.

If both the host server and the guest VM are DPDK, there are no VMExits in the host for guest packets processing. Guest virtual machines use the virtio-net PMD driver, which performs packets polling. There is nothing running in kernel here, so there are no system calls. Since both system calls and VM Exits are avoided, the performance boost is significant.

Physical Network Device Assignment (VFIO) and PCI Passthrough

A PCI device can be assigned to a guest in order to be used by a guest DPDK application. One of the host physical NICs is dedicated and exposed to this guest in order to get direct access to the physical network. This assigned NIC also needs to get direct access to the host memory.

When using PCI passthrough mechanism, the NIC has a full access to the host memory. With VFIO mechanism, this direct host memory access from a NIC device is restricted to the host memory space area specifically allocated for the data communication purpose with this NIC. This access restriction is provided by both VFIO and IOMMU.

By leveraging the VFIO driver in the host kernel we can provide a direct access to an assigned physical NIC from this guest protected by IOMMU. IOMMU protects host memory against malicious or bug writes, which can corrupt host memory at any time.

We have the same concern for a Virtual Machine. When a physical device is assigned to a guest VM, the guest memory address space is totally exposed to the hardware PCI device. A similar mechanism is also needed to protect the guest memory address space; this mechanism is provided by both VFIO and vIOMMU as shown in Figure 2.39.

By leveraging the VFIO driver also in the guest kernel we can provide a direct access to the assigned physical memory from the guest user space. Then vIOMMU provides a secure mechanism to manage DMA transfer between an assigned physical hardware and hosted guest virtual instance memory area.

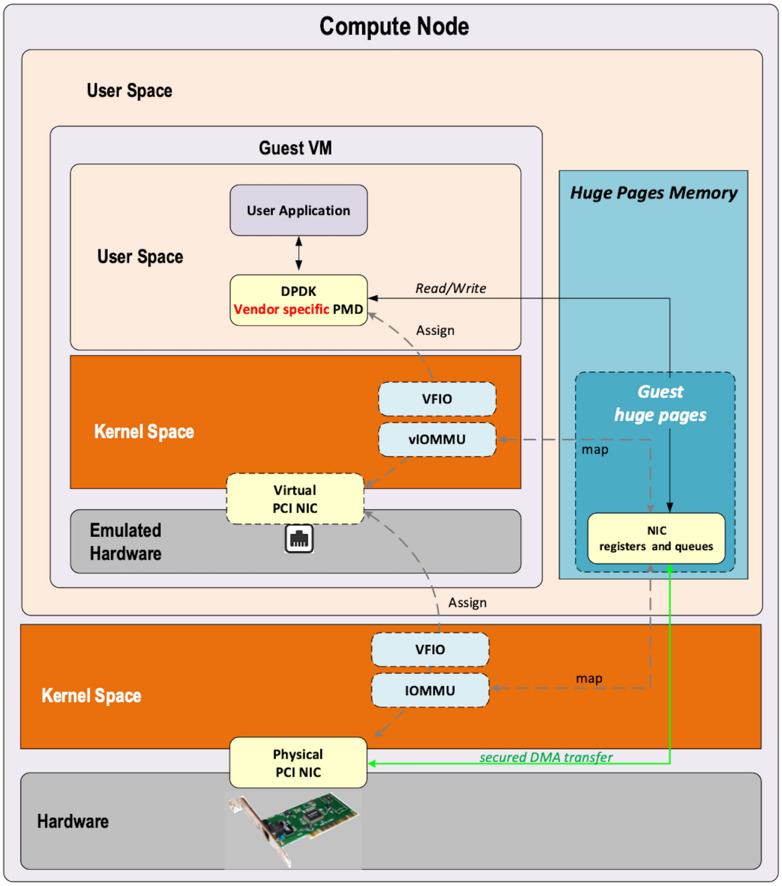


Figure 2.39 Physical Device Assigned to a Guest VM protected by both IOMMU and vIOMMU

SRIOV and DPDK in Guest VM

This use case is almost the same as PCI passthrough but VFIO and IOMMU are used to expose a SRIOV virtual function directly to a guest VM.

An additional physical function driver, which is vendor specific, is used to manage the virtual function creation on the physical NIC. This driver is used by a VMM (like libvirt) to create the virtual function before the virtual instance is spawned, see Figure 2.40.

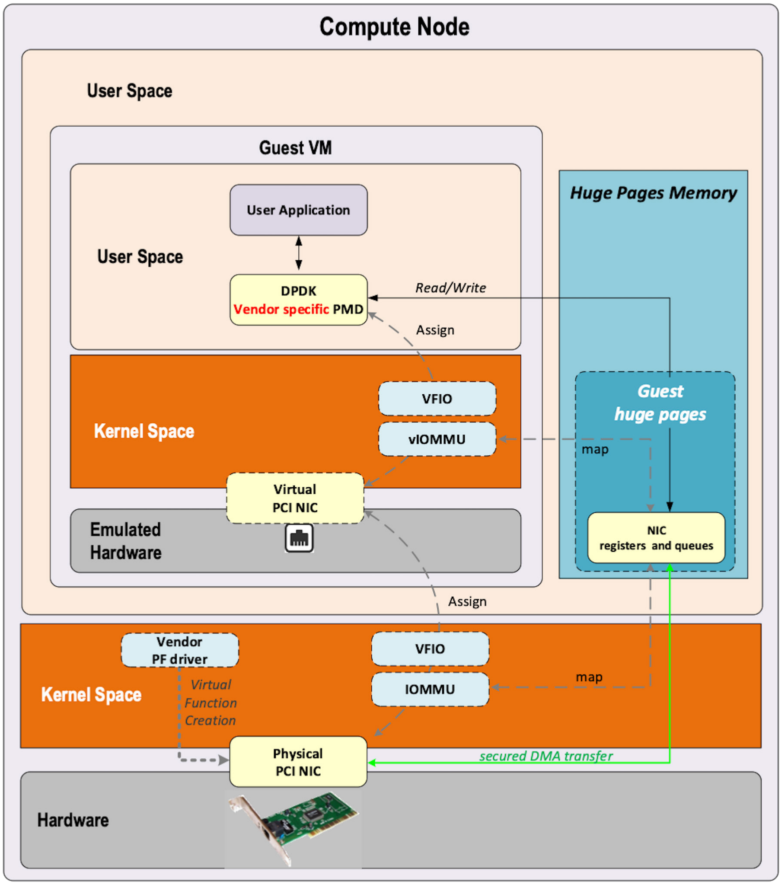


Figure 2.40 SRIOV and IOMMU Protection

Physical incoming packets are directly copied in the guest memory without involving the host server. SR-IOV only allows sharing a physical NIC between several guests, creating Virtual Function dedicated to a single guest; but does not change the packet processing path provided by PCI passthrough mechanism.

By leveraging the VFIO driver in the host kernel we can provide a direct access to an assigned SRIOV virtual function, with the guest memory protected by IOMMU (Figure 2.40).

Virtio-assisted Hardware Acceleration

With DPDK and virtio we have a technology that is allowing to get network virtualization at a high speed. This is a key technology for the SDN data plane. But this packet processing model has still some drawbacks, mainly:

- DPDK requires isolating some host CPUs for its exclusive use, meaning less CPU resources for the user application.
- Compute CPUs are generic and are not optimized for packet processing. DPDK is requiring lots of CPU usage to provide both feature rich and performance virtual network (host compute for DPDK vRouter/vSwich application and on guest VMs for DPDK end-user applications).

SR-IOV adds performance but its use is limited in SDN applications due to its direct path between the guest VM and the NIC hardware that bypasses the host operating system in which the SDN network function is running (vSwitch and vRouter).

In the coming sections, we will describe some evolving features on both the Virtio and direct device assignment to provide a solution that:

- is running in user space, like proposed by DPDK
- has good hardware performance, like that proposed by SRIOV and direct physical device assignment
- can be used in SDN, like proposed by Virtio software solution.

Virtio Full Offloading

With virtio full hardware offloading, both the Virtio data plane and Virtio control plane are offloaded to the NIC hardware. The physical NIC must support:

- the virtio control specification: discovery, feature negotiation, and establishing/terminating the data plane.
- the virtio data plane specification: virtio ring layout.

Hence once the guest memory is mapped with the NIC using Virtio physical device passthrough, the guest communicates directly with the NIC via PCI without involving any specific drivers in the host kernel.

Guest VM packet processing is directly performed in NIC hardware but presented to the guest instance like a regular Virtio-emulated interface. Guest VM does not make any difference between a Virtio-emulated interface and an assigned physical Virtio NIC, as they are exposed with the same Virtio driver frontend in the guest. See Figure 2.41.

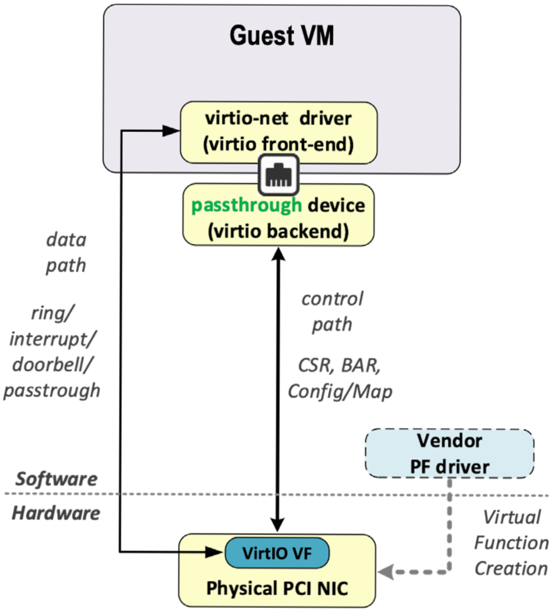


Figure 2.41 Virtio Full Hardware Offloading

Virtio Device Passthrough

Virtio device passthrough can be implemented on a NIC that supports or doesn't support SR-IOV. Like other physical device assignment technology presented in this book, VFIO and IOMMU are used to present the physical device NIC into the guest VM user space in a secured way. See Figure 2.42.

Hence, such a Virtio physical NIC can be used by a DPDK application running in a virtual instance. But, like other Virtio device passthroughs, this has the same limitations for SDN. As the host operating system is totally bypassed by this mechanism, you cannot interconnect instances using such NIC interfaces with a SDN virtual router or switch.

The main advantage of Virtio device passthrough is the flexibility it provides for a virtual instance to transparently use either a real physical interface or an emulated one. It offers an open public specification, which supports devices fully independent of any specific vendor.

Virtio full HW offloading can support live migration thanks to Virtio, which is not possible to achieve without any specific implementation with SR-IOV. But in order to be able to support such a feature, the latest Virtio specifications (1.1 version) must be implemented onto both the QEMU and the NIC hardware used on the cloud infrastructure.

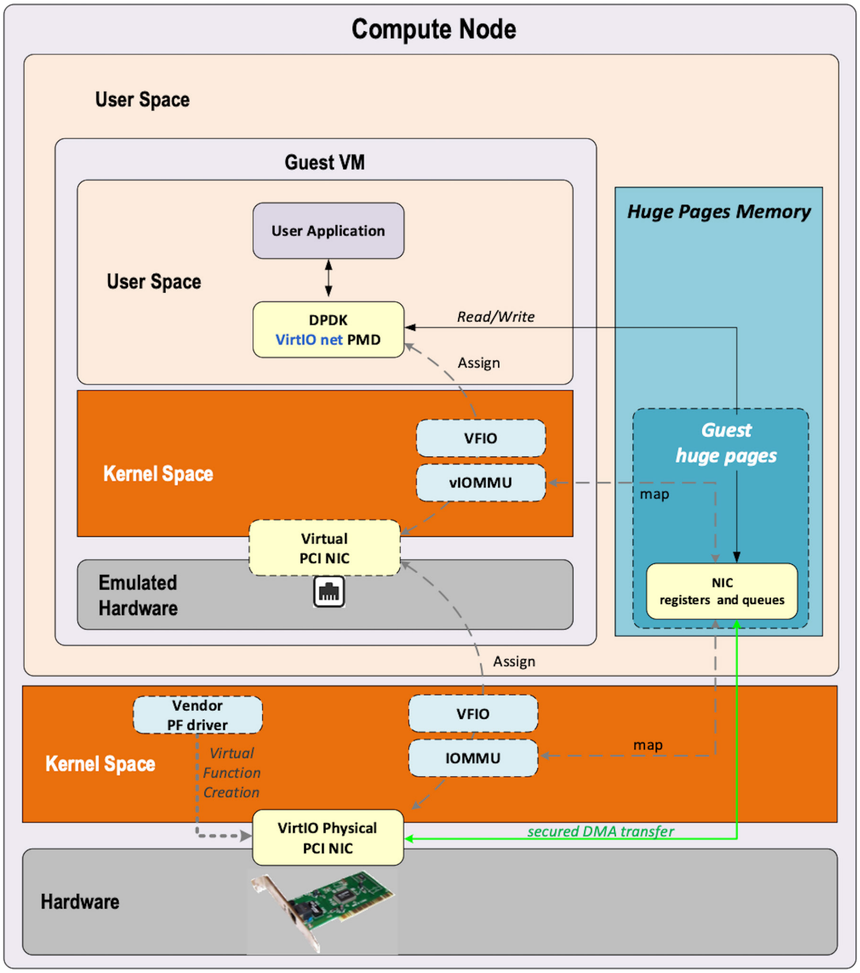


Figure 2.42 Virtio Device Passthrough

Virtio Datapath Acceleration

Like full hardware offloading, virtual Data Path Acceleration (vDPA) aims to standardize the physical data plane using the virtio ring layout and present a standard

Virtio driver in the guest decoupled from any vendor implementation for the control path.

vDPA presents a generic control plane through software which provides an abstraction layer on top of physical NIC. Like Virtio full hardware offloading, vDPA builds a direct data path between the guest network interface and the physical NIC, using the Virtio ring layout. But the control path a generic vDPA driver (mediation driver) is used to translate the vendor NIC driver/control plane to the Virtio control plane in order to allow each NIC vendor to keep using its own driver.

vDPA allows NIC vendors to support Virtio ring layout with smaller effort keeping wire speed performance on the data plane. See Figure 2.43.

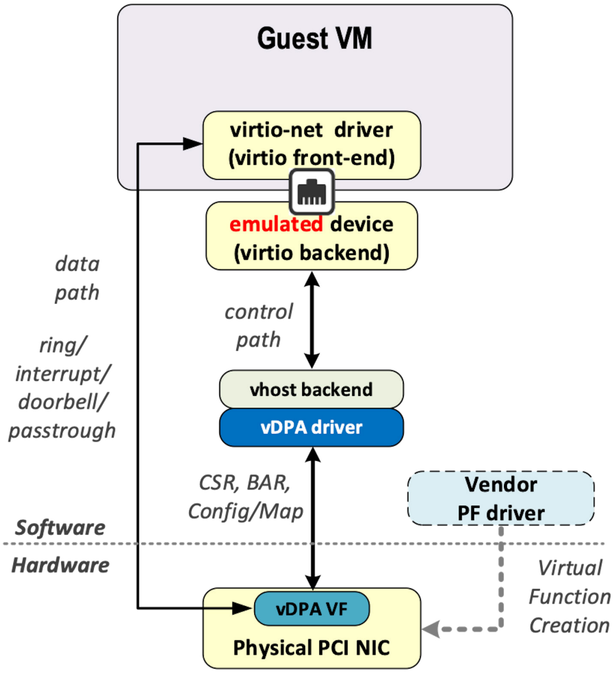


Figure 2.43 Virtual Data Path Acceleration

vDPA is requiring a vendor specific *mediation device driver* to be loaded in the host operating system as shown in Figure 2.44.

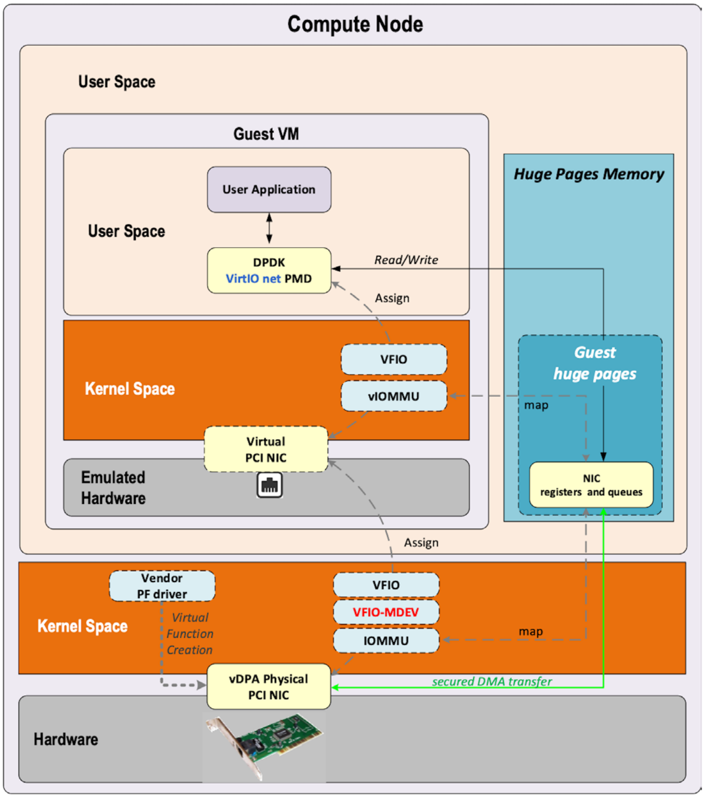


Figure 2.44 Mediation Device Driver Being Loaded in the Host OS

Smart NIC

A NIC card generator, commonly named smart NIC, is highly customizable thanks to the last evolution provided by some new capabilities (FPGA, ARM, P4). It's now possible to envisage SDN vSwitch/vRouter data plane functions to be moved onto the NIC card keeping only the control plane function in the host operating system.

For Contrail solutions, this is made by offloading several Contrail vRouter tables including:

- Interface Tables
- Next Hop Tables

- Ingress Label Manager (ILM) Tables
- IPv4 FIB
- IPv6 FIB
- L2 Forwarding Tables
- Flow Tables

It permits accelerating lookups and forwarding actions that are directly performed into the NIC, as shown in Figure 2.45.

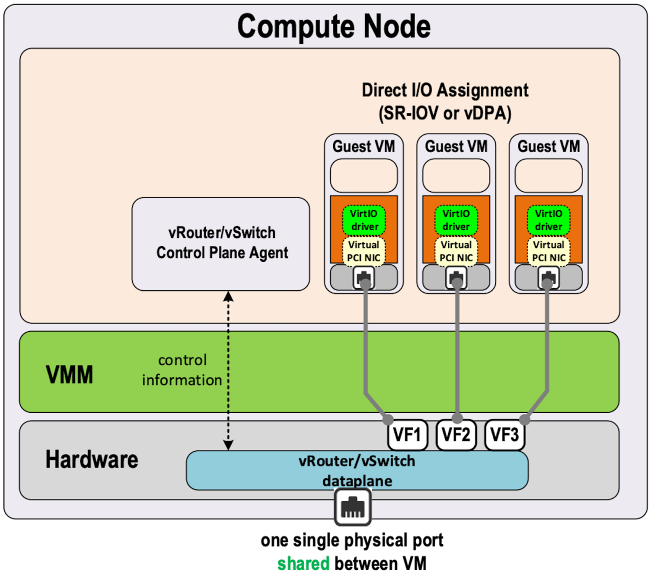


Figure 2.45 vRouter Data Plane in Smart NIC

You can see in Figure 2.45 that SDN packet processing is fully completed on the NIC card and no more host CPU processing is involved in packet processing.

Two implementations are proposed by Netronome are SRIOV + SmartNIC, and XVIO + Smart NIC, as shown in Figures 2.46 and 2.47.

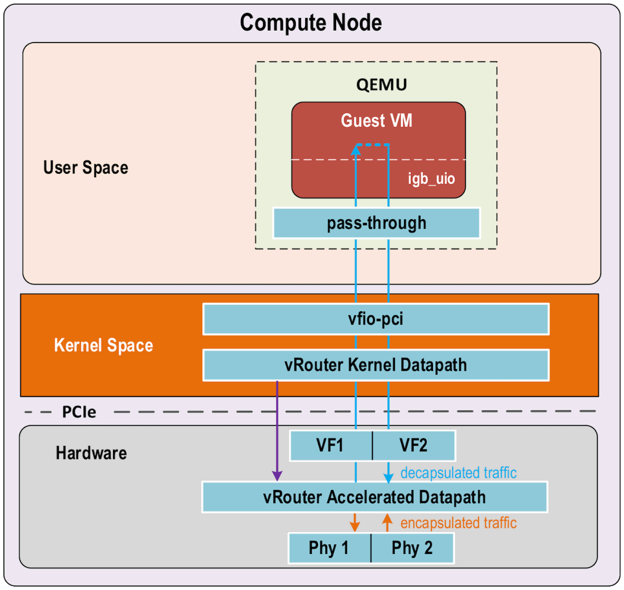
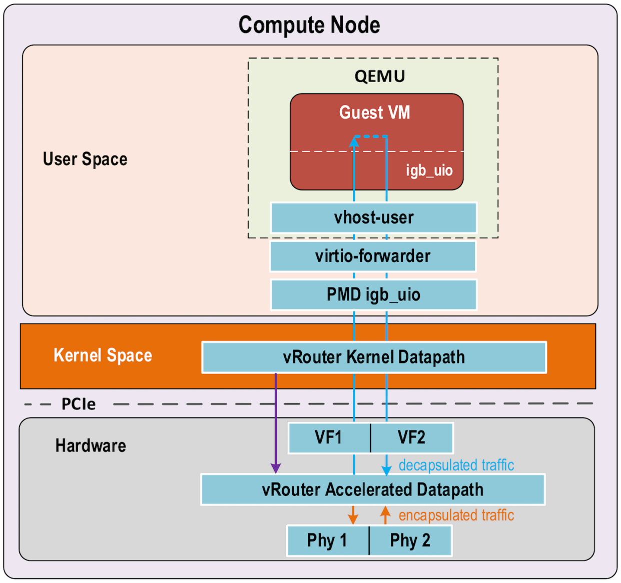


Figure 2.46 SRIOV + SmartNIC



Figuree 2.47 XVIO + Smart NIC

eBPF and XDP

Berkeley Packet Filter (BPF) was designed for capturing and filtering network packets that matched specific rules. In the past few years extended BPF (eBPF) was designed to take advantage of new hardware (64 bits usage for instance). An eBPF program is attached to a designated code path in the kernel.

eXpress Data Path (XDP) uses eBPF to achieve high-performance packet processing by running eBPF programs at the lowest level of the network stack, immediately after a packet is received by XDP. See Figure 2.48.

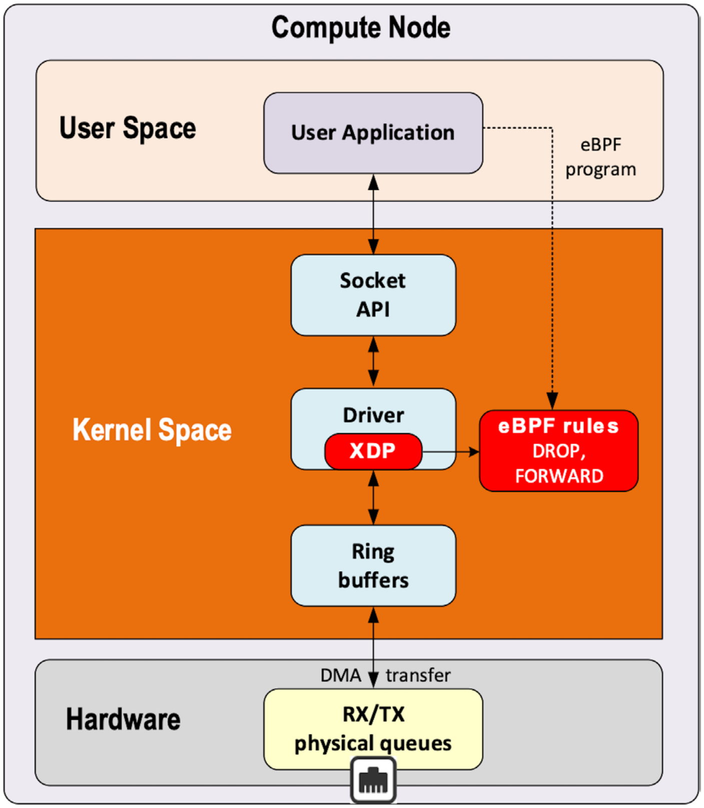


Figure 2.48 *eBPF and XDP*

XDP support is made available in the Linux kernel since version 4.8, while eBPF is supported in the Linux kernel since version 3.18.

XDP requires:

- MultiQ NICs
- Common protocol-generic offloads:
  - TX/RX checksum offload
  - Received Side Scaling
  - Transport Segmentation offload (TSO)

XDP packet processor performs:

- In kernel RX packets processing
- Process RX packets directly (without any additional memory allocation for software queue, nor socket buffer allocation)
- Assign one CPU to each RX queue. This CPU can be configured into poll mode or interrupt mode.
- Trigger BPF program for packet processing

BPF programs:

- parse packets
- perform table lookup
- manage stateful filters
- manipulate packets (encapsulation, decapsulation, NAT, …)

BPF program main actions are:

- Forward
- Forward after modification (NAT)
- Drop
- Normal receive (regular Linux packet processing with socket buffer and TCP/IP stack)
- Generic Receive Offload (coalesce several received packets of a same connection)

XDP is also able to offload an eBPF program to a NIC card which supports it, reducing the CPU load. See Figure 2.49.

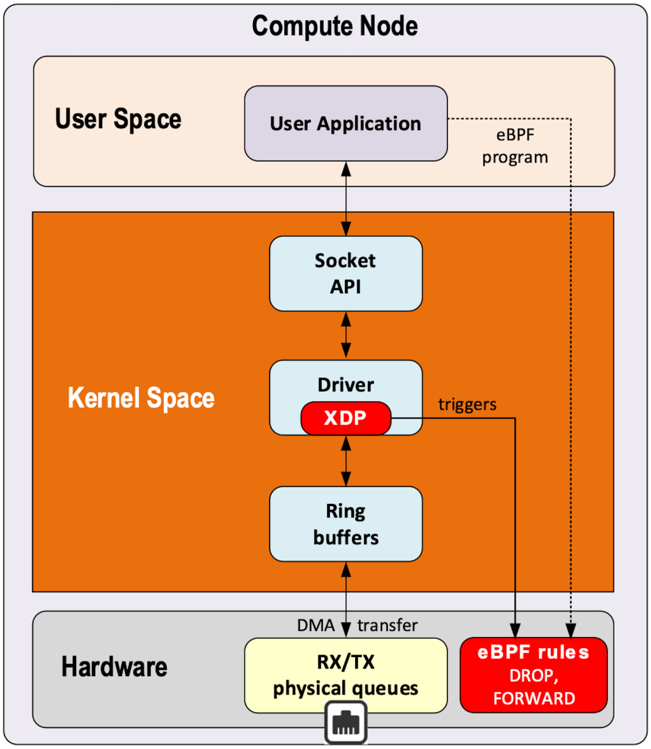


Figure 2.49 XDP Offloads eBPF Programs to a NIC Card

XDP and eBPF do not require:

- allocating large pages
- allocating dedicated CPUs
- choosing packet polling or interrupting driven networking model
- user space to kernel space context switching to perform eBPF filtering
- allowing packet processing offload when supported by used NIC card

NOTE eBPF rules are also supported in DPDK application. See: https://www.redhat.com/en/blog/using-express-data-path-xdp-red-hat-enterprise-linux-8.

NIC Virtualization Solutions Summary

We've seen lots of NIC virtualization models for virtual instances, from a full software implementation like that proposed by Virtio to fully hardware assisted solutions, like that proposed by SR-IOV. Also, DPDK provides the ability to move NIC packet processing from kernel space to user space.

Figure 2.50 provides an overview of the NIC virtualization solution:

- Full software solutions are very flexible and fit well with SDN and cloud feature expectation (such as live migration, east-west traffic inside host computes).
- Hardware-assisted solutions perform well but fit less with expected virtualization flexibility. Guest VM migration is poorly supported due to hardware dependencies. These solutions fit well with applications requiring a huge north-south traffic (from guest WM to cloud outside).

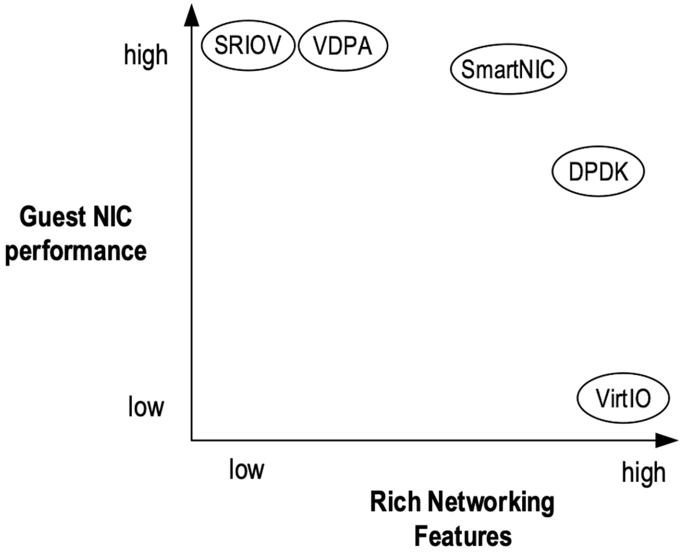
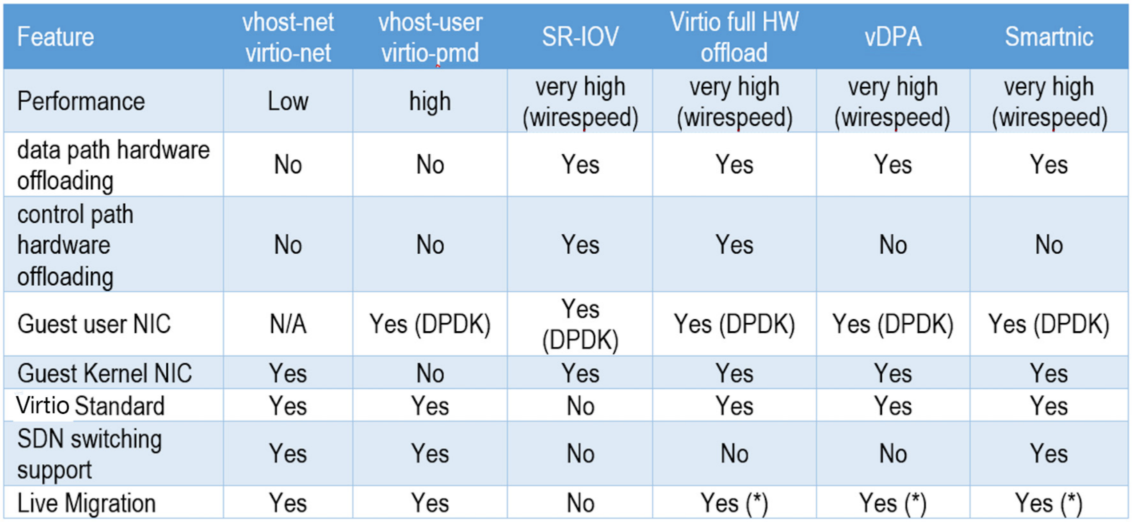


Figure 2.50 Overview of the NIC Virtualization Solution

In the middle of this matrix chart, SmartNIC and DPDK offer the best compromise for SDN usage. Smart NICs propose very high performance, but this is still not a fully mature solution (lots of implementations vendor specific, but there is no agreed standards). Figure 2.51 lists the feature sets of this chapter's examinations.



Feature	vhost-net virtio-net	vhost-user virtio-pmd	SR-IOV	Virtio full HW offload	vDPA	Smartnic
Performance	Low	high	very high (wirespeed)	very high (wirespeed)	very high (wirespeed)	very high (wirespeed)
data path hardware offloading	No	No	Yes	Yes	Yes	Yes
control path hardware offloading	No	No	Yes	Yes	No	No
Guest user NIC	N/A	Yes (DPDK)	Yes (DPDK)	Yes (DPDK)	Yes (DPDK)	Yes (DPDK)
Guest Kernel NIC	Yes	No	Yes	Yes	Yes	Yes
Virtio Standard	Yes	Yes	No	Yes	Yes	Yes
SDN switching support	Yes	Yes	No	No	No	Yes
Live Migration	Yes	Yes	No	Yes (*)	Yes (*)	Yes (*)

(*): depends on hardware and QEMU latest virtio specification support on the NIC card.

Figure 2.51 Feature Set of This Chapter's Examinations

Chapter 3

Contrail DPDK vRouter Architecture

Contrail Software Stack

Contrail is a SDN platform that provides virtual networking for overlay workloads like VMs and Containers. It consists of two components:

- Contrail Controller
- Contrail vRouter

Contrail Controller is a logically centralized but physically distributed SDN controller that is responsible for providing the management, control, and analytics functions for the whole cluster.

Figure 3.1 shows a high-level depiction of the Contrail architecture.

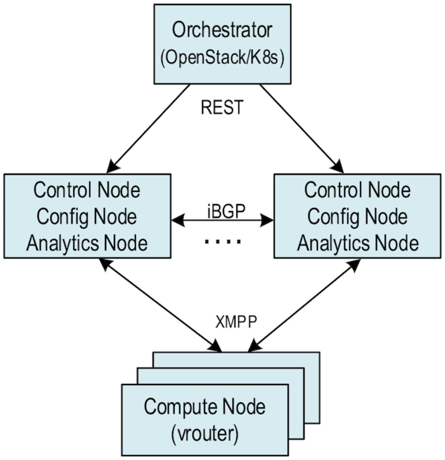


Figure 3.1 Contrail Architecture

You can see there is an orchestrator at the top of Figure 3.1 that can be OpenStack or Kubernetes. Below that, there are controller components like control node, config node, and analytics node. At the bottom are the compute nodes. Compute nodes are general purpose x86 servers and they are the main focus of this chapter.

Contrail Compute Node

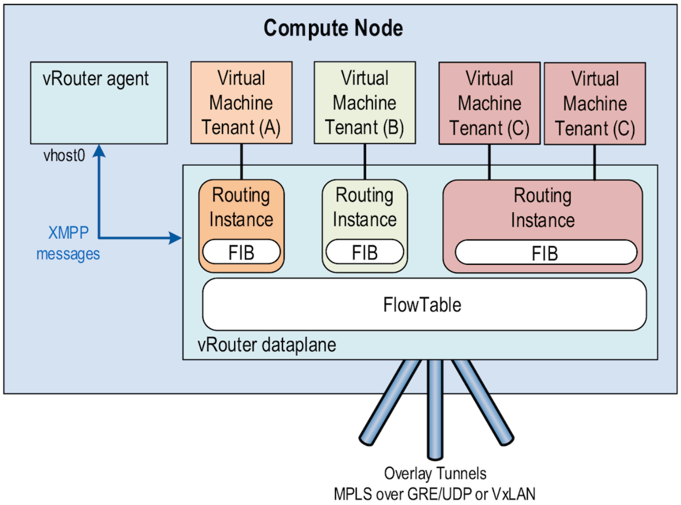


Figure 3.2 Contrail Compute Node

Figure 3.2 shows a more detailed view of the compute node. This is the place where vRouter runs. It is the most important component of the Contrail data plane. You can see some workloads running, and they can be either VMs or containers. These workloads have their interfaces plumbed into the vRouter.

At a high level, vRouter forms dynamic overlay tunnels with other workloads running on the same or different computes to send and receive data traffic. Within the server, it switches the packets between the VM interfaces and the physical interfaces after doing the required encapsulations or decapsulations. Currently, the encapsulation protocols supported by vRouter are MPLS over UDP (MPLSoUDP), MPLS over GRE (MPLSoGRE), and VXLAN. Each of these workloads have a corresponding forwarding state or routing instance inside vRouter which it uses to switch the packets. The physical interface that is connected to the top-of-rack (TOR) switch can be single or bonded mode.

The vRouter itself can be running either as a Linux kernel module or as a user space DPDK process. There is a vRouter agent process also running in user space. The agent has a connection to the controller using a XMPP channel, which is used to download configurations and forwarding information. The main job of the agent is to program this forwarding state to the vRouter forwarding plane.

vRouter Architecture

The vRouter is the workhorse of the Contrail system. Each and every packet to and from the Contrail cluster goes through the vRouter. The vRouter is high performance, efficient, and has the capability to process millions of packets per second. It is multi-threaded, multi-cored, and multi-queued to achieve maximum parallelism and exploit the x86 hardware to the maximum extent.

To support the rich and diverse features, vRouter has a sophisticated packet processing pipeline. The same pipeline can be stitched by the vRouter agent process from the simplest to the most complicated manner depending on the treatment that needs to be given to a packet. vRouter maintains multiple instances of forwarding bases. All the table accesses and updates use RCU (Read Copy Update) locks.

vRouter Interfaces

Figure 3.3 depicts the vRouter and its interfaces. Each of these vRouter interfaces is called a *vif* or vRouter interface. There are interfaces to each of the workloads (VM1, VM2, VMn) that it manages. These are typically tap interfaces.

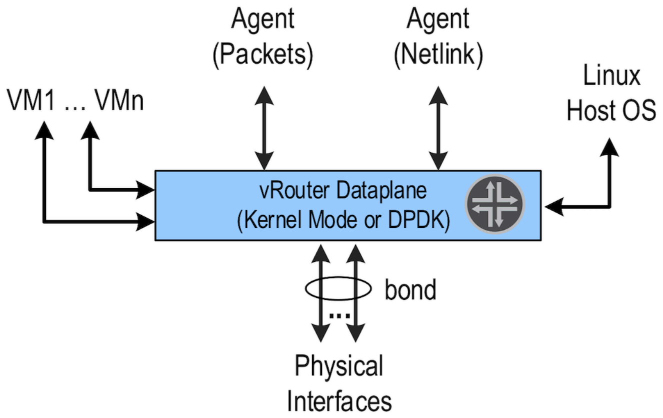


Figure 3.3 The vRouter Interfaces

To send packets to other physical servers or switches, vRouter uses the physical interfaces. They can be single or bonded NICs. vRouter is only interested in overlay packets or the packets to and from the workloads. For other packets, it uses the Linux interface to send them to the host OS.

This Linux interface is called vhost0. It also has Netlink interfaces toward the vRouter agent to download the forwarding state and also to send and receive some exception packets. The name of the later is called *pkt0 interface*.

This is sample output from the `vif --list` command, which provides the list of all vifs that are configured on a compute node:

```
[root@a7s3 ~]# vif --list
Vrouter Interface Table

Flags: P=Policy, X=Cross Connect, S=Service Chain, Mr=Receive Mirror
      Mt=Transmit Mirror, Tc=Transmit Checksum Offload, L3=Layer 3, L2=Layer 2
      D=DHCP, Vp=Vhost Physical, Pr=Promiscuous, Vnt=Native Vlan Tagged
      Mnp=No MAC Proxy, Dpdk=DPDK PMD Interface, Rfl=Receive Filtering Offload, Mon=Interface is
Monitored
      Uuf=Unknown Unicast Flood, Vof=VLAN insert/strip offload, Df=Drop New Flows, L=MAC Learning Enabled
      Proxy=MAC Requests Proxied Always, Er=Etree Root, Mn=Mirror without Vlan Tag, HbsL=HBS Left Intf
      HbsR=HBS Right Intf, Ig=Igmp Trap Enabled

vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
            Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:16
            RX device packets:14117825256  bytes:2456433542438 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:02:00.0  Status: UP  Driver: net_ixgbe
            Slave Interface(1): 0000:02:00.1  Status: UP  Driver: net_ixgbe
            Vlan Id: 101  VLAN fwd Interface: vfw
            RX packets:7058889673  bytes:1199976475061 errors:0
            TX packets:7059332226  bytes:1200700918913 errors:0
            Drops:392133
            TX device packets:14119406674  bytes:2457969960530 errors:0

vif0/1      PMD: vhost0 NH: 5
            Type:Host HWaddr:90:e2:ba:c3:af:20 IPaddr:8.0.0.4
            Vrf:0 Mcast Vrf:65535 Flags:L3DEr QOS:-1 Ref:13
            RX device packets:815137  bytes:780115621 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:815137  bytes:780115621 errors:0
            TX packets:873131  bytes:162620313 errors:0
            Drops:12
            TX device packets:873131  bytes:162620313 errors:0

vif0/2      Socket: unix
            Type:Agent HWaddr:00:00:5e:00:01:00 IPaddr:0.0.0.0
            Vrf:65535 Mcast Vrf:65535 Flags:L3Er QOS:-1 Ref:3
            RX port   packets:135922 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:135922  bytes:11689292 errors:0
            TX packets:36432  bytes:3198966 errors:0
            Drops:0

vif0/3      PMD: tap41a9ab05-64 NH: 32
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104
            Vrf:3 Mcast Vrf:3 Flags:PL3L2DEr QOS:-1 Ref:12
            RX queue  packets:7057651439 errors:7736
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 7736 0
            RX packets:7057833621  bytes:875156312738 errors:0
            TX packets:7057123054  bytes:875068202430 errors:0
            ISID: 0 Bmac: 02:41:a9:ab:05:64
            Drops:7947

vif0/4      PMD: tapd2d7bb67-c1 NH: 29
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.0.104
            Vrf:2 Mcast Vrf:2 Flags:PL3L2DEr QOS:-1 Ref:12
            RX queue  packets:782831 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:799687  bytes:81599398 errors:0
            TX packets:1110661  bytes:85243244 errors:0
            ISID: 0 Bmac: 02:d2:d7:bb:67:c1
            Drops:1665
```

The different types of interfaces listed here are:

- vif0/0: the underlay NIC card (usually a Linux bond interface)
- vif0/1: the interface to the Linux operating system (vhost0)
- vif0/2: the interface to the vRouter agent (pkt0)
- vif0/3 and higher: the VM Interfaces (vNIC)

vRouter Packet Processing Pipeline

The vRouter packet processing pipeline is shown in Figure 3.4.

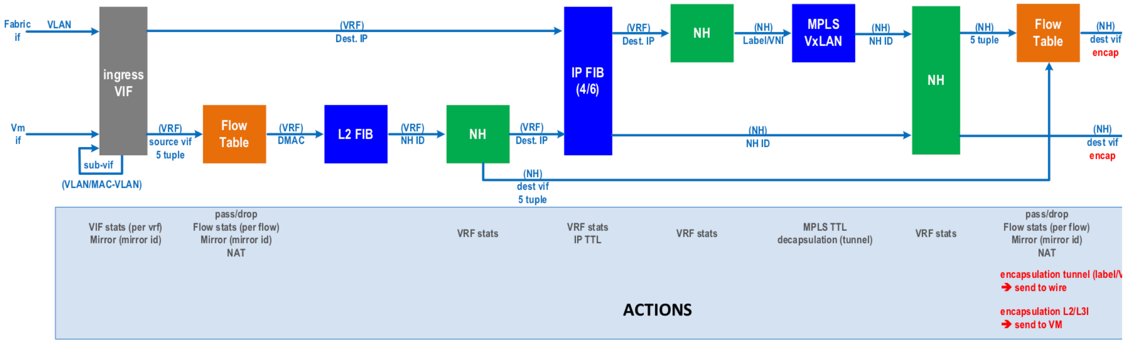


Figure 3.4 The vRouter Packet Processing Pipeline

There are various tables and engines in action in this pipeline. Some of the important tables are flow table, route table, NH table, and the MPLS/VXLAN table. The vRouter agent programs these tables based on the forwarding state it receives from the control node and also based on its own internal processing. Each packet, depending on which interface it is coming from, is subjected to the desired processing.

At a high level, all packets enter from a vif interface. The vifs are nothing but one of the vRouter interfaces described previously, for example: tap interface, physical interface, vhost0 interface, agent interface, etc. Depending upon the configuration of that interface, packets enter different pipeline stages, doing lookups in different tables based on what actions are defined in each stage, and the packets are modified accordingly.

At the end of the processing, it is sent to another vRouter interface or vif after encapsulation or decapsulation. This is a fairly generic pipeline and the agent stitches this based on the rich feature set that the Contrail cluster can configure.

Another important aspect of vRouter is that of forwarding modes. The vRouter can work in two modes - flow mode (bottom pipeline in Figure 3.4) or packet mode (top of Figure 3.4). By default, Contrail works in flow mode. This means that vRouter keeps track of every single flow traversing it. Depending on the flow action, it can either forward the packet or drop it. In the packet mode, the vRouter bypasses the flow table and directly uses the next hop for treatment that needs to be given to the packet. For example, if the next hop is a tunnel next hop, the packet is encapsulated in a tunnel header and forwarded to an outgoing interface.

vRouter Deployment Methods

Contrail supports three kinds of vRouter deployments.

Linux Kernel

In this deployment, vRouter is installed as a kernel module (vrouter.ko) inside the Linux OS, as seen in Figure 3.5. This is the default installation mode when configuring a compute node. vRouter registers itself with the Linux TCP/IP stack to get packets from any of the Linux interfaces that it wants to. It uses the netdev_rx_handler_register() API provided by Linux for this purpose. The interfaces can be bond, physical, tap (for VMs), veth (for containers) etc. It relies on Linux to send and receive packets from different interfaces. For example, Linux exposes a tap interface backed by a vhost-net driver to communicate with VMs. Once vRouter registers for packets from this tap interface, the Linux stack sends all the packets to it. To send a packet, vRouter just has to use regular Linux APIs like dev_queue_xmit() to send the packets out on a Linux interface.

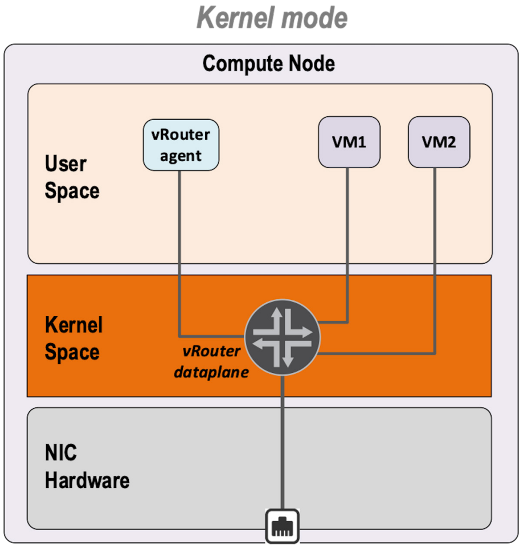


Figure 3.5 vRouter Running Inside Linux Kernel

NIC queues (either physical or virtual) are handled by Linux OS. With respect to packet processing performance, the tuning has to be done at that Linux OS level. See Figure 3.6.

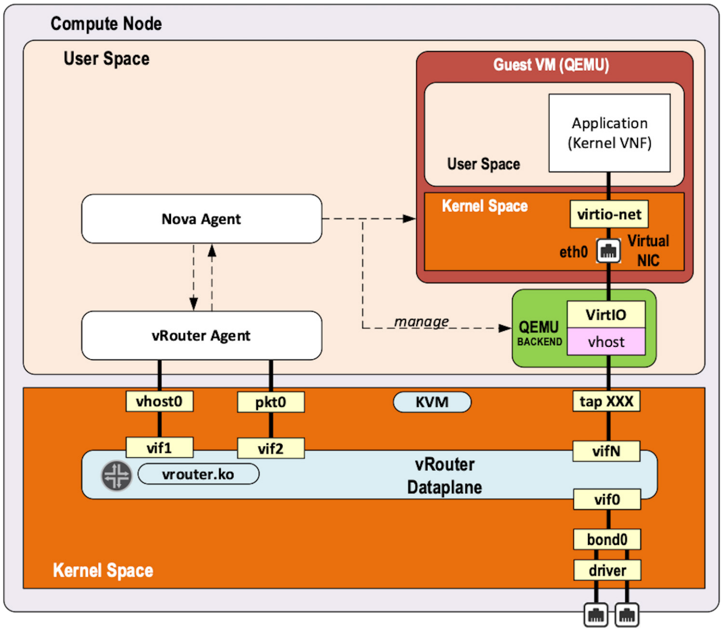


Figure 3.6 *Kernel vRouter Interfaces with Other Components In the Compute Node*

In Figure 3.6, packet processing works in interrupt mode. This mode generates interrupts, which results in lot of context switches. When the packet flow rate is low, it works well. But as soon as the packet rate starts increasing, the system gets overwhelmed with the number of interrupts generated, resulting in poor performance.

DPDK

In this mode, vRouter runs as a user space application that is linked to the DPDK library. This is the performance version of vRouter that is commonly used by telcos, where the VNFs themselves are DPDK-based applications. The performance of vRouter in this mode is more than ten times higher than the kernel mode. The physical interface is used by DPDK's poll mode drivers (PMDs) instead of Linux kernel's interrupt-based drivers.

A user-IO (UIO) kernel module like vfio or uio_pci_generic is used to expose a physical network interface's registers into user space so that they are accessible by DPDK PMD. When a NIC is bound to a UIO driver, it is moved from Linux kernel space to user space and therefore no longer managed nor visible by the Linux OS. Consequently, it is the DPDK application (which is the vRouter here) that fully manages the NIC. This includes packets polling, packets processing, and packets forwarding. No further action is taken by the operating system. All user packet processing steps are performed by the vRouter DPDK data plane. See Figure 3.7.

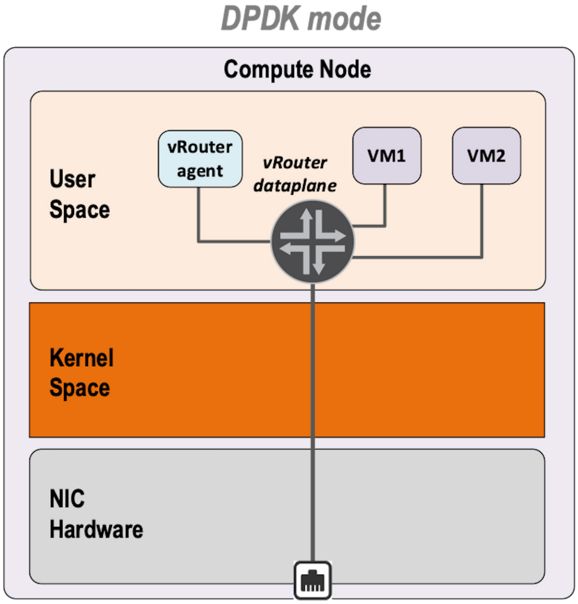


Figure 3.7 vRouter Running In DPDK Mode In Userspace

The nature of this "polling mode" makes the vRouter DPDK data plane packet processing/ forwarding much more efficient as compared to the interrupt mode when the packet rate is high. There are no interrupts and context switching during packet IO.

NOTE When the network packet rate is low, this way of working could be less efficient than the regular Kernel mode. In DPDK mode, a set of CPUs are fully dedicated for packet processing purposes and are always polling even in the absence of packets. If the network packets rate is too low, a lot of CPU cycle are unused and wasted. However, there is an inbuilt optimization technique that kicks in that yields the CPU for a small amount of time when there are no packets in the previous polling interval.

Finally, since the DPDK vRouter does not require any support from Linux kernel, it needs to be heavily tuned to get the best packet processing performance.

In this chapter we focus on the architecture of DPDK vRouter (see Figure 3.8).

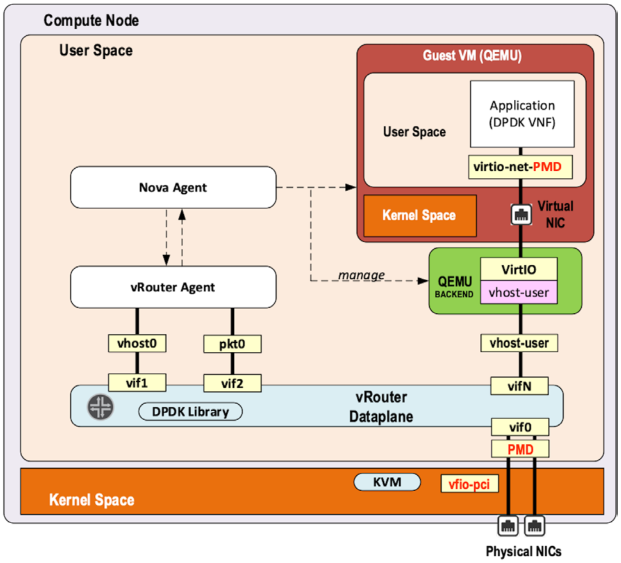


Figure 3.8 *DPDK vRouter Interfaces with Other Components In the Compute Node*

SmartNIC

In this mode, the Contrail vRouter runs inside the NIC card itself (SmartNIC) as shown in Figure 3.9. This means, compute host resources are not involved in packet processing. It saves the CPU resources that will be used by vRouter for packet processing. Since all the packet processing is done by the NIC hardware, the performance is the best compared to the previous two types of deployments.

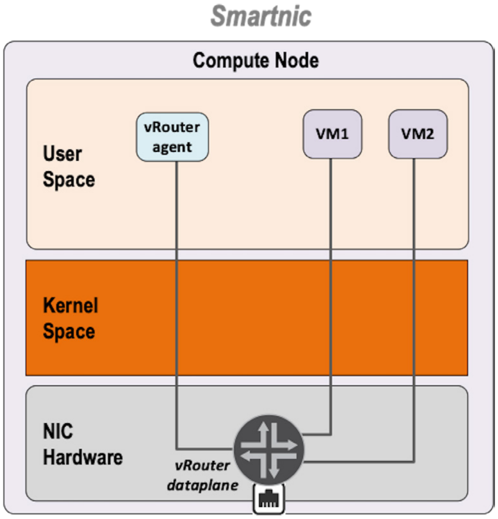


Figure 3.9 *vRouter Running Inside Smart NIC*

Currently, Contrail offers solutions with Smart NICs from Netronome and Mellanox.

DPDK vRouter Architecture

DPDK vRouter Software Architecture

As mentioned previously, DPDK vRouter is a user space application. It is comprised of multiple pthreads, which are also called lcores (logical cores) in DPDK terminology. Each pthread has a specific role to perform. The lcores run in a tight loop, also called the poll mode. They can exchange packets among themselves using DPDK queues. Each lcore has a receive queue, which can be used by other lcores to enqueue packets that need to be processed by that lcore. They also poll different vRouter interfaces queues like physical, VM, and tap. See Figure 3.10.

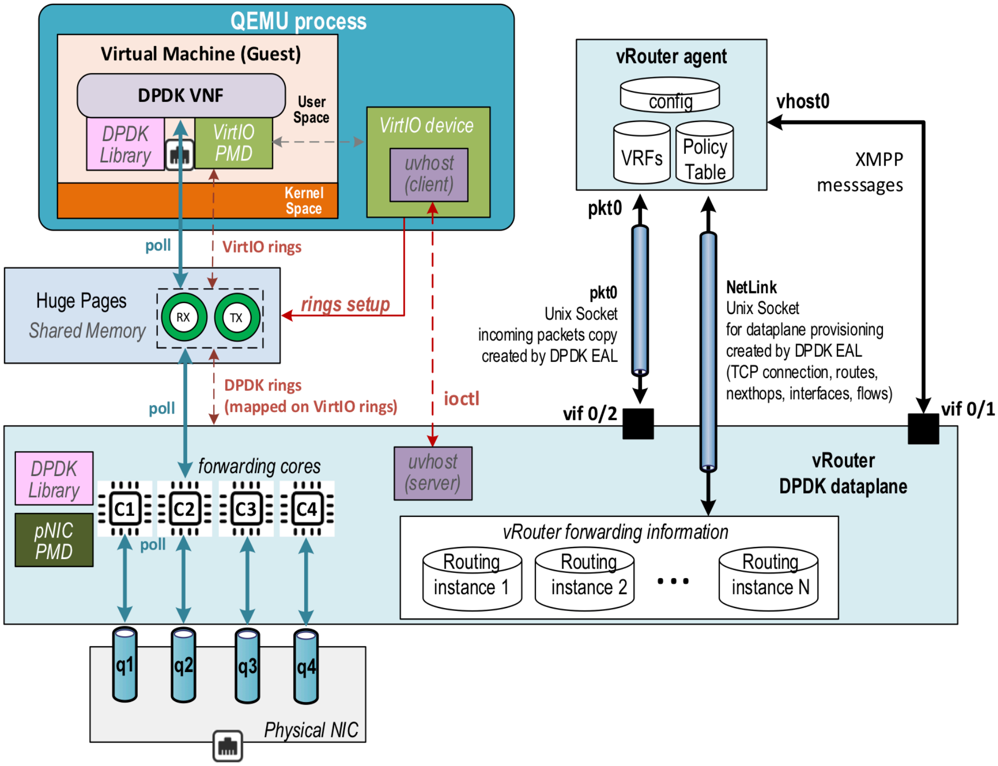


Figure 3.10 DPDK vRouter Software Architecture

DPDK vRouter and lcores

vRouter is a multi-threaded user-space application. It spawns several pthreads or lcores that run in a tight while loop. Each lcore is responsible for a specific task. The different types of lcores are:

- Forwarding lcores
- Service lcores
  - tapdev lcore
  - timer lcore
  - uvhost lcore
  - packet (Pkt0) lcore
  - netlink lcore

Forwaring lcores

Forwarding lcores are responsible for polling the physical and virtual interfaces. Physical interfaces can be a bonded interface too. In addition, they can do the vRouter packet processing, which is briefly illustrated in this chapter's section, "vRouter packet processing Pipeline." These lcores can assume the role of both polling and processing.

These lcores are spawned by the vRouter with a well-defined CPU list. It gets the CPU list as mask (core mask) using the `taskset` Linux command:

```
taskset 0x1e0 /usr/bin/contrail-vrouter-dpdk --no-daemon
```

The hex representation of 0x1e0 is as follows:

CPU number	0	0	0	5	4	3	2	1	0
Bit value	0	0	0	0	1	1	1	1	0

This will make the vRouter spawn four forwarding cores and they will be pinned to CPU numbers 1,2,3,4.

The first forwarding lcore is named as lcore10, the next one is named as lcore11, and so on. Hence if a DPDK vRouter has been configured with four forwarding lcores onto its CPU list, four pthreads will be launched: lcore10, lcore11, lcore12, and lcore13.

Here is the output which lists the threads running in vRouter, its names and also its PIDs:

```
[root@a7s4 ~]# ps -T -p $(pidof contrail-vrouter-dpdk)
  PID  SPID TTY          TIME CMD
 3685  3685 ?        03:47:37 contrail-vroute  >>> Main thread and tuntap lcore
 3685  3800 ?        00:04:32 eal-intr-thread  >>> DPDK library control thread
 3685  3801 ?        00:00:00 rte_mp_handle    >>> DPDK library control thread
 3685  3802 ?        04:55:48 lcore-slave-1    >>> Timer lcore
 3685  3803 ?        00:00:02 lcore-slave-2    >>> uvhost lcore
 3685  3804 ?        00:00:11 lcore-slave-8    >>> Packet (pkt0) lcore
 3685  3805 ?        00:04:12 lcore-slave-9    >>> netlink lcore
 3685  3806 ?        6-16:39:37 lcore-slave-10 >>> forwarding thread #1
 3685  3807 ?        6-16:40:48 lcore-slave-11 >>> forwarding thread #2
```

```
3685  3808 ?        6-16:35:35 lcore-slave-12 >>> forwarding thread #3
3685  3809 ?        6-16:37:52 lcore-slave-13 >>> forwarding thread #4
3685  5048 ?        00:00:00 lcore-slave-9    >>> fork of netlink core
                                                         (for client mode qemu)
```

Packet Processing Models in Data Plane Software

Multi-threaded data plane applications follow three types of packet processing models:

- Run-to-completion model
- Pipeline model
- Hybrid model

In the *run-to-completion model*, the software does not have multiple stages and it does the entire processing in a single context or *single stage*. There is no inter-threads packet buffering, hence latency overheads are less.

In the *pipeline model*, the software is divided into *multiple stages*. Each stage completes part of the processing and hands it over to the next stage, and so on. It's handed over by using a FIFO buffer between the stages. These buffers introduce latency but the main advantage of this model is that it ensures even load balancing of all stages in the event when only a few stages are loaded.

Contrail vRouter uses a *hybrid model* where it employs a pipelining model in some scenarios and a run-to-completion model in others. This ensures good load balancing of all lcores with a reasonable latency. It needs to have FIFO buffers due to pipelining.

The vRouter performs the following types of packet processing:

- Run-to-completion: A forwarding lcore polls for packets from a vif Rx queue. Then it performs the vRouter packet processing and determines the encap/decap that needs to be done. It also finds which outgoing vifs the modified packets needs to be sent. Finally, it sends them on those outgoing vif Tx queues.
- Pipeline: A first forwarding lcore polls for packets from a vif Rx queue. It then distributes these packets to another forwarding lcore using the DPDK software rings (FIFO buffer) between them. The other forwarding lcores pick up the packets and perform the packet processing. Then they send the modified packets to other vif Tx queues.

vRouter uses the Run-to-completion model in following scenarios:

- If the option --vr_no_load_balance is configured, all packets in any direction are processed using this model.
- Without that option, only the packets coming from the physical NIC encapsulated in MPLSoUDP or VXLAN are processed using this model.

vRouter uses Pipeline model (only when option --vr_no_load_balance is absent), in the following scenarios for:

- packets coming from the physical NIC encapsulated in MPLSoGRE
- packets received by the vRouter from the workloads (Virtual Machines or containers) with or without multiqueue virtio.

Service lcores

Service lcores are responsible for tasks other than packet forwarding, They handle all vRouter interfaces (vRouter ports) tasks like port setup, workload interface to vRouter port binding, routing information propagation, etc.

It also handles other book-keeping and miscellaneous tasks for vRouter like timer management and Virtio (vhost-user) control paths. By default, these lcores are not pinned to any physical CPU.

Most of the service lcores make use of user sockets to talk to other processes using Inter Process Communication (IPC). Some of the entities they communicate with are vRouter agent, Qemu (Virtual Machine), and Linux stack.

User Sockets in vRouter

DPDK vRouter data plane uses sockets and eventfds to exchange IO messages. uSocket (User Socket) is the primary IO socket transport mechanism in vRouter.

A specific data message protocol is used over sockets to carry the IO messages between two processes (vRouter data plane and qemu or vRouter agent).

DPDK vRouter uses three kinds of protocols:

- Netlink: the netllink protocol socket carries IO messages that have a Netlink header.
- Packet: the packet protocol socket carries packets that have an agent_header. A packet message contains a ring, a vif, and a child usocket. This child usocket represents an eventfd that is used by the datapath threads to wake up the packet thread whenever there are new packets that are enqueued on the ring.
- Event: the event protocol represents an eventfd. You can write an 8-byte value that accumulates over writes to be read by the reader. This is used as a wakeup mechanism for one or more threads.

Message protocol, multiple transport protocol types can be used for each of these. In Contrail DPDK vRouter data plane, the Netlink protocol is used over UNIX Socket transport protocol and the packet protocol is carried over the Raw Socket transport.

Tapdev lcore

The DPDK vRouter needs to be able to exchange packets with the Linux kernel networking stack. The vhost0 interface is a network interface which is shared by both the vRouter application and other Linux applications. For instance, on a single network interface compute, the physical IP of the compute is migrated onto the vhost0 interface. This IP is used by the SSH server daemon and can't be migrated into the DPDK application (vRouter data plane). See Figure 3.11.

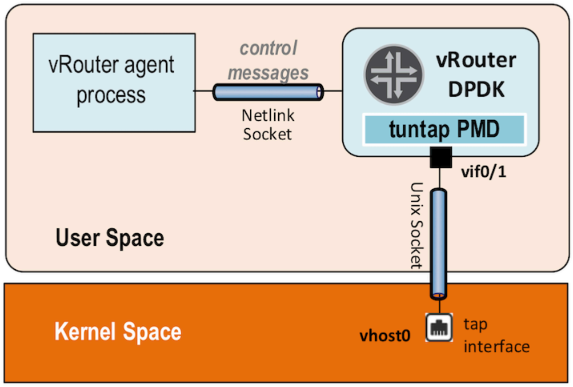


Figure 3.11 Tapdev lcore

The DPDK control plane provides two solutions that allow user space applications to exchange packets with the kernel networking stack:

- Kernel NIC Interface (KNI)
- tuntap Interface

The vRouter implements a custom PMD for tuntap devices that can be used to send and receive packets between the vRouter and the Linux host OS kernel.

Currently “vhost0” and “monitoring” interfaces (used by vifdump utility, which is explained later) make use of it.

When a tap device is initialized, vRouter uses the “tun” driver (/dev/net/tun) in Linux and creates a tuntap device:

```
[root@a7s3 ~]# ethtool -i vhost0
driver: tun
version: 1.6
firmware-version:
expansion-rom-version:
bus-info: tap
supports-statistics: no
supports-test: no
```

```
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no
```

Once the Netlink communication channel between the vRouter agent and vRouter DPDK data plane has been set up using the Netlink lcore, the agent sends a message to the vRouter DPDK to add the vhost0 interface, as shown in Figure 3.12.

As part of this sequence, a new vhost0 vif or vif0/1 is created and is set up so that the tapdev lcore is responsible for polling the vHost0 interface. The vhost0 is the Linux network interface used by the vRouter agent to send XMPP packets to Contrail Control nodes.

In each iteration, the PMD uses raw "read" and "write" socket calls to receive and transmit packets to the tuntap device.

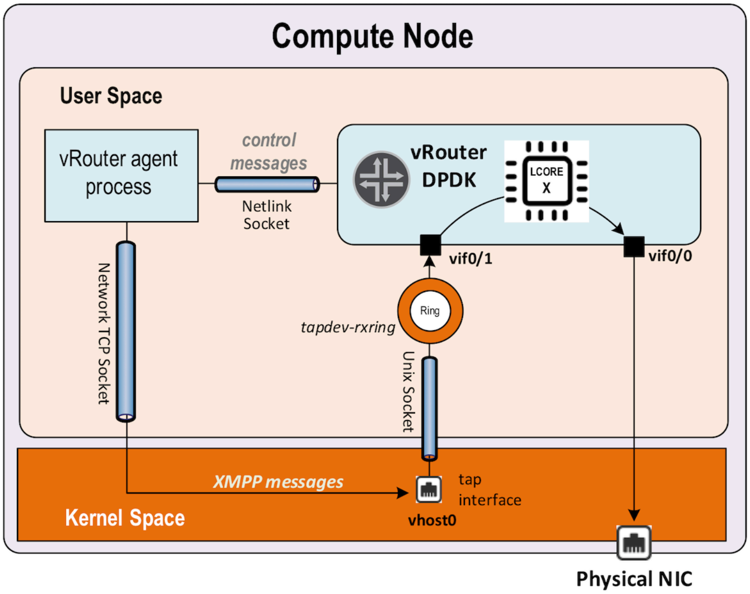


Figure 3.12 Receiving Packets From vhost0

One of the forwarding cores will be assigned to process the vhost0 packets and will be polling a dedicated DPDK ring, called the tapdev_rx_ring. This ring will be added to the forwarding lcore's poll list when the vhost vif is added by the vRouter agent.

The tapdev PMD receives packets from the vhost0 interface using the "read()" socket call and enqueues them to the above mentioned DPDK ring, as shown in Figure 3.13. One forwarding core is designated to pick one of these packets and process it. In most cases, this is lcore 10.

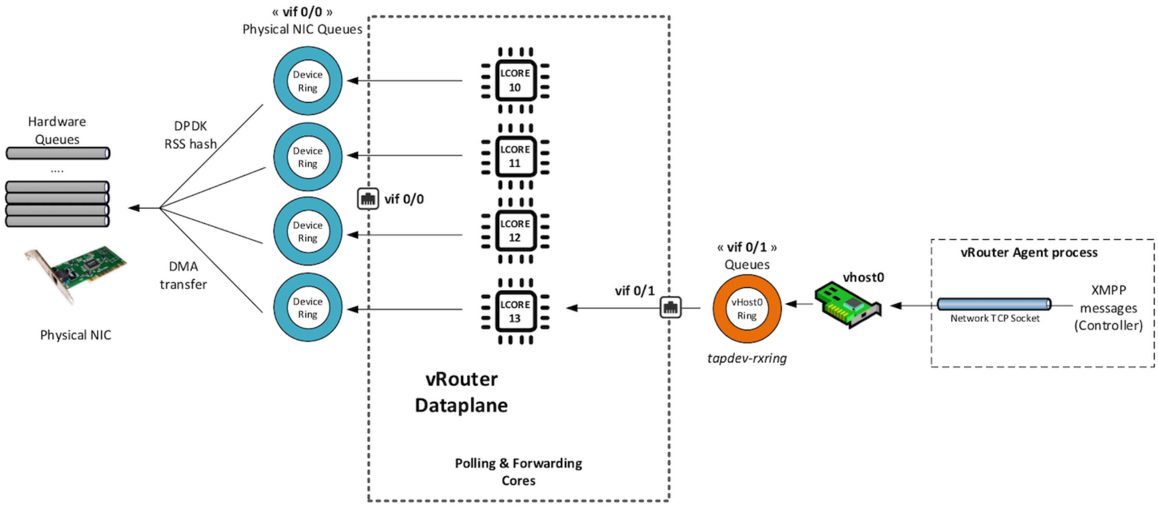


Figure 3.13 Receiving Packets From vhost0 to DPDK vRouter

Sending Packets to vhost0

All the forwarding cores will have Tx rings for vhost0. Packets that need to be sent to vhost0 will be enqueued to these Tx rings by the lcores. The tapdev PMD polls these Tx rings and dequeues the packets from these rings. It then sends the packets to the "vhost0" interface using the write socket call. See Figure 3.14.

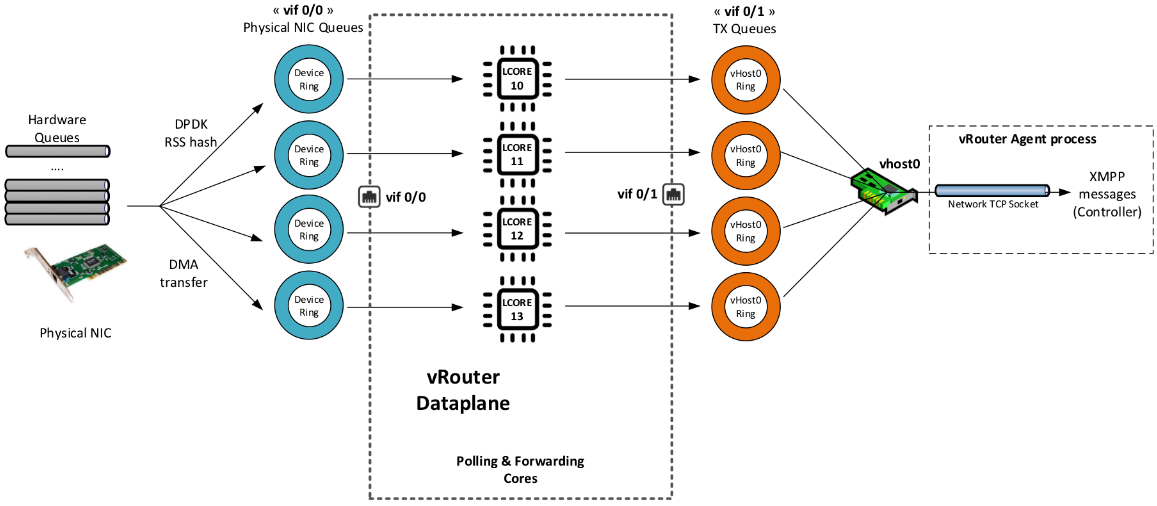


Figure 3.14 Sending Packets From DPDK vRouter to vhost0

Timer lcore

Timer lcore is responsible for managing the timer list and executing timer callbacks of the different timers in vRouter. The timers include internal DPDK library timers for bonding or vRouter timers for fragmentation, etc. It executes an API provided by the DPDK library called "rte_timer_manage()", which manages the timers. The precision of the timer depends on the call frequency of this function. The more often the function is called, the more CPU resources it will use. In the case of vRouter, the precision is 100us.

Uvhost lcore

Uvhost lcore is responsible for handling the messages between the qemu and the vRouter, sometimes called the "vhost-user control channel." It handles both the cases – when qemu is server or qemu is client. The communication is through UNIX sockets. Once the communication channel is established, the userspace vhost protocol takes place. During this protocol message exchange, qemu and the vRouter exchange information about the VM's memory regions, virtio ring addresses, and the supported features. At the end of the message exchange, the virtio ring is enabled and data communication between VM and vRouter can take place.

Client Mode qemu and Server Mode qemu

A communication channel between qemu and vRouter DPDK needs to be established for memory mapping the VM's memory in the vRouter. This enables the virtio rings as a shared memory to exchange packets. When the communication channel needs to be established, there are two choices:

- Client mode qemu
- Server mode qemu

Client Mode qemu

Here, the Contrail-vRouter-DPDK process is in `LISTENING` state, which indicates the vRouter DPDK is the server. This means the qemu process is the client and connects to it:

```
$ netstat -anp|grep tap3dd8d56d-ca
unix  2      [ ACC ]     STREAM     LISTENING     850276771 11736/contrail-vrou /var/run/vrouter/
uvh_vif_tap3dd8d56d-ca
unix  3      [ ]         STREAM     CONNECTED     850270646 11736/contrail-vrou /var/run/vrouter/
uvh_vif_tap3dd8d56d-ca

[2020-10-25 10:49:33]root@bcomp80:~
$ ps -eaf|grep 11736
root     11736 19907 99 Oct20 ?        39-14:31:44 /usr/bin/contrail-vrouter-dpdk --no-
daemon --vdev eth_bond_bond0,mode=4,xmit_policy=l34,socket_id=1,mac=14:02:ec:66:b8:dc,slave=0000:87:
00.0,slave=0000:09:00.1 --vlan_tci 722 --vlan_fwd_intf_name bond0 --socket-mem 1024 1024
```

In the above example, vRouter DPDK creates the UNIX socket named "uvh_vif_tap3dd8d56d-ca". This socket name is passed by the Contrail controller to the vRouter agent. The agent then adds this virtual interface to the vRouter DPDK using the Netlink channel. In parallel, qemu process is spawned by the nova plugin which waits for this socket to be created. Once created by the vRouter process, qemu then initiates a connection to this socket as a client. Hence the name "client mode qemu".

This is a not a default mode but can be enabled in the configuration, although this mode is not preferred due to the "reconnect issue." This means when the vRouter DPDK process is restarted, the VMs also need to be restarted to trigger the vhost-user protocol.

Server Mode qemu

Here, the qemu-kvm process is in `LISTENING` state, which indicates that qemu is the server. This means vRouter DPDK is the client:

```
(contrail-tools)[root@a7s4-kiran /]$ netstat -anp|grep tap35d2a912-fe
unix  2      [ ACC ]     STREAM     LISTENING     4027198  6729/qemu-kvm        /var/run/vrouter/
uvh_vif_tap35d2a912-fe
unix  3      [ ]         STREAM     CONNECTED     4953961  6729/qemu-kvm        /var/run/vrouter/
uvh_vif_tap35d2a912-fe
```

Further, the virsh XML shows that the socket mode is "server":

```
(nova-libvirt)[root@a7s4-kiran /]# virsh dumpxml 10 | grep server -B2 -A5
    <interface type='vhostuser'>
      <mac address='02:35:d2:a9:12:fe'/>
      <source type='unix' path='/var/run/vrouter/uvh_vif_tap35d2a912-fe' mode='server'/>
      <model type='virtio'/>
      <driver queues='8'/>
      <alias name='net0'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x03' function='0x0'/>
    </interface>
```

In the above example, libvirt (instead of vRouter DPDK) creates the UNIX socket named "uvh_vif_tap3dd8d56d-ca" with the help of Contrail nova plugin. The socket name is then passed by the Contrail controller to the vRouter agent. The agent then adds this virtual interface to the vRouter DPDK using the Netlink channel. The vRouter then initiates a connection to this socket as a client, with qemu being the server. Hence the name "server mode qemu".

This is the default and preferred mode since it avoids the above-mentioned "reconnect issue." This means when the vRouter DPDK process is restarted, it is the responsibility of the vRouter to connect to qemu and to trigger the vhost-user protocol to set up the virtio data channel.

vhost User Protocol

The vhost user protocol consists of a control path and a data path. Once the UNIX socket channel is established between the vRouter and qemu, the vhost-user protocol is initiated by qemu.

All control information is exchanged via this UNIX socket. This includes information for exchanging memory mappings for direct memory access, as well as kicking / interrupting the other side if data is put into the virtio queue. The UNIX socket is named “uvh_tap_vif_xxxxxxxx-xx” and the VM interface is named as “tap_vif_xxxxxxxx-xx”.

Packet (pkt0) lcore

This lcore is responsible for sending and receiving packets between the vRouter agent and vRouter DPDK. The initialization sequence for this communication is triggered once the agent to vRouter Netlink channel has been established. The vRouter agent adds the “pkt0” or the “vif0/2” interface to the vRouter DPDK process using the Netlink channel. The vRouter DPDK then creates a UNIX socket in the path “/var/run/vrouter/dpdk_pkt0”. This socket will be used by the agent to send packets to the vRouter DPDK. In addition, it connects to another UNIX socket which the vRouter agent has already created. The path for the agent’s UNIX socket is – “/var/run/vrouter/agent_pkt0”. This socket will be used by the vRouter DPDK application to send packets to the agent process. The socket protocol is of type PACKET, meaning that the channel only carries packets. The socket type is RAW. Once the sockets are created and configured, the vRouter DPDK blocks on the “poll()” system call for activity of any of these sockets. When there is any activity, the “poll()” system call breaks and appropriate socket handling will take place:

```
(vrouter-agent-dpdk)[root@a7s4-kiran /]$ netstat -anp|grep pkt0
unix  3      [ ]         DGRAM 4952638  3728/contrail-vrout  /var/run/vrouter/dpdk_pkt0
unix  3      [ ]         DGRAM 4951879  22346/contrail-vrou /var/run/vrouter/agent_pkt0
```

The above command shows the two UNIX sockets that are used to send and receive packets between the vRouter agent and vRouter DPDK. The first line of the output shows the dpdk_pkt0 socket and is owned by vRouter DPDK process. The second line of the output shows the agent_pkt0 socket and is owned by the vRouter agent process.

Receiving Packets From Agent

The vRouter agent uses this channel to send packets to vRouter DPDK. These packets can be DHCP, ARP, ICMP, etc. The agent stamps a custom header called the “agent_hdr” to these packets. The agent_hdr consists of information needed by vRouter DPDK to identify the VRF, interface, and other information it needs to route that packet. The structure of the agent_hdr is given below:

```
__attribute__packed__open__
struct agent_hdr {
    unsigned short hdr_ifindex;
    unsigned short hdr_vrf;
    unsigned short hdr_cmd;
    unsigned int hdr_cmd_param;
    unsigned int hdr_cmd_param_1;
    unsigned int hdr_cmd_param_2;
    unsigned int hdr_cmd_param_3;
    unsigned int hdr_cmd_param_4;
    uint8_t hdr_cmd_param_5;
    uint8_t hdr_cmd_param_5_pack[3];
} __attribute__packed__close__;
```

When the agent wants to send a packet, it uses the send() system call to send it the UNIX socket dpdk_pkt0. As soon as that happens, since the vRouter DPDK is listening on that socket, the poll() system call breaks and the packet is read using the read() system call. The buffer being read needs to be converted into the "mbuf" structure, which DPDK understands. To accomplish this, a new mempool called the "packet_mbuf_pool" is created during initialization. This mempool has a collection of mbufs. A new mbuf is allocated from this mempool and the buffer is copied into the mbuf. It is then routed like a regular packet received on the pkt0 (vif0/2) interface of vRouter. This processing happens in the context of this packet lcore. See Figure 3.15.

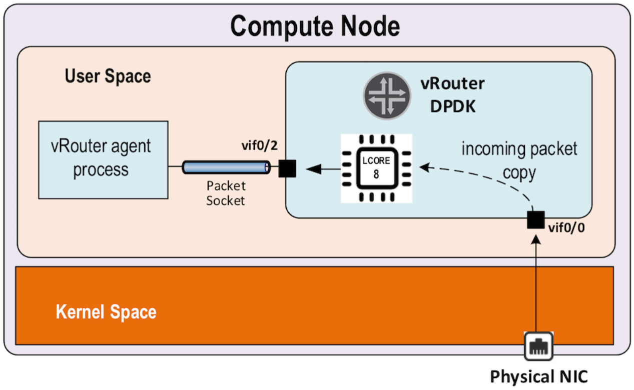


Figure 3.15 *Packet lcore*

Sending Packets to Agent

Packets can be sent to agent from any of the forwarding lcores. During the initialization sequence of the packet lcore, a DPDK ring called the "packet_tx" ring is created. Along with it, an "event_fd" is registered to wake up the packet lcore thread from any forwarding lcore. When the forwarding lcore determines that a packet needs to be punted to the agent, it enqueues the packet into the "packet_tx" ring and wakes up the packet lcore using the event_fd that it registered. The packet lcore then wakes up and drains this ring and uses the send() system call to send the packet to the agent using the socket agent_pkt0.

Netlink lcore

Netlink lcore is responsible for establishing a communication channel with the agent for programming the forwarding state (like routes, next hops, labels, etc.). See Figure 3.16.

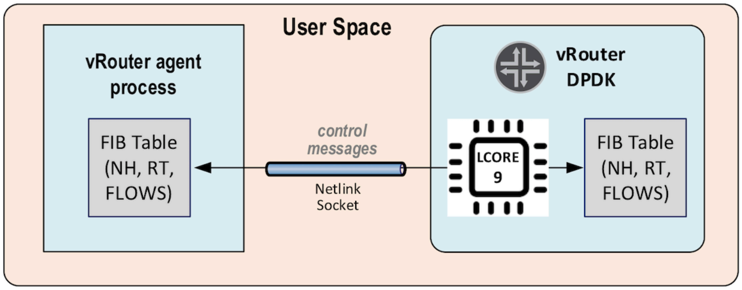


Figure 3.16 Netlink lcore

The Netlink lcore creates a unix server socket at `/var/run/vrouter/dpdk_netlink` to which the agent connects:

```
(vrouter-agent-dpdk)[root@a7s4-kiran /]$ netstat -anp | grep dpdk_netlink
unix  2      [ ACC ]     STREAM     LISTENING     46105    3728/contrail-vrout  /var/run/vrouter/
dpdk_netlink
unix  3      [ ]         STREAM     CONNECTED     4952631  3728/contrail-vrout  /var/run/vrouter/
dpdk_netlink
(vrouter-agent-dpdk)[root@a7s4-kiran /]$ ps -eaf|grep 3728
root      3728  2551 99 Oct02 ?        210-14:44:48 /usr/bin/contrail-vrouter-dpdk --no-
daemon --socket-mem 1024 --vlan_tci 101 --vdev eth_bond_bond0,mode=4,xmit_policy=l34,socket_
id=0,mac=00:1b:21:bb:f9:48,lacp_rate=0,slave=0000:02:00.0,slave=0000:02:00.1
```

The first line of the output shows the state as "LISTENING" for DPDK vRouter, which indicates that it is a server and is waiting for clients such as agent to connect to it.

The second line shows the agent connected to it and so the state is "CONNECTED".

The protocol that is carried in this socket is Netlink, which means all messages have a Netlink header that is 24 bytes in size followed by the payload. The socket type is UNIX. The Netlink header is comprised of the following:

- Netlink message header
- Generic Netlink message header
- Netlink attribute

The header can be viewed easily using gdb to the DPDK vRouter:

```
(gdb) ptype struct nlmsghdr   □ Netlink message header
type = struct nlmsghdr {
    unsigned int nlmsg_len;
    unsigned short nlmsg_type;
    unsigned short nlmsg_flags;
    unsigned int nlmsg_seq;
    unsigned int nlmsg_pid;
}
(gdb) ptype struct genlmsghdr   □ Generic netlink message header
type = struct genlmsghdr {
    __u8 cmd;
    __u8 version;
    __u16 reserved;
}
(gdb) ptype struct nlattr     □ Netlink attribute
type = struct nlattr {
    __u16 nla_len;
    __u16 nla_type;
}
(gdb) p sizeof(struct nlmsghdr) + sizeof(struct genlmsghdr) + sizeof(struct nlattr)
$1 = 24
```

The payload of this message is in "Sandesh" format. This is a proprietary data format (similar to XML) used by the agent and vRouter. The format is:

Object name	Type serial number value	Type serial number length value	…..	Type serial number value

MORE? For more on the Sandesh format: http://juniper.github.io/contrail-vnc/sandesh.html.

The object name specifies the type of object the message contains - like next hop, route, MPLS, etc.

The type can be fixed length datatypes like uint8, uint16, uint32. It can also be a variable length data type like "list," in which case there will be a "length" field to specify the length of the list.

These messages are parsed by inbuilt parser and appropriate callbacks called depending on the objects. Example: For a next hop object, the next hop callback within vRouter is called, which in turn programs that next hop in the next hop table.

If the vRouter needs to return a status or error message to the agent after processing the Sandesh object, it can do so. That way the agent gets to know if the programming was successful or not.

DPDK vRouter Packet Processing

Packet Polling and Processing

During initialization of the physical NIC interface, the vRouter configures it with the same number of queues as the number of forwarding cores it contains. For example, if the vRouter has four forwarding cores, the number of Rx queues it configures to the NIC is four.

A vif queue is made up of two DPDK rings as seen in Figure 3.17:

- one RX ring in which there are stored packets received from a NIC to be processed by the vRouter, and
- one TX ring in which there are stored packets to be sent by the vRouter to a NIC.

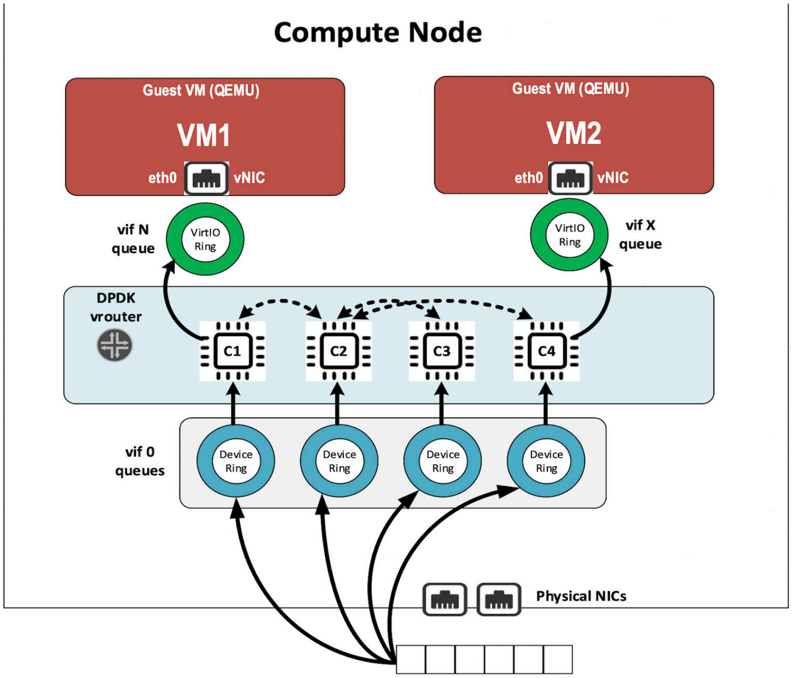


Figure 3.17 Packet Polling and Processing

Packets stored in vif RX rings are polled by a forwarding lcore. There is a one-to-one mapping between forwarding cores and the NIC's Rx queues. The polled packets are then processed by either the same lcore or different one, and pushed to a target vif's TX ring, as shown in Figure 3.18.

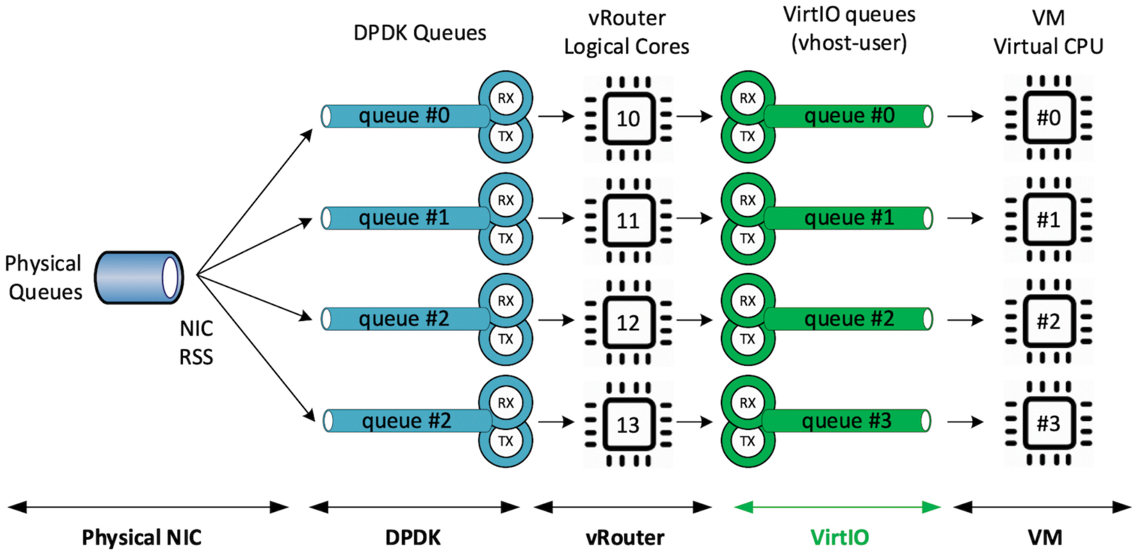


Figure 3.18 Packet Path From NIC to VM

Each of the lcore10 and higher started by a DPDK vRouter is a polling and a processing thread. They run based on the CPU list defined by CPU_LIST variable during provisioning.

MPLS Over GRE Overlay

Incoming overlay-encapsulated packets are received on the compute physical NIC, usually a bond made up of two NICs. See Figure 3.19.

Incoming overlay packets are placed into physical NIC queues using the DPDK RSS (Received Side Scaling) hashing algorithm. The RSS hashing algorithm for MPLSoGRE only uses three tuple values: IP source, IP destination, and protocol number. Unfortunately, the entropy of these three values is low when GRE is used.

For all overlay the flows between a pair of computes, the three tuple hash values remain the same. Hence the hashing algorithm will provide an identical value for all network flows coming from each single compute, which decreases the entrope.

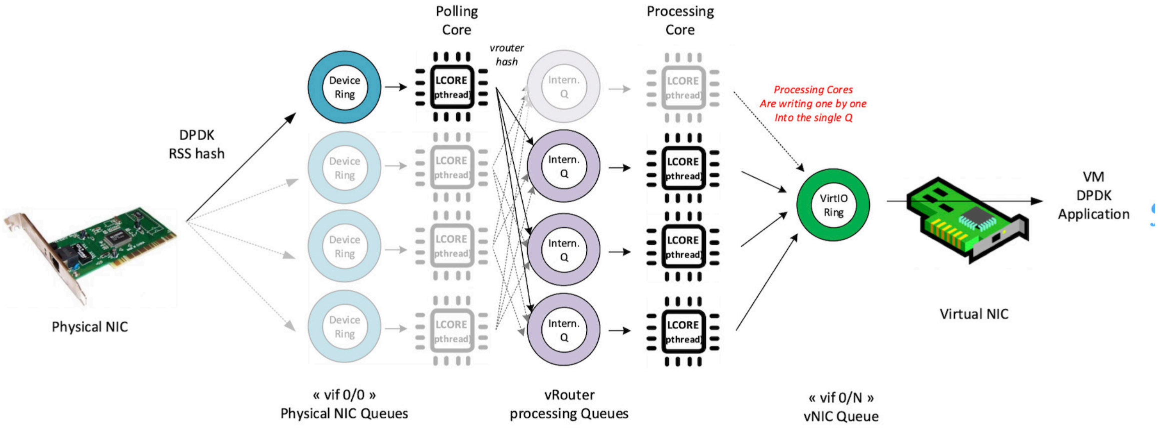


Figure 3.19 Load Balancing in the Case of MPLS Over GRE Overlay

Consequently, all packets coming from VMs located on the same compute will be received only in one DPDK RX ring of the vif0/0 interface (the vRouter interface connected to the underlay network). So incoming MPLS GRE overlay packets will not be well balanced onto the different forwarding cores.

The DPDK pipeline model will used to mitigate this. One lcore will perform only packet polling and another lcore will perform packet processing.

A hash algorithm is applied onto the decapsulated packet headers (inner packet) in order to increase the entropy. As a result of this mechanism, packets are well balanced across all the available forwarding lcores, even if the encapsulation is MPLS over GRE.

UDP Overlay (VxLAN or MPLS over UDP)

When an UDP overlay protocol is used (MPLS over UDP or VxLAN) there is better inherent entropy due to the five tuples of the packet headers: IP source, IP destination, source port, destination port, and protocol.

Different network flows coming from a same remote VM generate different RSS hash results, which results in even load balancing. See Figure 3.20.

Consequently, incoming overlay packets are balanced onto all of the DPDK RX rings configured for the physical interface. It is not necessary to split polling and processing steps. Therefore, when an UDP overlay protocol is used to transport user packets between compute nodes, the vRouter uses the same lcore for both polling and processing steps. This is the same as the run-to-completion model.

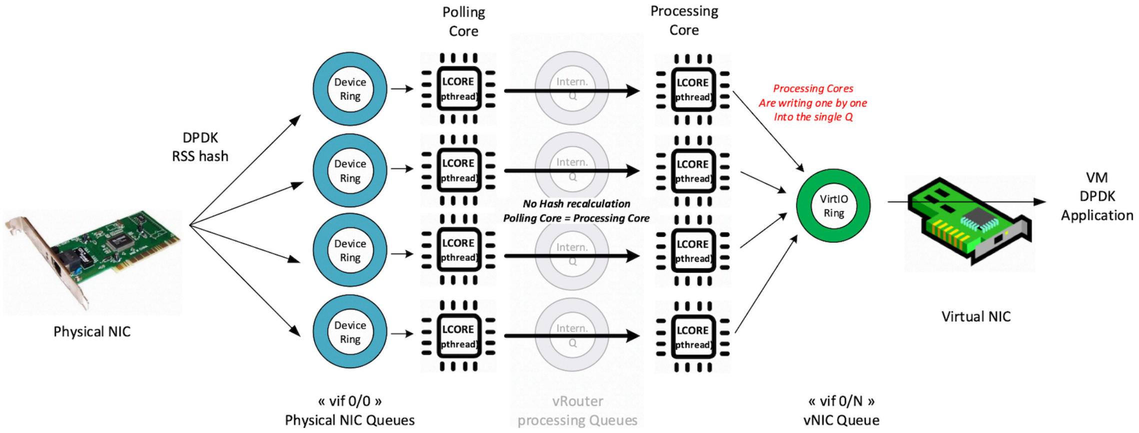

Polling Core
Processing Core
DPDK RSS hash
Device Ring
LCORE pthread
Intern. Q
Processing Cores Are writing one by one Into the single Q
No Hash recalculation Polling Core = Processing Core
VirtIO Ring
VM DPDK Application
Physical NIC
Virtual NIC
« vif 0/0 » Physical NIC Queues
vRouter processing Queues
« vif 0/N » vNIC Queue

Figure 3.20 Load Balancing in the Case of MPLS Over UDP and VXLAN Overlays

It is more efficient to use UDP overlay protocols. Performance ratings with the same DPDK vRouter configuration are higher when an UDP overlay protocol is chosen instead of MPLS over GRE.

Single Queue versus Multi-Queue NIC

A virtual NIC of a VM that is connected to a vRouter can be configured to have one or more queues. Each of the vNIC queues is automatically pinned to a single vRouter forwarding thread (lcore10 and higher). Consequently, when a vNIC is configured with only a single queue, all incoming and outgoing packets are processed by a single vRouter forwarding thread.

In order to avoid binding all single queue interfaces to the same forwarding thread, each interface queue is pinned to a different forwarding lcore in a round robin manner when each interface is created. For example, single queue vif0/3 is automatically pinned to lcore 10, single queue vif0/4 is automatically pinned to lcore 11, and so on.

Here, the vRouter's total CPU power is automatically distributed among all the single queue interfaces. This distribution is automatically defined for each interface and is kept unchanged during the interface duration.

When a vNIC is configured with several queues, each single queue is bound to a different forwarding thread. Hence the vRouter total CPU power is automatically distributed among the different queues on each multi-queue vNIC.

Even if there is no hard rule that prevents a user from configuring a different number of queues on a vNIC compared to the number of forwarding lcores configured, the best scenario is to configure each multi- queue NIC with the same number of queues as the number of configured forwarding threads. See Figure 3.21.

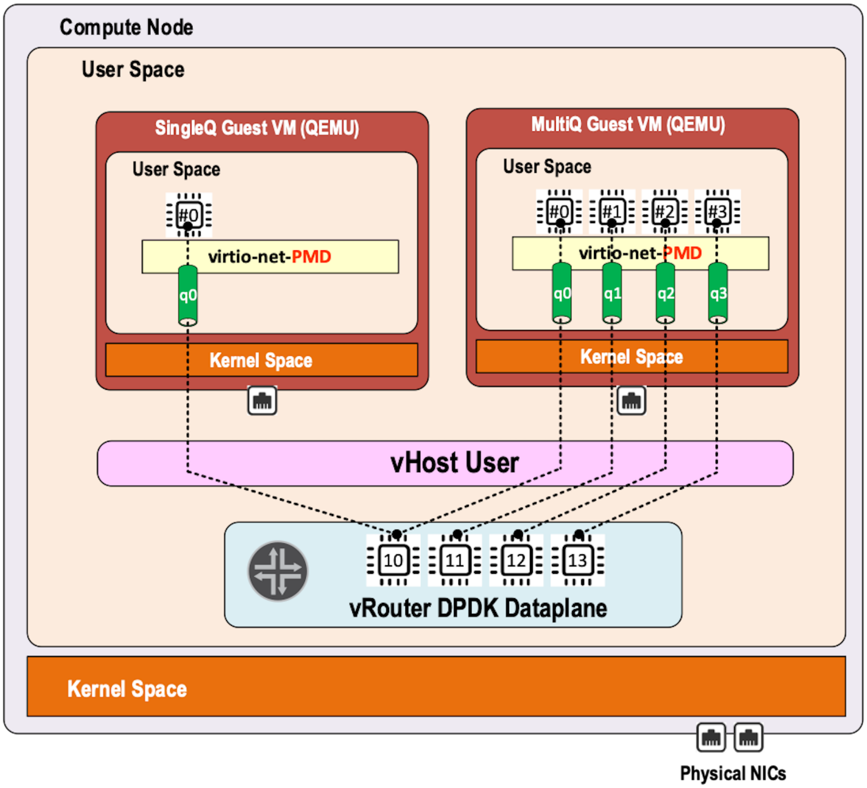


Figure 3.21 Single Queue versus Multi-Queue vNIC

NOTE You also have to take into consideration that the current DPDK vRouter is unable to process correctly a multi-queue vNIC, which is configured with more queues than the number of forwarding threads configured.

Supported Scenarios

Contrail DPDK vRouter supports DPDK virtual machines as well as Linux Kernel virtual machines. Likewise, a contrail Kernel vRouter supports both DPDK and non-DPDK virtual machines. See Figure 3.22.

However, only two scenarios really make sense:

- Kernel mode vRouter supporting kernel mode virtual machines
- DPDK vRouter supporting DPDK virtual machines

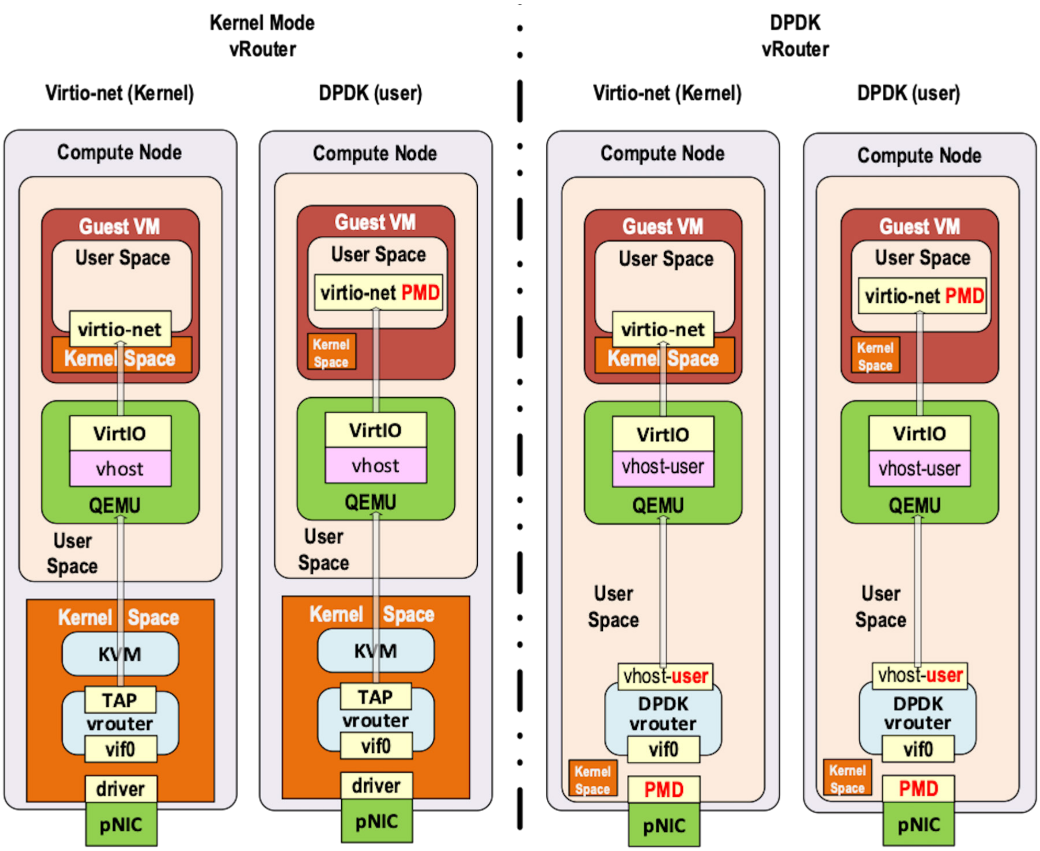


Figure 3.22 Supported Scenarios of Kernel and DPDK vRouters

In the Kernel scenario, both VMs and Contrail vRouter work with a regular Linux TCP/IP stack using interrupt mode packet processing. They both suffer the same limitations (packet processing does not scale due to interrupt mode) and the same advantages (not required to book lots of CPU for packet processing). So this scenario is best used when the VMs do not expect a high network connectivity performance.

In the DPDK scenario, both VMs and Contrail vRouter work with a DPDK library using poll mode packet processing. Both suffer from the same limitations and have the same advantages as poll mode requires some CPUs for packet processing, and it allows you to reach line rate packet processing. This scenario is the best used when the VMs require a high network connectivity performance, which is typically, Virtual Network Functions (VNF).

Hybrid cases are unsuitable, but unavoidable in certain circumstances. Many VMs have both kernel and DPDK interfaces. Generally, kernel interfaces are used for management purposes and DPDK interfaces are used for data.

When a Kernel mode VM is plugged into a DPDK vRouter, it impacts the whole Contrail vRouter and performance suffers. The DPDK vRouter has to emulate interrupt mode using KVM features in order to kick the VM. It involves a "VMExit," which is like a system call to the hypervisor and takes lots of CPU cycles. This not only impacts the kernel mode VM but all the other DPDK VMs as well.

A DPDK VM plugged into a Contrail kernel mode vRouter is also very inefficient. Even if the VM is able to process its network packets at a very high speed, the Linux kernel packet processing used by kernel mode vRouter does not scale well. So, at the end, lots of packets generated by a high speed VNF plugged on a Contrail kernel mode vRouter could be lost.

This is why Contrail users have to be consistent and to plug data interfaces of DPDK VMs into a DPDK vRouter and data interfaces of kernel mode VMs into kernel mode vRouter.

When virtual infrastructure is made up of several kinds of VMs (both DPDK and non DPDK), placement strategy has to be defined in order to spawn DPDK VM into computes fitted with Contrail DPDK vRouter and to spawn non-DPDK VMs into computes fitted with Contrail kernel mode vRouter.

Chapter 4

Contrail DPDK vRouter Setup

The Contrail DPDK vRouter setup mainly consists of:

- Defining NIC cards to be used by the vRouter for its interconnection with the physical network.
- Defining CPU resources to be allocated to the DPDK vRouter.
- Defining the huge pages memory to be used by the DPDK vRouter to create vRouter interface DPDK rings for physical and VM NICs.
- Configuring the number of queues of DPDK vRouter physical and VM NICs. Queues will be configured automatically with a one-to-one mapping. On physical NICs, the vRouter will configure as many queues as the number of allocated polling cores. For each VM NIC, the vRouter will bind each queue to a single polling core. That means the vRouter provides one-to-one polling core/queue mapping until the number of VM queues are not bigger than vRouter allocated cores.

In Centos or RedHat Enterprise Linux, Contrail vRouter DPDK-specific setup is defined in the `/etc/sysconfig/network-scripts/ifcfg-vhost0` configuration file. To activate changes, the vRouter agent vhost0 network interface has to be recreated to get the modified set up enforced:

```
$ sudo ifdown vhost0
$ sudo ifup vhost0
```

DPDK vRouter Physical Network Interface

Only one physical interface can be plugged into the vif0/0 port of the vRouter. Usually, a bond interface is created to group two physical interfaces in a single entity, which is plugged onto the vRouter for resiliency purposes.

Physical NICs used in the bond interface are defined in BIND_INT parameter:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
DEVICE=vhost0
DEVICETYPE=vhost
TYPE=dpdk
BIND_INT=0000:02:01.0,0000:02:02.0
```

As well as other parameters like bond mode, policy, and driver:

```
BOND_MODE=4
BOND_POLICY=layer3+4
DRIVER=uio_pci_generic
```

Using the following command, you can display PCI identifier of physical interfaces, which are available in the Linux OS:

```
$ sudo lshw -class network | grep pci@
       bus info: pci@0000:02:01.0
       bus info: pci@0000:02:02.0
       bus info: pci@0000:03:00.0
```

Once the Contrail DPDK vRouter has been started, you can see the actual physical interfaces used for the underlay network interconnection:

```
$ sudo docker exec contrail-vrouter-agent-dpdk /opt/contrail/bin/dpdk_nic_bind.py -s
Network devices using DPDK-compatible driver
============================================
0000:02:01.0 '82540EM Gigabit Ethernet Controller' drv=uio_pci_generic unused=e1000
0000:02:02.0 '82540EM Gigabit Ethernet Controller' drv=uio_pci_generic unused=e1000
Network devices using kernel driver
===================================
0000:03:00.0 'Virtio network device' if= drv=virtio-pci unused=virtio_pci,uio_pci_generic
Other network devices
=====================
<none>
```

DPDK vRouter CPU Setup

DPDK requires the NIC used by the vRouter application to be managed with a PMD to directly read and write packets from the user space application without the use of IRQ to get notified packets processed. Some CPUs will perform an infinite loop to continuously check that there are no packets to be processed in the queues of the NICs used by the application. This is why some CPUs have to be booked for an exclusive use by the DPDK application for packet polling purposes. CPU allocation planning is a very important task that must be completed by architecture team in charge of the virtual infrastructure.

On each compute node you have to allocate:

- Some CPUs to be kept available for the Linux OS.
- Some CPUs for the VMs. Generally, this is the main purpose of the your virtual infrastructure creation.
- Some CPUs for the vRouter high-speed packet processing (polling, processing, and forwarding steps).

This section only considers servers with a NUMA architecture and hyper-threading enabled. The term CPU will be used to mean both logical cores (main core and its sibling) created on each physical core. The term core will be used to mean a single logical core. In this section, each CPU is made up of two cores (physical and its sibling). It also assumes a containerized version of OpenStack and Contrail is being used. In Figure 4.1, the virtual infrastructure architect is starting to define the number of CPUs to be allocated in each group as described.

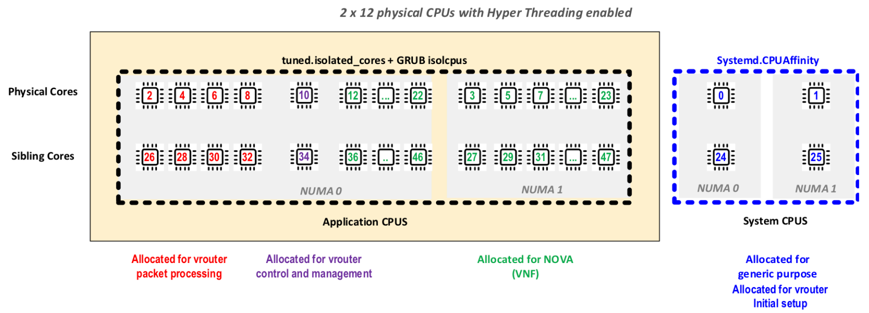

2 x 12 physical CPUs with Hyper Threading enabled
tuned.isolated_cores + GRUB isolcpus
Systemd.CPUAffinity
Physical Cores
Sibling Cores
2
4
6
8
10
12
...
22
3
5
7
...
23
0
1
26
28
30
32
34
36
..
46
27
29
31
...
47
24
25
NUMA 0
NUMA 1
NUMA 0
NUMA 1
Application CPUS
System CPUS
Allocated for vrouter packet processing
Allocated for vrouter control and management
Allocated for NOVA (VNF)
Allocated for generic purpose
Allocated for vrouter Initial setup

Figure 4.1 Compute Node CPU Allocation

In order to get the best performance, CPUs allocated to VMs and to vRouter have to be isolated from those that are kept to the Linux OS. CPU isolation is the first setup to be completed to define the CPUs that will no longer be used by the Linux OS. Those CPUs will be dedicated to the DPDK vRouter or used by OpenStack Nova to spawn VMs.

Figure 4.2 shows the CPU core topology of a two sockets system with 2*12 physical cores (hyper-threading enabled). This topology will be used in the configuration examples provided in the following sections.

```
NUMA node0 CPU(s):
        PHY cores: 0  2  4  6  8  10 12 14 16 18 20 22
        HT cores : 24 26 28 30 32 34 36 38 40 42 44 46
NUMA node1 CPU(s):
        PHY cores: 1  3  5  7  9  11 13 15 17 19 21 23
        HT cores : 25 27 29 31 33 35 37 39 41 43 45 47
```

Figure 4.2 CPU Core Topology of a Two Sockets System

CPU Kept for Linux OS

By default, all CPUs are included in the group of CPUs available for OS needs. These CPUs are *isolated* because they are no longer used to process all tasks. Several mechanisms are used in order to isolate a CPU:

- remove the CPU from the “common” CPU list used to process all tasks
- change the scheduling algorithm (cooperative to pre-emptive)
- participate in “not to interrupt” processing

It is possible to remove some CPUs using the isolcpus kernel parameter. This kernel parameter has to be provisioned at the system startup. The GRUB configuration is updated to define isolcpus parameter and then the system restarted.

This next example keeps only CPU 0,1,24, and 25 for the Linux OS, excluding them from the isolcpus list. It’s strongly recommended to use at least the first CPU (main core and its sibling) on each NUMA:

```
$ vi /etc/default/grub
GRUB_CMDLINE_LINUX="console=tty0 console=ttyS0,115200n8 crashkernel=auto rhgb quiet default_
hugepagesz=1GB hugepagesz=1G hugepages=28 iommu=pt intel_iommu=on isolcpus=2-23,26-47"
$ grub2-mkconfig -o /etc/grub2.cfg
```

You also need to specify the CPUs that have to be used by the Linux OS in the Systemd configuration file (this step is useless when RedHat OS is used with tuned as described next):

```
$ vi /systemd/system.conf
CPUAffinity=0-1,24-25
$ sudo systemctl deamon-reexec
$ sudo systemctl system.slice restart
```

When a RedHat Linux OS is used, it’s recommended to configure tuned to get a stronger CPU isolation:

```
$ vi /etc/tuned/cpu-partitioning-variables.conf
isolated_cores=2-23,26-47
$ sudo systemctl restart tuned
```

When tuned is used, the CPU Affinity value will automatically be overwritten with the CPUs that are not listed in isolated_cores. This is important in order to keep enough CPUs for the Linux OS. Not-isolated CPUs are used by all tasks started and managed by the Linux OS scheduler:

- System configuration and control tasks
- Contrail vRouter agent (SDN control plane)
- Hypervisor configuration and control tasks (VM configuration, for instance)

CPU Allocated to the DPDK vRouter

Packet Polling and Processing Threads

DPDK vRouter speed is dependent on the number of CPUs allocated for packet polling and processing. Each user will encounter a trade-off on how many CPUs will be used for their own applications running on VMs and how many CPUs will be booked for the vRouter to increase network packets' processing speed.

First you must define how many CPUs will be booked for the DPDK vRouter polling and packet processing threads. You can consider that each allocated CPU (two cores – physical and its HT sibling) will bring up to 3MPPS packets network processing speed to the vRouter. This 3MPPS value is dependent on lots of factors: CPU speed, number of CPUs, NUMA usage, packet size, and vRouter mode (packet or flow mode). But it can range between 0.8MPPS to 1.5MPPS per core (1.6MPPS to 3MPPS per CPU).

In the best case scenario, a kernel-mode vRouter generally provides 1MPPS packet speed. There is no easy way to increase kernel mode vRouter performance because it relies on the Linux packet interrupt mode processing model and does not benefit from lots of DPDK optimization (zero packet copy, huge page usage, no context switch between kernel and user space, and no interruption of packet processing threads). Also, it's not easy to build a one-to-one relationship between system CPU resources and Linux processes involved in packet processing.

The DPDK vRouter usually allocates from four to eight packet processing CPUs (physical cores with their siblings).

A higher CPU number (more than eight) is not bringing that much more performance due to some side effects of inter-core communications or the multi-queue setup it would require on VMs (see the multi-queue section later in this chapter).

The amount of CPU allocated to packet polling and processing is defined in the CPU_LIST parameter. This CPU parameter can use two different syntaxes: mask or list.

Here, four physical CPUs (eight logical including second thread/siblings) are allocated to the vRouter for packet polling and processing:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
CPU_LIST=2,4,6,8,26,28,30,32
```

NOTE The mask for CPUs 2,4,6,8,26,28,30,32 maps to the binary value: b0000 0000 0000 0001 0101 0100 0000 0000 0000 0001 0101 0100 (0x154000154h).

vRouter DPDK Data Plane Configuration and Control Threads

Two DPDK vRouter parameters allow you to define CPUs to be allocated for control and configuration threads:

- DPDK_CTRL_THREAD_MASK: defines which CPUs will be allocated for DPDK initialization setup.
- SERVICE_CORE_MASK: defines which CPUs will be allocated for vRouter data plane setup (vRouter interface setup).

The DPDK initialization setup is done only at the vRouter startup while the vRouter data plane setup tasks are done at the vRouter initialization and each time a new interface is plugged or remove into the vRouter. The same CPUs can be shared for these two tasks.

Here we are allocating CPU 10 and 34 data plane control and configuration threads:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
DPDK_CTRL_THREAD_MASK=0x400000400
SERVICE_CORE_MASK=0x400000400
```

NOTE The mask for CPUs 10,34 allocated maps to binary value: b0000 0000 0000 0100 0000 0000 0000 0000 0000 0100 0000 0000 (0x400000400h).

NOTE These parameters can use two different syntaxes: mask or list, the same as for CPU_LIST.

CPU Allocated to Virtual Machines

Host compute CPUs used for user VMS are defined into the Nova configuration file. Here we are enforcing Nova CPU assignment in Nova compute container:

```
$ openstack-config --set /etc/nova/nova.conf DEFAULT vcpu_pin_set 3,5,7,9,11-23,27,29,31,33,35-47

$ cat /etc/nova/nova.conf | grep vcpu_pin_set
vcpu_pin_set=3,5,7,9,11-23,27,29,31,33,35-47
```

In order to get these changes taken into consideration, the Nova compute service must be restarted:

```
$ sudo docker restart nova_compute
```

vRouter Memory Setup

The DPDK library needs to get huge pages allocated by the Linux OS for vNIC rings setup. This is why Contrail DPDK vRouter data plane requires:

- Huge page memory set up on the compute node

- Some of these huge pages to be allocated for the vRouter Physical NIC
- Some of these huge pages to be allocated for the VM NICs
- The huge pages allocated to be visible from both DPDK vRouter application and VMs

Huge Page Memory Configuration On the Compute Node Operating System

Only small huge pages (2MB) can be configured dynamically using sysctl. Bigger huge pages (1GB) must be configured at the system startup.

The following parameters are used:

- default_hugepagesz defines which huge page size is the default (this size will appear in /proc/meminfo and this size will be mounted by default when a pagesize mounting option is not used).
- hugepagesz followed by hugepages defines size and amount, respectively, and the pair can be repeated to configure different sizes of huge pages.

For instance, in order to configure forty 1GB huge pages and forty 2M huge pages at Linux system startup, proceed as described here and then restart the system:

```
$ vi /etc/default/grub
default_hugepagesz=1GB hugepagesz=1G hugepages=40 hugepagesz=2M hugepages=40
$ grub2-mkconfig -o /etc/grub2.cfg
```

Huge Page Allocation for the DPDK vRouter

Some of the available operating system’s huge pages have to be allocated to the vRouter DPDK application to be used to create DPDK rings for the physical NIC. In order for the vRouter DPDK application to be able to use Linux available huge pages, a hugetlbfs pseudo filesystem needs to be mounted. The following line needs to be added to /etc/fstab:

```
$ vi /etc/fstab
hugetlbfs on /dev/hugepages type hugetlbfs (rw,relatime,seclabel,pagesize=1G)
```

The DPDK vRouter detects the huge page hugetlbfs mount point. Here, the DPDK vRouter will try to use 1GB huge pages. If no page size is specified, the DPDK vRouter assumes 2MB huge pages have to be used. If no huge pages of the specified (or 2MB if size is not specified) are available, the Contrail DPDK vRouter will fail to start.

The amount of huge page memory requested by the vRouter at start-up for its physical NIC DPDK rings setup is specified in the socket-mem parameter. In order for the vRouter to request huge pages memory only on the first NUMA socket, we are using this option with only one parameter:

```
--socket-mem <value>
```

In order for the vRouter to request huge pages memory on both the NUMA0 and NUMA1 sockets, now we are using this option with only two parameters:

```
--socket-mem <value>,<value>
```

It is important to allocate huge page memory to all NUMA nodes that will have DPDK interfaces associated with them. If memory is not allocated on a NUMA node associated with a physical NIC or VM, they cannot be used. If you are using two or more ports from different NICs, it is best to ensure that these NICs are on the same CPU socket.

Here we are configuring the vRouter to request 1GB huge pages memory on both NUMA nodes:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
DPDK_COMMAND_ADDITIONAL_ARGS="--socket-mem 1024,1024"
$ sudo ifdown vhost0
$ sudo ifup vhost0
```

DPDK Physical Interface Rings Setup

In the previous section we describe how hugepages memory is allocated to Contrail DPDK vRouter. This memory is mainly used by the DPDK vRouter to create DPDK rings for the physical interface.

Contrail DPDK vRouter will create two DPDK rings for each polling core (which is defined in the CPU_LIST parameter). DPDK rings are circular arrays of RX and TX descriptors that point to mbufs in which the packet content is stored. All mbufs for each TX/RX pair are stored in a single mempool memory area, representing an interface queue. See Figure 4.3.

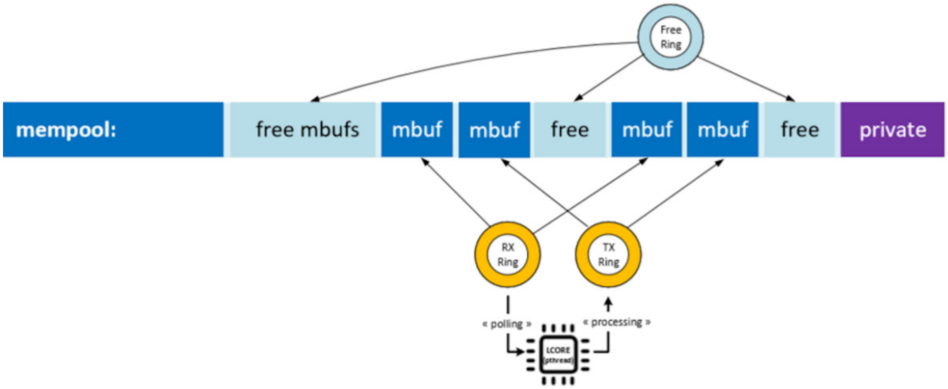


Figure 4.3 Mempool and Mbufs

The following parameters are used for DPDK vRouter physical NIC configuration:

- `--vr_mempool_sz` : this is used to define mempool memory size. Default value is 16384.
- `--dpdk_txd_sz` : this is used to define Physical NIC TX Ring descriptor size. Default value is 256.
- `--dpdk_rxd_sz` : this is used to define Physical NIC RX Ring descriptor size. Default value is 256.

The following formula has to be used to define the mempool size:

```
--vr_mempool_sz = 2 * (dpdk_txd_sz + dpdk_rxd_sz) * number_of_vrouter_cores *
number_of_ports_in_dpdk_bond
```

Next we are configuring the vRouter physical NIC DPDK rings with 512 RX and TX, eight cores, and two ports in a bond. Based on the formula for descriptors, the mempool size should be 32MB:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
DPDK_COMMAND_ADDITIONAL_ARGS="--dpdk_rxd_sz 512 --dpdk_txd_sz 512 --vr_mempool_sz 32768"
$ sudo ifdown vhost0
$ sudo ifup vhost0
```

NOTE Physical NIC DPDK ring size modification can lead to some unexpected side effects (packet loss). The mempool size needed depends on the configured maximum packet size (physical NIC MTU), the number of NICs using the physical bond, and the configured number of RX and TX descriptors.

DPDK vRouter Internal Queues Rings Setup

In some scenarios, Contrail DPDK vRouter is using a DPDK pipeline model in order to split packet polling and processing task in two different threads, as shown in Figure 4.4. When this DPDK pipeline mode is used, some internal queues are created in order to store packets that have been polled by the polling lcore thread before they are processed by the processing lcore thread.

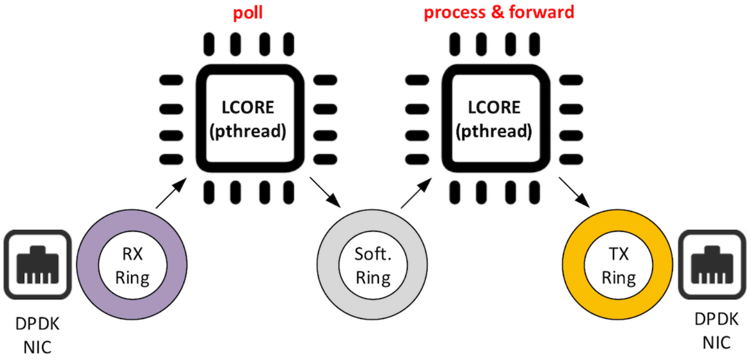


Figure 4.4 DPDK Pipeline Model

Two parameters are used for the DPDK vRouter internal queue (software rings) configuration:

- `--vr_dpdk_tx_ring_sz`: This is used to define forwarding lcores TX Ring descriptor size (1024 by default)
- `--vr_dpdk_rx_ring_sz`: This is used to define forwarding lcores RX Ring descriptor size (1024 by default).

Here we are configuring the vRouter internal rings with 2048 RX and TX descriptors:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
DPDK_COMMAND_ADDITIONAL_ARGS="--vr_dpdk_rx_ring_sz 2048 --vr_dpdk_tx_ring_sz 2048"
$ sudo ifdown vhost0
$ sudo ifup vhost0
```

CAUTION If descriptors: `dpdk_txd_sz` and `dpdk_rxd_sz` are set, the formulas to be used for the `vr_mempool_sz` becomes:

```
--vr_mempool_sz = 2 * [vr_dpdk_rxd_sz + vr_dpdk_txd_sz + (dpdk_txd_sz + dpdk_rxd_sz) * number_of_
ports_in_dpdk_bond] * number_of_vrouter_lcores
```

DPDK Virtual Machine Interface Rings Setup

VM NIC queues are not configured by Contrail vRouter, they are managed by OpenStack. By default, Nova is configuring 256 rx and tx descriptor size Virtio interfaces on the VM.

VM NIC queue size is defined at OpenStack level in the /etc/nova/nova.conf configuration file. They are configured using `rx_queue_size` and `tx_queue_size` parameters:

```
$ cat /etc/nova/nova.conf | grep x_queue_size
rx_queue_size=512
tx_queue_size=512
```

In order to get these changes taken into consideration, Nova compute service has to be restarted:

```
$ sudo docker restart nova_compute
```

The VM NIC and vRouter vif to which each interface is connected are sharing the same queues (DPDK rings):

- a vRouter vif tx ring is the same as the virtual NIC rx ring it is connected to.
- a vRouter vif rx ring is the same as the virtual NIC tx ring it is connected to.

It avoids duplicating the same information and processing overhead (that would be generated to manage data copy between vRouter vif and the Virtual Machine queues).

This is why VM NIC queues have to be accessible from both vRouter and the VM it belongs to. VMs have to be created by the QEMU/KVM hypervisor with a specific property that allows them access to the host OS huge pages and to request huge page allocations.

Huge pages size to be allocated by the hypervisor to the VM has to be specified with *hw:mem_page_size*. The configured huge pages memory size must be the same as those used by the DPDK vRouter (defined into huge page size `hugetlbfs` mount point).

Here we are configuring an OpenStack flavor named m1.large, which defines 1GB size hugepages in `hw:mem_page_size` property:

```
$ openstack flavor set m1.large --property hw:mem_page_size=1GB
```

Then this flavor is used at the instance creation:

```
$ openstack server create --flavor FLAVOR_ID --image IMAGE_ID INSTANCE_NAME
```

NOTE The `hw:mem_page_size` property can also be defined at image level:

```
$ openstack image set --property hw:mem_page_size=1GB < IMAGE_ID>
```

Virtual Machine vif Multiqueue Setup

As explained earlier, when supported, it is suitable to enable multi-queues on VMs NIC. The most suitable scenario is to configure the same number of queues on virtual NIC than the number of polling cores defined on Contrail vRouter. So if Contrail DPDK vRouter is configured with four queues, the best scenario is to configure four queues on the VM's network interfaces.

In OpenStack, in order to get VMs configured with multiqueues, you have to enable multiqueue support on virtual instance image. It can be done with the following command:

```
$ openstack image set --property hw_vif_multiqueue_enabled="true" < IMAGE_ID>
```

Then this image is used at the instance creation:

```
$ openstack server create --flavor <FLAVOR_ID> --image <IMAGE_ID> <INSTANCE_NAME>
```

When an instance is started with multiqueue vif property enabled, each interface is automatically configured with several queues. The number of queues to be enabled on each interface is automatically defined by Nova.

If the compute host (hypervisor node running qemu/kvm) is running Linux Kernel 3.X, the number of queues configured on the VM interface is the same as the number of virtual CPUs configured on the VM, but can't exceed eight queues. That means for a VM configured with ten vCPUs, all its virtual network interface cards will be configured with eight queues when multiqueue is enabled.

If the compute host (hypervisor node running qemu/kvm) is running Linux Kernel 4.X, the number of queues configured on the VM interface is the same as the number of virtual CPUs configured on the VM but can't exceed 256 queues. That means for a VM configured with ten vCPUs, all its virtual network interface cards will be configured with ten queues when multiqueue is enabled.

As explained earlier, Contrail vRouter is not able to process packets generated by connected virtual network interface cards configured with more queues than the number of CPU defined into its CPU_LIST (number of polling and processing cores defined on Contrail vRouter).

Consequently, a Contrail vRouter configured with only four polling and processing cores won't be able to collect a VM configured with ten vCPUs if vif multiqueue property enabled is connected.

One of the following changes has to be performed:

- disable multiqueue on the VM
- add more polling and processing cores on the vRouter (increase to eight cores instead of only four)
- decrease the number of queues configured by Nova on the VM

Unfortunately, no mechanism is provided by Nova to specify a specific value for the number of queues to be enabled on VM network interfaces. Only the described algorithm is proposed.

In order to decrease the number of queues on the VM network interfaces, you have to run the ethtool command inside this VM. For instance, here we are configuring four logical queues on eth0 vNIC :

```
$ sudo ethtool –L eth0 combined 4
```

The VM initialization script has to be modified to automatically decrease the default value defined by Nova for the number of queues configured on its network interfaces to a lower number. This is why, the most efficient setup today is to use Linux Kernel 3.X on OpenStack compute node running QEMU/KVM and to configure eight CPUs into the CPU_LIST of the Contrail DPDK vRouter.

vRouter Routing and Switching Object Tables Dimensioning Parameters

Some parameters supported on the Kernel, as well as DPDK vRouter, allow you to define the size of internal objects tables. They are:

- --vr_flow_entries: maximum flow entries (default is 512K)
- --vr_oflow_entries: maximum overflow entries (default is 8K)
- --vr_bridge_entries: maximum bridge entries (default is 256K)

- --vr_bridge_oentries: maximum bridge overflow entries (default is 0)
- --vr_mpls_labels: maximum MPLS labels used in the node (default is 5K)
- --vr_nexthops: maximum next hops in the node (default is 512K)
- --vr_vrfs: maximum VRFs supported in the node (default is 4096)
- --vr_interfaces: maximum interfaces that can be created (default is 4352)

In order to override their default values, you can configure an updated value using DPDK_COMMAND_ADDITIONAL_ARGS parameter defined in vhost0 DPDK vRouter configuration file. For instance, we can decrease the next hops table size to 32K instead of 512K configured by default:

```
$ vi /etc/sysconfig/network-scripts/ifcfg-vhost0
DPDK_COMMAND_ADDITIONAL_ARGS=”--vr_nexthops=32768”
$ sudo ifdown vhost0
$ sudo ifup vhost0
```

All these parameters could increase vRouter performance but could also have a negative impact when not properly configured.

vRouters DPDK Fine Tuning Parameters

Here is a list of fine-tuning parameters for the DPDK:

- --dpdk_ctrl_thread_mask: (20.03/1912 L2 and later version) CPUs to be used for vRouter control threads (CPU list or hexadecimal bitmask).
- --service_core_mask: (20.03/1912 L2 and later version) CPUs to be used for vRouter service threads (CPU list or hexadecimal bitmask).
- --yield_option: (20.03/1912 L2 and later version) is used to enable or disable yield on forwarding cores (0 or 1 - enabled by default). The yield action occurs in a computer program during multithreading, forcing a processor (core) to relinquish control of the current running thread (vRouter polling and processing tasks) and sending it to the end of the running queue, with the same scheduling priority. As only one single thread is pinned onto the vRouter allocated CPUs listed in CPU_LIST, yield is useless (if the CPU isolation has properly be enforced). In the case below, yield is disabled onto forwarding cores:
  ```
  --yield_option 0
  ```
- --vr_no_load_balance : (20.08 and later version) is used to disable packets processing pipeline model (internal load-balancing in which the processing and forwarding core is different from the polling one). When this parameter is present the internal load-balancing is disabled. When this parameter is absent, the internal load-balancing is enabled (default setup). In the case below, the internal load balancing is disabled:
  ```
  --vr_no_load_balance
  ```

- --vr_uncond_close_flow_on_tcp_rst : (20.08 and later version) is used to enable/disable unconditional closure of Flow on TCP RST (0 or 1 - disabled by default). In the case below, the unconditional closure of Flow on TCP RST is enabled:

  ```
  --vr_uncond_close_flow_on_tcp_rst 1
  ```

- ---no-gro : (troubleshooting purpose) is used to disable GRO (Generic Receive Offload) on the DPDK vRouter data plane. In the case below, GRO is disabled:

  ```
  --no-gro
  ```

- ---no-gso : (troubleshooting purpose) is used to disable GSO (Generic Segmentation Offload) on DPDK vRouter data plane. In the case below, GSO is disabled:

  ```
  --no-gso
  ```

- ---no-mrgbuf : (troubleshooting purpose) is used to turn off mergeable buffers on DPDK vRouter data plane. In the case below, mergeable buffers are disabled:

  ```
  --no-mrgbuf
  ```

- --vr_dpdk_tx_ring_sz : (20.03 and later version) is used to define forwarding lcores TX Ring descriptor size (1024 by default). In the case below, TX Ring descriptor size has been set to 2048:

  ```
  --vr_dpdk_tx_ring_sz 2048
  ```

- --socket-mem: is used to define the amount of memory pre-allocated for contrail vRouter. In the case below, 1GB of huge-page memory is pre-allocated on NUMA node 0 and NUMA node 1:

  ```
  --socket-mem 1024,1024
  ```

- --vr_mempool_sz : is used to define mempool memory size. In the case below 128 MB mempool memory size is defined:

  ```
  --vr_mempool_sz 131072
  ```

- --dpdk_txd_sz : is used to define Physical NIC TX Ring descriptor size. In the case below 2048 bytes RX ring descriptor size is defined:

  ```
  --dpdk_txd_sz 2048
  ```

- --dpdk_rxd_sz : is used to define Physical NIC RX Ring descriptor size. In the case below 2048 bytes RX ring descriptor size is defined:

  ```
  --dpdk_rxd_sz 2048
  ```

These values, (especially --vr_mempool_sz, --dpdk_txd_sz and --dpdk_rxd_sz) as shown in Figure 4.5, have to be adjusted depending on:

- the inter NIC model used
- the number of NIC members of vhost0 bond
- the number of logical cores allocated to the vRouter

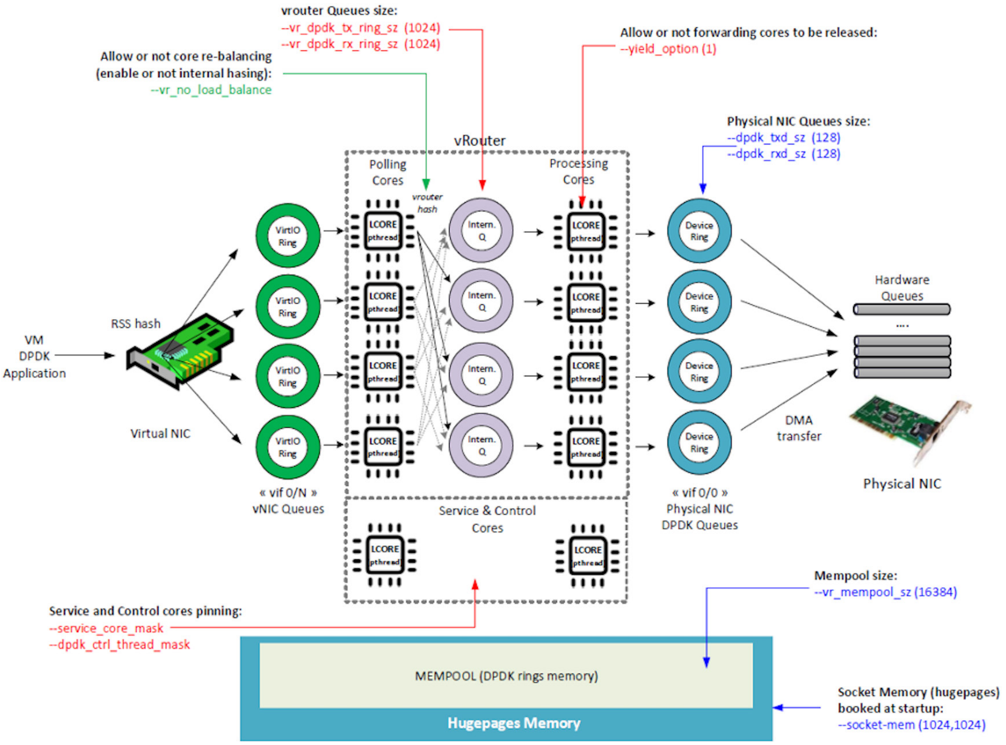


Figure 4.5 Overview of DPDK vRouter Fine Tuning Parameters

The --vr_no_load_balance provides significant boost of decreasing latency in packet processing. It is strongly recommended to use this the option if is possible.

The --vr_no_load_balance activation is also expecting:

- UDP encapsulation protocol (MPLSoUDP or VXLAN)
- multiqueue usage on Virtual Machine
- good traffic entropy and well balanced on queues (especially on Virtual Machine side)

Once --vr_no_load_balance is enabled there is no need to configure --vr_dpdk_tx_ring_sz and --vr_dpdk_rx_ring_sz as they are not used.

Chapter 5

Contrail Networking and Test Tools Installation

The previous chapters have gone through most of the important topics about SDN and about DPDK in general: DPDK vRouter architectures, vRouter packet processing details, and so on. When you read about these topics you may wonder how to get a running Contrail networking environment with a few DPDK vRouters in it, so you can play around, test those theories, and familiarize yourself with what you've learned. Indeed, those topics are important, but unfortunately they are not so straightforward. So even after we've put great effort into illustrating how they work, some of them may still sound confusing, especially when you get down to the implementation details.

This chapter and Chapter Six focus mostly on hands-on lab testing to verify some of the most important DPDK vRouter concepts and working mechanisms.

- In this chapter, we'll begin by introducing steps we've used to install the latest version of Contrail networking cluster.
- On top of that, we'll start to build a testing environment. That includes a few VMs running OPENFV PROX software. On each VM, based on its role, the PROX software is configured as either a traffic generator or a traffic receiver.
- In Chapter Six we'll introduce some of the commonly used DPDK tools, scripts, and log entries to provides useful information to help you understand how things run in a DPDK environment.
- At the end of Chapter Six, we'll go over some case studies. We'll use PROX and rapid we've installed to start different traffic patterns in the setup, and then use DPDK tools to analyze what we are seeing.

After reading these two chapters, you will have a deeper understanding of some of the main concepts covered in this book. Let's start with Contrail installation.

Contrail Installation

Installation Methods

We've been focusing on the DPDK vRouter that runs in each individual compute node, which basically runs in a relatively standalone mode. But if you look at the forwarding plane as a whole, those nodes are actually a distributed system. In fact, as discussed in Chapter 1, the whole TF cluster is a complex distributed system involving a lot more different software modules, especially in control plane. Again, each of the software modules can be a completely different distributed system by themselves.

The Cassandra database that the TF cluster uses is one such example. Explaining and understanding details about how things works in a distributed system is never easy, nor is the installation process. Don't be surprised if you run into some installation issues in your lab. Generally speaking, it is always much more efficient to follow a detailed, verified process with step-by-step instructions to *avoid* issues, rather than starting with a *try-and-see* mode and then trying to fix issues.

Currently, the TF cluster has been widely integrated with many major deployment systems and platforms. Therefore, depending on your environment, there can be totally different ways of installing the Contrail system. Here is an incomplete list of currently supported installation methods:

- Installing Contrail with OpenStack and Kolla Ansible
- Installing Contrail with RHOSP
- Installing Kubernetes Contrail Cluster using the Contrail Command UI
- Installing and provisioning Contrail VMware vRealize Orchestrator Plugin
- Installing a Standalone Red Hat OpenShift Container Platform 3.11 Cluster with Contrail Using Contrail OpenShift Deployer
- Installing a Nested Red Hat OpenShift Container Platform 3.11 Cluster Using Contrail Ansible Deployer
- Installing Contrail with OpenStack or Kubernetes by using Juju Charms

For example, in the second method listed above, you can install Contrail with Red Hat OpenStack platform director 13 (RHOSPd), which is a toolset based on the OpenStack project TripleO (OOO, or OpenStack on OpenStack). A TF environment built out of RHOSPd uses concepts of *undercloud* and *overcloud*. Basically, undercloud is a single server containing complete OpenStack components, whose role is just to deploy and manage an overcloud, which is a tenant-facing environment that hosts the *resulting* OpenStack and TF nodes. This deployment is currently used by many major service providers in production.

However, the installation process of such a deployment involves the understanding of RHOSPd, TripleO, and of lots of different types of network isolation topologies, which add many unnecessary complexities to our lab setup. These topics are enough for another book to cover. So in this section, we'll provide detailed steps about the first method: installing Contrail with OpenStack and Kolla Ansible.

Kolla is an OpenStack project that provides tools to build container images for OpenStack services. Kolla Ansible provides Ansible playbooks to deploy the Kolla images. The contrail-kolla-ansible playbook works in conjunction with contrail-ansible-deployer to install OpenStack and Contrail Networking containers.

This chapter's testbed is shown in Figure 5.1.

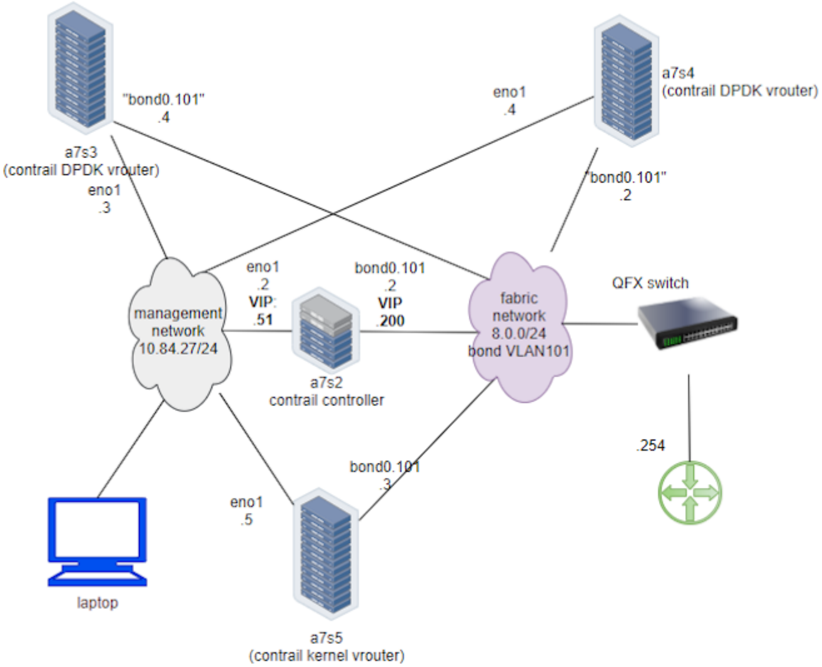


Figure 5.1 The Contrail DPDK vRouter Test Bed

Configure Bond and VLAN

To enable bond interface in centos, under `/etc/sysconfig/network-scripts/` of all the nodes where the bond interface is needed, add these configuration files:

bond	members

$ cat ifcfg-bond0
SUBCHANNELS=1,2,3
NM_CONTROLLED=no
BOOTPROTO=none
BONDING_OPTS="miimon=100 mode=802.3ad xmit_hash_policy=layer3+4"
DEVICE=bond0
BONDING_MASTER=yes
ONBOOT=yes
	$cat ifcfg-enp2s0f0
	HWADDR=00:1b:21:bb:f9:46
	SLAVE=yes
	NM_CONTROLLED=no
	BOOTPROTO=none
	MASTER=bond0
	DEVICE=enp2s0f0
	ONBOOT=yes

$cat ifcfg-bond0.101
HWADDR=00:1b:21:bb:f9:46
SLAVE=yes
NM_CONTROLLED=no
BOOTPROTO=none
MASTER=bond0
DEVICE=enp2s0f0
ONBOOT=yes
	$cat ifcfg-enp2s0f1
	HWADDR=00:1b:21:bb:f9:47
	SLAVE=yes
	NM_CONTROLLED=no
	BOOTPROTO=none
	MASTER=bond0
	DEVICE=enp2s0f1
	ONBOOT=yes

Then restart network service to invoke these configurations:

```
service network restart
```

Once the restart is successful, you should see bond0 interface appearing in all nodes with one of these IP addresses in each node: 8.0.0.1 to 8.0.0.4. Now you should have IP connectivity in both the management network and fabric network.

Next you'll need to install Ansible and use it to automate the rest of the installations. Most of Ansible's magic is performed through its playbooks, and configuration for all playbooks is done in a single file with a default name `instances.yaml`. This configuration file has multiple main sections. We'll go over some of the main parameters in this file and then introduce the steps to run the playbooks. Here's the configuration file for `instances.yaml`:

```
global_configuration:
  CONTAINER_REGISTRY: svl-artifactory.juniper.net/contrail-nightly
  REGISTRY_PRIVATE_INSECURE: True
provider_config:
  bms:
    ssh_pwd: c0ntrail123
    ssh_user: root
    ntpserver: 10.84.5.100
    domainsuffix: englab.juniper.net
instances:
  a7s2:
    provider: bms
    ip: 10.84.27.2
    roles:
      openstack_control:
      openstack_network:
      openstack_storage:
      openstack_monitoring:
      config_database:
      config:
      control:
      analytics_database:
      analytics:
      webui:
  a7s3:
    provider: bms
    ip: 10.84.27.3
    ssh_user: root
    ssh_pwd: c0ntrail123
    roles:
      openstack_compute:
      vrouter:
        PHYSICAL_INTERFACE: bond0.101
        CPU_CORE_MASK: 0x1fe
        DPDK_UIO_DRIVER: uio_pci_generic
        HUGE_PAGES: 32000
        AGENT_MODE: dpdk
  a7s4:
    provider: bms
    ip: 10.84.27.4
    ssh_user: root
    ssh_pwd: c0ntrail123
    roles:
      openstack_compute:
      vrouter:
        PHYSICAL_INTERFACE: bond0.101
        CPU_CORE_MASK: 0x1fe
        DPDK_UIO_DRIVER: uio_pci_generic
        HUGE_PAGES: 32000
        AGENT_MODE: dpdk
  a7s5:
    provider: bms
    ip: 10.84.27.5
    ssh_user: root
    ssh_pwd: c0ntrail123
    roles:
      openstack_compute:
      vrouter:
        PHYSICAL_INTERFACE: bond0.101
contrail_configuration:
  CONTRAIL_VERSION: 2008.108
  OPENSTACK_VERSION: rocky
  CLOUD_ORCHESTRATOR: openstack
  CONTROLLER_NODES: 8.0.0.1
  OPENSTACK_NODES: 8.0.0.1
  CONTROL_NODES: 8.0.0.1
  KEYSTONE_AUTH_HOST: 8.0.0.200
  KEYSTONE_AUTH_ADMIN_PASSWORD: c0ntrail123
  RABBITMQ_NODE_PORT: 5673
  KEYSTONE_AUTH_URL_VERSION: /v3
  IPFABRIC_SERVICE_IP: 8.0.0.200
  VROUTER_GATEWAY: 8.0.0.254
```

```
73   two_interface: true
74   ENCAP_PRIORITY: VXLAN,MPLSoUDP,MPLSoGRE
75   AUTH_MODE: keystone
76   CONFIG_API_VIP: 10.84.27.51
77   ssh_user: root
78   ssh_pwd: c0ntrail123
79   METADATA_PROXY_SECRET: c0ntrail123
80   CONFIG_NODEMGR__DEFAULTS__minimum_diskGB: 2
81   CONFIG_DATABASE_NODEMGR__DEFAULTS__minimum_diskGB: 2
82   DATABASE_NODEMGR__DEFAULTS__minimum_diskGB: 2
83   XMPP_SSL_ENABLE: no
84   LOG_LEVEL: SYS_DEBUG
85   AAA_MODE: rbac
86 kolla_config:
87   kolla_globals:
88     kolla_internal_vip_address: 8.0.0.200
89     kolla_external_vip_address: 10.84.27.51
90     contrail_api_interface_address: 8.0.0.1
91     keepalived_virtual_router_id: 111
92     enable_haproxy: yes
93     enable_ironic: no
94     enable_swift: no
95   kolla_passwords:
96     keystone_admin_password: c0ntrail123
97     metadata_secret: c0ntrail123
98     keystone_admin_password: c0ntrail123
```

Definitions for the configuration are:

- line 1-3: global configurations
- line 2: the registry from which to pull Contrail containers
- line 3: set to True if containers that are pulled from a private registry (named CONTAINER_REGISTRY) are not accessible
- line 4-9: provider-specific settings
- line 5: bare metal server (bms) environment
- line 6-9: ssh password, user name, ntpserver, and domainsuffix
- line 10-59: Instances means the node on which the containers will be launched. Here we defined four nodes, named a7s2, a7s3, a7s4, and a7s5, respectively.
- line 11-24: this is the configuration section for node a7s2
- line 12-14: this server's provider type (baremetal server), ip address, and roles
- line 14-24: roles of containers that will be installed in this node, according to the configuration, this server a7s2 will be installed with all "controller" software modules, in both OpenStack and Contrail
- line 25-37: parameters for our first DPDK compute node. OpenStack compute components and Contrail vRouter will be installed in it.

- line 33: under vRouter, bond0.101 will be the PHYSICAL_INTERFACE, which is also called a fabric interface, which carries all the underlay data packets.
- line 34-37: these are the DPDK specific configurations. For kernel based vRouter these are not needed.
- line 34: CPU_CORE_MASK defines DPDK vRouter forwarding lcore pinning. The hex code 0x1fe, if converted to its binary format, is 0b000111111110. That means physical CPU core NO.1 through 8 is used as forwarding lcores lcore#10 through lcore#17.
- line 35: DPDK_UIO_DRIVER specifies which UIO driver to use. Here it is uio_pci_generic. Another popular option is UIO driver is igb_uio.
- line 36: HUGE_PAGES defines number of huge pages. Here we allocate 32000 huge pages. Considering page size 2M it will be 64G memory usage in total. free -h command output in compute node will confirm this.
- line 37: agent mode set to dpdk
- line 38-50: the second DPDK vRouter on server a7s4
- line 51-59: defines the third vRouter, this one is a kernel-based, so we don't need any DPDK specific parameters
- line 60-85: contrail_configuration section contains parameters for Contrail services
- line 61-62: Contrail and OpenStack versions
- line 63: the cloud orchestrator. It can be OpenStack or vcenter. Our setup is with OpenStack only.
- line 64-66: who is the controller node. In our setup both OpenStack and contrail controllers are installed in same node.
- line 71, 76: these are the two virtual IPs configured
- line 80-82: these are needed only for lab setup. Without these parameters, contrail-status command will print a warning to indicate that the storage space is not big enough.
- line 86-98: the parameters for Kolla
- line 87-94: refers to OpenStack service
- line 88-89: VIPs configured for management and data control network, respectively. One usage of these VIPs is to make it possible to access the OpenStack horizon service (webUI) from management network, by default all OpenStack services listen on the IP in data/ctrl network. With these VIPs configured and used by keepalived, HAproxy can forward the access request coming from the management network to the Horizon service.

Installation Steps

Once the YAML file is carefully prepared, the installation process is relatively easy. Basically you select one node as the deployment node, the node from where you want to automate the installation of all other nodes. In practice, use the controller node as the deployment node.

In this node you need to install some prerequisite software, such as python libraries, `ansible`, `git`, etc., and the python modules (`python-wheel`) that are used by Ansible, and Ansible is our automation tool. git is used to clone a github repository which includes all Ansible playbooks. Then you use Ansible to automate the software installation in all the nodes based on the playbook and your configuration file `instances.yaml`. The details start here:

1. Install prerequisite packages on deployment node, in this case, it's the controller `a2s2`:

```
yum -y remove python-netaddr
yum -y install epel-release python-pip gcc python-cffi python-devel bcrypt==3.1.7 sshpass python-
wheel
pip install wheel requests
yum -y install git
pip install ansible==2.5.2.0
```

2. Use git to clone install the Ansible deployer folder into deployment node:

```
git clone http://github.com/tungstenfabric/tf-ansible-deployer
cd tf-ansible-deployer
```

3. Place the prepared configuration file `instances.yaml` to `tf-ansible-deployer/config`:

4. Install Contrail with Ansible:

```
ansible-playbook -i inventory/ -e orchestrator=openstack playbooks/configure_instances.yml
ansible-playbook -i inventory/ playbooks/install_openstack.yml
ansible-playbook -i inventory/ -e orchestrator=openstack playbooks/install_contrail.yml
```

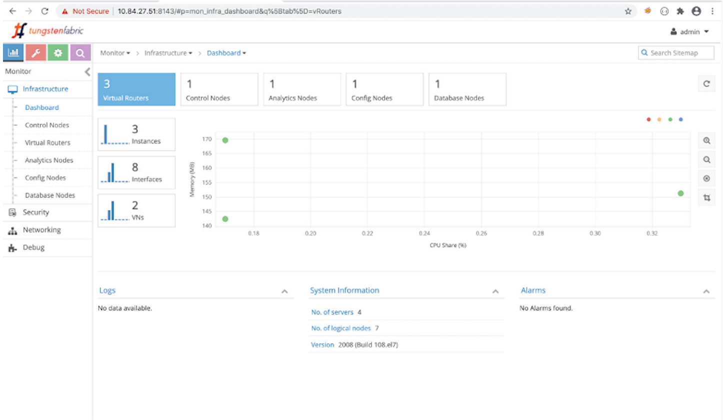
5. Install the OpenStack client:

```
pip install --ignore-installed python-openstackclient python-ironicclient openstack-heat
```

After everything loads, you will have an up and running four-node Contrail cluster (one controller node and three vRouter/compute nodes). You can log in to the setup via a webUI or ssh session to check system status.

Post-installation Verification

Here is the Contrail web UI for a working setup in Figure 5.2.



No data available.

No. of servers 4

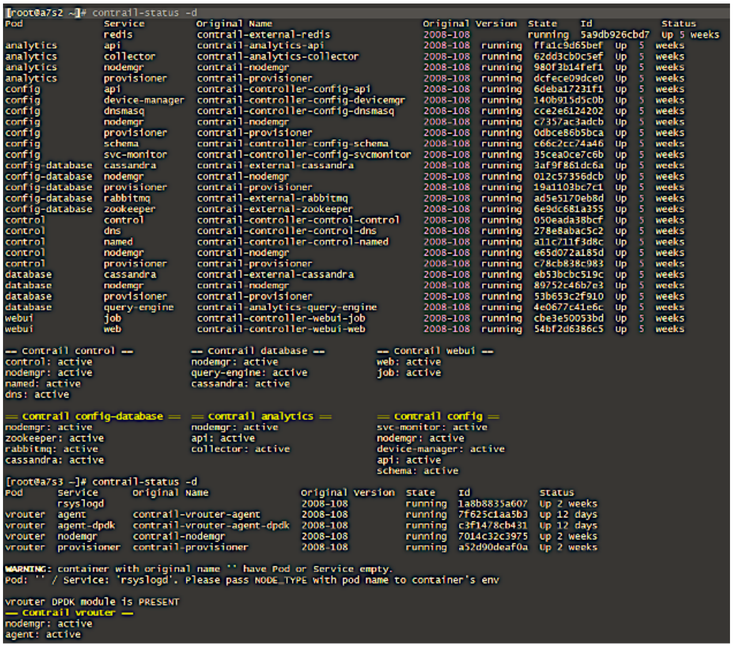
No. of logical nodes 7

Version 2008 (Build 108.el7)

No Alarms found.

Figure 5.2 Contrail WebUI Dashboard

You can also log in to each individual node with SSH, and the run `contrail-status` command to verify the running status of all the components as shown in Figure 5.3.



```
[root@a7s2 ~]# contrail-status -d
Pod              Service         Original Name                          Original Version  State    Id             Status
                 redis           contrail-external-redis                2008-108          running  5a9db926cbd7   Up 5 weeks
analytics        api             contrail-analytics-api                 2008-108  running  ffa1c9d65bef  Up  5  weeks
analytics        collector       contrail-analytics-collector           2008-108  running  62dd3cb0c5ef  Up  5  weeks
analytics        nodemgr         contrail-nodemgr                       2008-108  running  980f3b14fef1  Up  5  weeks
analytics        provisioner     contrail-provisioner                   2008-108  running  dcfece09dce0  Up  5  weeks
config           api             contrail-controller-config-api         2008-108  running  6deba17231f1  Up  5  weeks
config           device-manager  contrail-controller-config-devicemgr   2008-108  running  140b915d5c0b  Up  5  weeks
config           dnsmasq         contrail-controller-config-dnsmasq     2008-108  running  cce2e6124202  Up  5  weeks
config           nodemgr         contrail-nodemgr                       2008-108  running  c7357ac3adcb  Up  5  weeks
config           provisioner     contrail-provisioner                   2008-108  running  0dbce86b5bca  Up  5  weeks
config           schema          contrail-controller-config-schema      2008-108  running  c66c2cc74a46  Up  5  weeks
config           svc-monitor     contrail-controller-config-svcmonitor  2008-108  running  35cea0ce7c6b  Up  5  weeks
config-database  cassandra       contrail-external-cassandra            2008-108  running  3af9f861dc6a  Up  5  weeks
config-database  nodemgr         contrail-nodemgr                       2008-108  running  012c57356dcb  Up  5  weeks
config-database  provisioner     contrail-provisioner                   2008-108  running  19a1103bc7c1  Up  5  weeks
config-database  rabbitmq        contrail-external-rabbitmq             2008-108  running  ad5e5170eb8d  Up  5  weeks
config-database  zookeeper       contrail-external-zookeeper            2008-108  running  6e9dc681a355  Up  5  weeks
control          control         contrail-controller-control-control    2008-108  running  050eada38bcf  Up  5  weeks
control          dns             contrail-controller-control-dns        2008-108  running  278e8abac5c2  Up  5  weeks
control          named           contrail-controller-control-named      2008-108  running  a11c711f3d8c  Up  5  weeks
control          nodemgr         contrail-nodemgr                       2008-108  running  e65d072a185d  Up  5  weeks
control          provisioner     contrail-provisioner                   2008-108  running  c78cb838c983  Up  5  weeks
database         cassandra       contrail-external-cassandra            2008-108  running  eb53bcbc519c  Up  5  weeks
database         nodemgr         contrail-nodemgr                       2008-108  running  89752c46b7e3  Up  5  weeks
database         provisioner     contrail-provisioner                   2008-108  running  53b653c2f910  Up  5  weeks
database         query-engine    contrail-analytics-query-engine        2008-108  running  4e0677c41e6c  Up  5  weeks
webui            job             contrail-controller-webui-job          2008-108  running  cbe3e50053bd  Up  5  weeks
webui            web             contrail-controller-webui-web          2008-108  running  54bf2d6386c5  Up  5  weeks

== Contrail control ==           == Contrail database ==          == Contrail webui ==
control: active                  nodemgr: active                  web: active
nodemgr: active                  query-engine: active             job: active
named: active                    cassandra: active
dns: active

== Contrail config-database ==   == Contrail analytics ==         == Contrail config ==
nodemgr: active                  nodemgr: active                  svc-monitor: active
zookeeper: active                api: active                      nodemgr: active
rabbitmq: active                 collector: active                device-manager: active
cassandra: active                                                 api: active
                                                                  schema: active
[root@a7s3 ~]# contrail-status -d
Pod      Service      Original Name                Original Version  State    Id            Status
         rsyslogd                                  2008-108          running  1a8b8835a607  Up 2 weeks
vrouter  agent        contrail-vrouter-agent       2008-108          running  7f625c1aa5b3  Up 12 days
vrouter  agent-dpdk   contrail-vrouter-agent-dpdk  2008-108          running  c3f1478cb431  Up 12 days
vrouter  nodemgr      contrail-nodemgr             2008-108          running  7014c32c3975  Up 2 weeks
vrouter  provisioner  contrail-provisioner         2008-108          running  a52d90deaf0a  Up 2 weeks

WARNING: container with original name '' have Pod or Service empty.
Pod: '' / Service: 'rsyslogd'. Please pass NODE_TYPE with pod name to container's env

vrouter DPDK module is PRESENT
== Contrail vrouter ==
nodemgr: active
agent: active
```

Figure 5.3 Contrail-Status Command Output

If everything works, congratulations! You now have your own lab to play in. Now let's go over the steps of setting up testing tools to send and receive traffic - the PROX and rapid script.

DPDK vRouter Test Tools

PROX

PROX (Packet pROcessing eXecution Engine) is an OPNFV project application built on top of DPDK. It is capable of performing various operations on packets in a highly configurable manner. It also supports performance statistics that can be used for performance investigations. Because of the rich feature set it supports, it can be used to create flexible software architectures through small and readable configuration files. This section introduces you to how to use it to test vRouter performance in DPDK environments.

In a typical test you need two VMs running PROX. VM1 generates packets, sending them to VM2, which will perform a swap operation on all packets so that they are sent back to VM1.

- traffic generator VM (gen VM)
- traffic receiver and looping VM (swap VM, or loop VM)

This book calls them *gen* and *swap* VM, respectively. One special feature used here is that the swap PROX is configured in such a way that once it receives the packets sent from the generator, it will swap, or loop, them back to the generator VM so the latter can collect them and calculate how much traffic was forwarded by the DUT - in this case it's the DPDK vRouter.

Rapid

Rapid (Rapid Automated Performance Indication for Data Plane) is a group of wrapper scripts interacting with PROX to simplify and automate the configuration of PROX. It's a set of files and scripts offering an even easier way to do sanity checks of the data plane performance.

Rapid is both powerful and configurable. A typical workflow works as follows:

- A script name `runrapid.py` sends the proper configuration files to the gen and swap VMs involved in the testing, so each one knows its role (generator or swapper) in the test.
- It then starts PROX within both VMs, as generator and swapper, respectively.
- While the test is going on it collects the results from PROX. Results are visible onscreen and logged in the log and csv files.
- The same tests will be done for different packet sizes and different amounts of flows, under certain latency and packet drop rates.

The rapid scripts are typically installed in a third VM, called *jump* VM in this book. The purpose of this VM is to control the traffic generator to start, stop, and pause the test, as well as to collect statistics.

PROX and Rapid Test Setup

A typical PROX and Rapid testing setup looks like Figure 5.4.

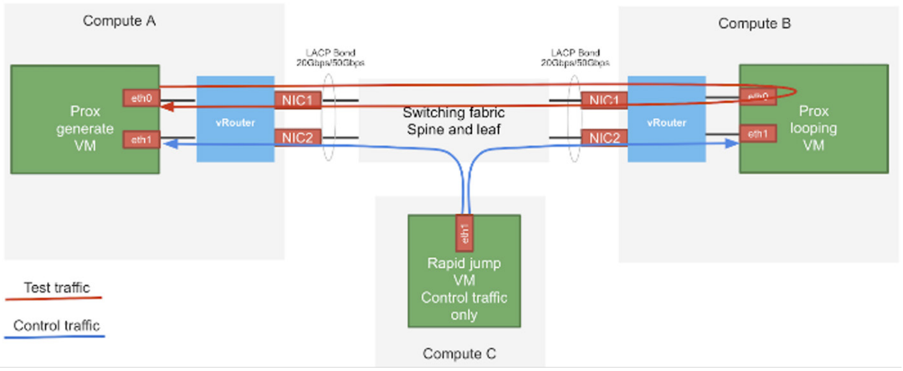


Figure 5.4 PROX and Rapid Test Diagram

The test consists of three compute nodes running the mentioned three VMs, respectively:

- PROX generated VM runs on compute-A: This is the traffic generator VM for traffic generation.
- PROX looping VM runs on compute-B: This is the swap VM for looping traffic out of the same interface where it came in. This is the DUT (device under test) where the vRouter is running.
- Rapid jump VM runs on compute-C: This is the VM where rapid scripts are installed; it is responsible for control traffic generation and collecting results.

Hardware Requirements

Here's a brief summary of hardware requirements for different VMs:

- Swap VM: This is where the DUT (vRouter) is located. Based on the test requirements, a specific amount of hardware resources should be allocated and all applications that could unnecessarily consume the hardware resources should be removed.

- Gen VM: In order to saturate the DUTs, the traffic generator VM and the compute should allocate much more CPU resources than the DUT.
- Jump VM: No high speed VM is required, and it can be run on the kernel or the DPDK compute.
- Optionally, the generator and receiver computes can run on a bonded interface configured with 802.3ad LACP mode. This is a common configuration recommended in a practical environment.

The PROX setup in this book's lab is shown in Figure 5.5.

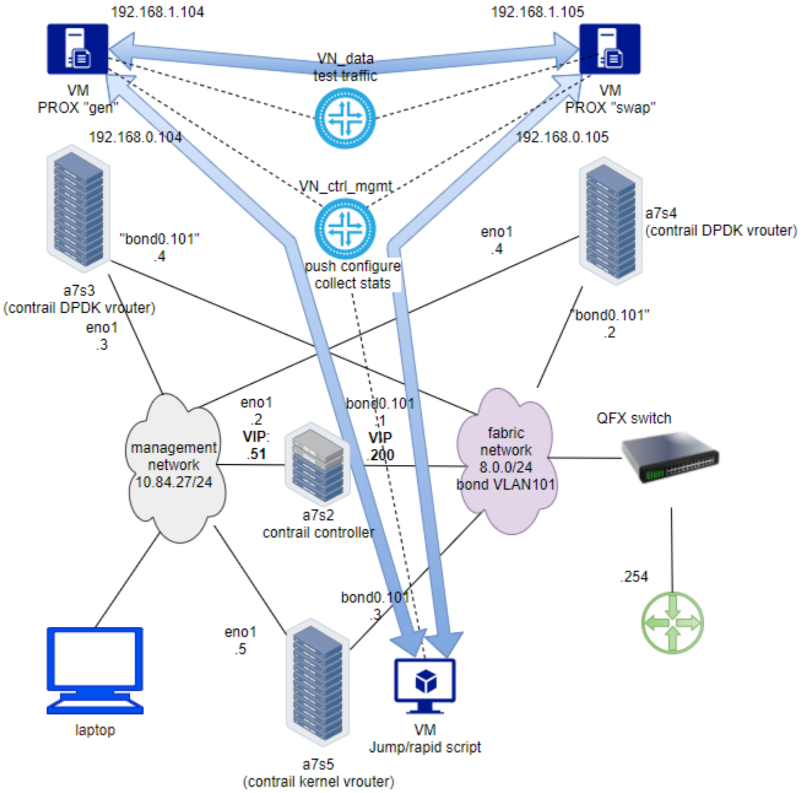


Figure 5.5 The PROX Setup in the Book's Lab

By default, multi-queue is enabled on both the PROX gen and swap VMs via OpenStack image property. You can refer to Chapter 3 for more details about the multi-queue feature and its configurations. Additionally, rapid scripts also provide CPU pinning to protect PROX PMDs against CPU-stealing by other processes and the VM OS.

Installation

Creating OpenStack Resources

As mentioned earlier, to perform the test we need two VMs both running PROX. One sending traffic and the other receiving and swapping it back. The same exact PROX application is running but here with different configuration files.

Apparently, the IP level connectivity is required in order for the two VMs to be able to exchange packets with each other. In this case, the two VMs will be spawned by OpenStack Nova. Needless to say, all supporting objects and resources associated with the VMs, like IPAM, subnet, virtual-network, and VM flavor (size of CPU/memory/storage/etc.), also need to be created out of OpenStack infrastructure, either from horizon webUI or OpenStack CLIs. A quick list of common tasks are listed here:

- create IPAMs/subnets/virtual networks
- create flavors
- create images
- create host aggregates
- create instances
- create key-pairs

On top of these, installing PROX inside of the VMs, like with many other opensource projects, often requires downloading the source code and compiling it on your platform. That means you download the PROX source codes, compile them to get the execute, then configure and run the application. In this section we introduce you to how PROX is installed in the setup we built for this book.

MORE? You can find more details in PROX website here: https://wiki.opnfv.org/display/SAM/PROX+installation.

The software and CPU model used here are:

```
[root@a7s3 ~]# cat /etc/centos-release
CentOS Linux release 7.7.1908 (Core)
[root@a7s3 ~]# uname -a
Linux a7s3 3.10.0-1062.el7.x86_64 #1 SMP Wed Aug 7 18:08:02 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux
[root@a7s3 ~]# lscpu | grep Model
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2620 v2 @ 2.10GHz
```

In our lab setup the VM OS is the same as the host, and the emulated CPU Model is Intel Xeon E3-12xx:

```
[root@stack2-gen ~]# cat /etc/centos-release
CentOS Linux release 7.7.1908 (Core)
```

```
[root@stack2-gen ~]# uname -a
Linux stack2-gen.novalocal 3.10.0-1062.18.1.el7.
x86_64 #1 SMP Tue Mar 17 23:49:17 UTC 2020 x86_64 x86_64 x86_64 GNU/Linux
[root@stack2-gen ~]# lscpu | grep -i Model
Model:                 58
Model name:            Intel Xeon E3-12xx v2 (Ivy Bridge, IBRS)
```

NOTE There is a good chance that your servers and VMs have totally different hardware and software architectures. The steps here are tested and working fine in the book's lab setup, but depending on your environment, you may run into some errors. Check PROX online documentation for more detailed instructions.

Compiling and Building the DPDK

PROX is a DPDK application. When running, it connects to the DPDK libraries to implement most of its features. Therefore, to build it you need a DPDK environment.

You can either build it inside of the VM where it runs, or build it directly in the host environment where the VM got spawned and copy it into the VM. The steps to build the DPDK start with installing libraries that are required for compiling:

```
sudo yum install numactl-devel net-tools wget gcc unzip libpcap-devel \
     ncurses-devel libedit-devel pciutils lua-devel kernel-devel
```

Add the following to the end of ~/.bashrc file:

```
export RTE_SDK=/root/dpdk
export RTE_TARGET=x86_64-native-linuxapp-gcc
export RTE_KERNELDIR=/lib/modules/`ls /lib/modules`/build
export RTE_UNBIND=$RTE_SDK/tools/dpdk_nic_bind.py
```

Source that file:

```
.~/.bashrc
```

Build DPDK:

```
git clone https://github.com/DPDK/dpdk
cd dpdk
git checkout v19.11
make install T=$RTE_TARGET
```

NOTE The stable and recommended version of DPDK at the time of writing this book is 19.11.

Compiling PROX

Now with the DPDK libraries built, let's start to download, extract, and build the PROX application. Here are the steps:

```
git clone https://github.com/opnfv/samplevnf
cd samplevnf/VNFs/DPPD-PROX
git checkout origin/master
make
```

When make succeeds, the compiled binary PROX will be available in the `build` folder of the current directory. (We'll demonstrate this shortly.)

Configuration Files

The set of sample configuration files can be found in: `./config` folder. Sample configs of PROX functioning as the generator is available `in ./gen/` folder. Assuming that the current directory is where you've just built PROX, you can just launch PROX with a proper configuration file:

```
./build/prox -f <prox configuration file>
```

When it runs, a ncurse-based UI will pop up, and through it you will see updates about the running states in real time. We'll give an example on this shortly.

Rapid Installation

Rapid scripts can be downloaded from here: https://github.com/opnfv/samplevnf/tree/master/VNFs/DPPD-PROX/helper-scripts/rapid .The scripts were developed in Python, so you can run them directly with no need to compile.

Installation: Heat Automation

Those are the steps of manually compiling PROX from source code.

Now here's a list of tasks to create all the necessary objects required by the VMs from OpenStack. Doing this one time is not a big deal, but suppose you are working in a dynamic environment where you often need to:

- quickly build up a PROX test environment to do some tests.
- tear it down after the test is finished.
- redo the same test all over again in another cluster.

Repeating these manual steps will become tedious and even painful. To simplify the building, creation, and configuration of PROX, as well as creating all necessary OpenStack resources, the number one choice for automation is *heat*. With *heat* all tasks are typically programmed in a template file, which calls all parameters from another environment file. In the github site for this book we provide all sample template files, as well as an environment file and associated scripts which are tested and proven to be working fine, at least in our setup. You can use them as a starting point, then make necessary customizations based on your environment to build your own automation. The VM, where the tools are running, including the rapid scripts and PROX DPDK applications pre-compiled on it, also has been built as an image.

With all these automations carefully designed and tested, what we need to do now becomes much more simple:

- download this pre-built image and load it into OpenStack image service
- create the *heat* stack with the sample template files

If everything goes well, you will have your whole PROX testing environment available in just a few minutes. The detail steps are listed below:

1. Prepare pre-built VM image, heat template files, and scripts:

- VM image: this is the image with PROX compiled, as shown in the previous section.

Adjust the heat template, environment variables, and automation scripts based on your environment: (These files are available in this https://github.com/pinggit/dpdk-contrail-book.)

- environment.yaml
- build-rapid.yml
- configure.rapid.sh

2. Load rapid image into OpenStack glance service:

```
openstack image create --disk-format qcow2 --container-format bare --public --file rapidVM.
qcow2 rapidVM-1908
openstack image set --property hw_vif_multiqueue_enabled=true rapidVM-1908
```

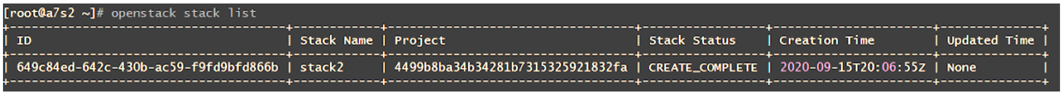
3. (Optionally) If you're using ceph backend:

```
qemu-img convert rapidVM-1908.qcow2 rapidVM-1908.raw
openstack image create --disk-format raw --container-format bare --public --file rapidVM.
raw rapidVM-1908
openstack image set --property hw_vif_multiqueue_enabled=true rapidVM-1908
```

4. Create heat stack with the prepared yaml files:

```
openstack stack create -t build-rapid.yml -e environment.yaml stack2
```

Wait for a few minutes and use the `openstack stack list` command to check the stack creation progress shown in Figure 5.6.



```
[root@a7s2 ~]# openstack stack list
+--------------------------------------+------------+----------------------------------+-----------------+----------------------+--------------+
| ID                                   | Stack Name | Project                          | Stack Status    | Creation Time        | Updated Time |
+--------------------------------------+------------+----------------------------------+-----------------+----------------------+--------------+
| 649c84ed-642c-430b-ac59-f9fd9bfd866b | stack2     | 4499b8ba34b34281b7315325921832fa | CREATE_COMPLETE | 2020-09-15T20:06:55Z | None         |
+--------------------------------------+------------+----------------------------------+-----------------+----------------------+--------------+
```

Figure 5.6 OpenStack Stack List

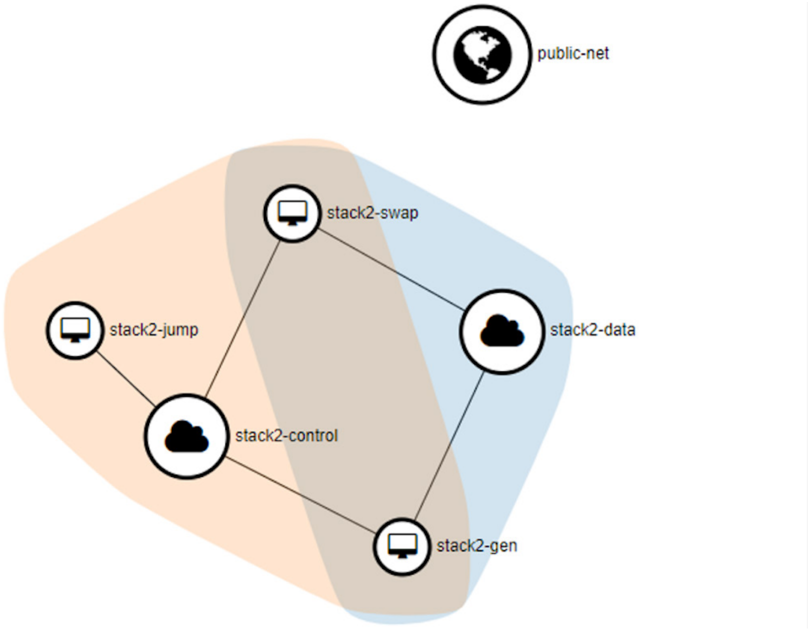
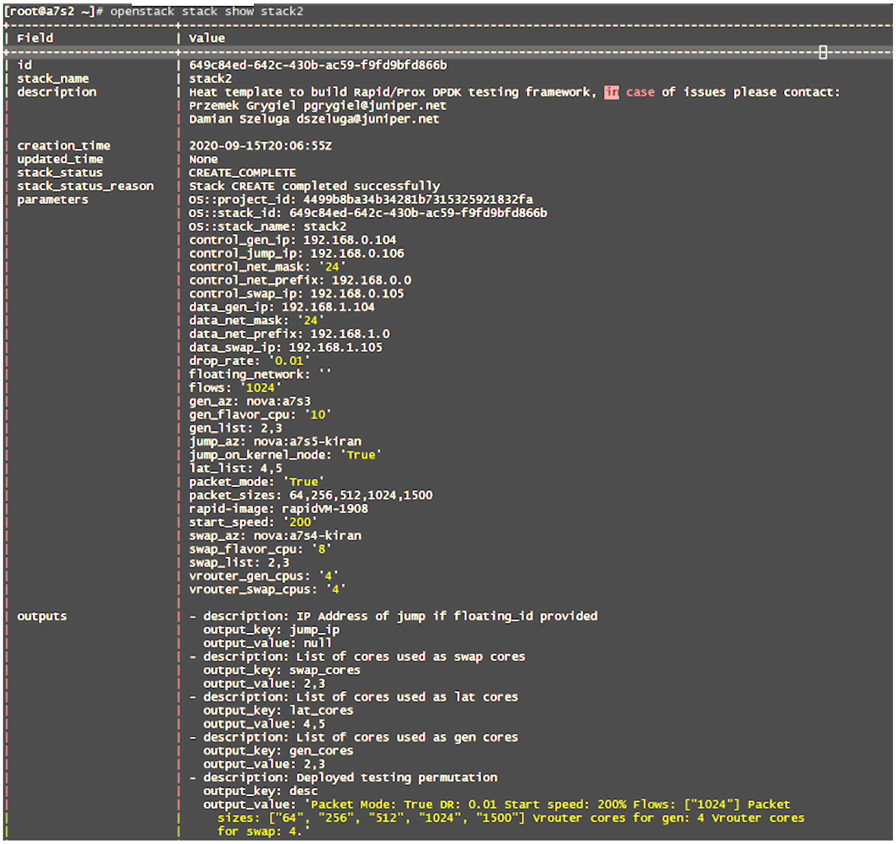


Figure 5.7 OpenStack Topology (graph)

5. Once loaded you can use these different sub-commands (see Figure 5.7) of the `openstack stack` command to retrieve the parameters of the stack components (see Figure 5.8):

```
openstack stack list STACK
openstack stack resource list
openstack stack resource list --filter type=OS::Nova::Server
openstack stack show STACK
openstack stack output show STACK
```



```
[root@a7s2 ~]# openstack stack show stack2
+------------------------+------------------------------------------------------------------------------------------
| Field                  | Value
+------------------------+------------------------------------------------------------------------------------------
| id                     | 649c84ed-642c-430b-ac59-f9fd9bfd866b
| stack_name             | stack2
| description            | Heat template to build Rapid/Prox DPDK testing framework, in case of issues please contact:
|                        | Przemek Grygiel pgrygiel@juniper.net
|                        | Damian Szeluga dszeluga@juniper.net
|                        |
| creation_time          | 2020-09-15T20:06:55Z
| updated_time           | None
| stack_status           | CREATE_COMPLETE
| stack_status_reason    | Stack CREATE completed successfully
| parameters             | OS::project_id: 4499b8ba34b34281b7315325921832fa
|                        | OS::stack_id: 649c84ed-642c-430b-ac59-f9fd9bfd866b
|                        | OS::stack_name: stack2
|                        | control_gen_ip: 192.168.0.104
|                        | control_jump_ip: 192.168.0.106
|                        | control_net_mask: '24'
|                        | control_net_prefix: 192.168.0.0
|                        | control_swap_ip: 192.168.0.105
|                        | data_gen_ip: 192.168.1.104
|                        | data_net_mask: '24'
|                        | data_net_prefix: 192.168.1.0
|                        | data_swap_ip: 192.168.1.105
|                        | drop_rate: '0.01'
|                        | floating_network: ''
|                        | flows: '1024'
|                        | gen_az: nova:a7s3
|                        | gen_flavor_cpu: '10'
|                        | gen_list: 2,3
|                        | jump_az: nova:a7s5-kiran
|                        | jump_on_kernel_node: 'True'
|                        | lat_list: 4,5
|                        | packet_mode: 'True'
|                        | packet_sizes: 64,256,512,1024,1500
|                        | rapid-image: rapidVM-1908
|                        | start_speed: '200'
|                        | swap_az: nova:a7s4-kiran
|                        | swap_flavor_cpu: '8'
|                        | swap_list: 2,3
|                        | vrouter_gen_cpus: '4'
|                        | vrouter_swap_cpus: '4'
|                        |
| outputs                | - description: IP Address of jump if floating_id provided
|                        |   output_key: jump_ip
|                        |   output_value: null
|                        | - description: List of cores used as swap cores
|                        |   output_key: swap_cores
|                        |   output_value: 2,3
|                        | - description: List of cores used as lat cores
|                        |   output_key: lat_cores
|                        |   output_value: 4,5
|                        | - description: List of cores used as gen cores
|                        |   output_key: gen_cores
|                        |   output_value: 2,3
|                        | - description: Deployed testing permutation
|                        |   output_key: desc
|                        |   output_value: 'Packet Mode: True DR: 0.01 Start speed: 200% Flows: ["1024"] Packet
|                        |     sizes: ["64", "256", "512", "1024", "1500"] Vrouter cores for gen: 4 Vrouter cores
|                        |     for swap: 4.'
```

Figure 5.8 OpenStack Stack Show STACK

Log In to the VMs

All three VMs, once up and running, will inherit the same log in credentials defined in the heat template and scripts. There are a few common ways to access a VM running in a specific compute node in Contrail/OpenStack integration environments:

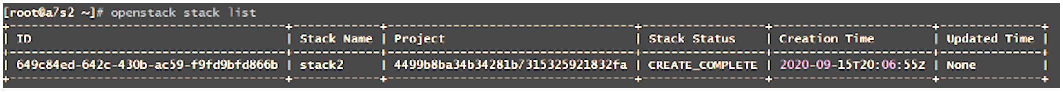
- floating IP: This is a routable IP address that is visible from outside of the cluster that maps to an internal IP of the VM. Once the VM is launched, you can log in to a specific VMs with this IP address from anywhere that is able to reach the IP.
- virsh console: virsh provides access to the VM console. This does not require any IP address to be configured.
- meta_ip_address: This is a non-routable private IP visible only from a specific compute. This IP address is automatically generated and mapped to the VM's tap interface IP.

In our test we didn't configure any floating IP, so we will use console and meta_ip_address to access the VM. To access the VM console use the `virsh console` command from `nova_libvirt` docker in the compute node:

```
[root@a7s3 ~]# docker exec -it nova_libvirt virsh list
 Id    Name                           State
----------------------------------------------------

 2     instance-00000041              running
[root@a7s3 ~]# docker exec -it nova_libvirt virsh console 2
Connected to domain instance-00000041
Escape character is ^]
CentOS Linux 7 (Core)
Kernel 3.10.0-1062.18.1.el7.x86_64 on an x86_64
stack2-gen login: root
Password:
Last login: Fri Sep 25 17:31:21 from 192.168.0.2
[root@stack2-gen ~]#
```

Compared with the console, an SSH session is usually preferred. Let's take a look at each VM's allocated interface IPs with openstack server list (as seen in Figure 5.9) command:



```
[root@a7s2 ~]# openstack stack list
+--------------------------------------+------------+----------------------------------+-----------------+----------------------+--------------+
| ID                                   | Stack Name | Project                          | Stack Status    | Creation Time        | Updated Time |
+--------------------------------------+------------+----------------------------------+-----------------+----------------------+--------------+
| 649c84ed-642c-430b-ac59-f9fd9bfd866b | stack2     | 4499b8ba34b34281b7315325921832fa | CREATE_COMPLETE | 2020-09-15T20:06:55Z | None         |
+--------------------------------------+------------+----------------------------------+-----------------+----------------------+--------------+
```

Figure 5.9 OpenStack Server List

Let's examine our jump VM `stack2-jump` for a moment. OpenStack allocated an IP address `192.168.0.106` to its tap interface from the `stack2-control virtual`-network. However, this IP address is not directly reachable from the host. In order to SSH into the VM, you need to first locate the `meta_ip_address` allocated to the VM's tap interface, or more specifically, the `vif` interface in the vRouter. You can use vRouter `vif` command to confirm which `vif` interface has this IP:

```
[root@a7s5-kiran ~]# contrail-tools vif -l | grep -B2 -A6 192.168.0.106

vif0/3      OS: tap0160123b-14 NH: 28
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.0.106
            Vrf:2 Mcast Vrf:2 Flags:PL3L2DEr QOS:-1 Ref:6
            RX packets:47246  bytes:2362255 errors:0
            TX packets:42996  bytes:2133684 errors:0
            ISID: 0 Bmac: 02:01:60:12:3b:14
            Drops:3553
```

Good, `vif0/3` has the IP, so this `vif` connects to the tap interface of our jump VM. In Contrail vRouter, for each `vif` there is also a hidden `meta_data_ip` of 169.254.0.N, where N is the same number as the number in the interface vif0/N. Therefore, in this case, the `meta_data_ip` is 169.254.0.3. Let's try to start an SSH session into it:

```
[root@a7s5-kiran ~]# ssh 169.254.0.3
Password:
```

```
Last login: Wed Sep 23 11:13:58 2020
[root@stack2-jump ~]#
```

It works. The benefit of this approach is that not only is the interaction with the VM much faster, but it also supports file copies with the `scp` tool. Remember, in many cases the VM does not have any internet connection, so in case you need to copy files into (or out of) the VM, the `meta_data_ip` method will be especially useful.

Run Rapid Automation: runrapid.py

With the stack created and all VMs up and running, we can now talk about how to run the test with rapid. Remember rapid is installed in the jump VM, so we'll need to execute the script from there.

On the jump VM, go to `/root/prox/helper-scripts/rapid/` folder, where you can locate a python script named runrapid.py. To run the test you can just run it without any other parameters:

```
cd /root/prox/helper-scripts/rapid/
./runrapid.py
```

This is a symbolic link, by default this rapid folder links to: `/opt/openstackrapid/samplevnf/VNFs/DPPD-PROX/helper-scripts/rapid/`.

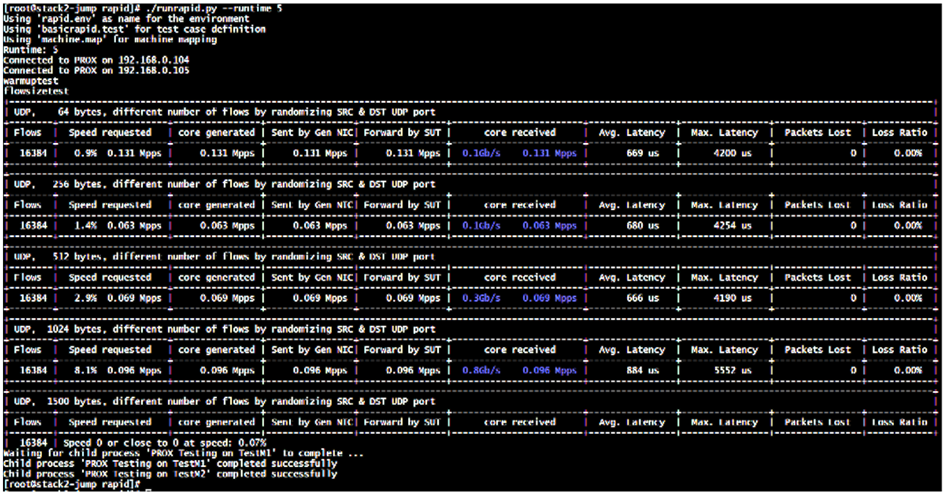
This will start rapid script and send traffic for ten seconds by default. the period of time for sending traffic can be adjusted by the `--runtime` option:

```
cd /root/prox/helper-scripts/rapid/
./runrapid.py --runtime <time> # replace <time> with time per one execution in seconds
```

A few other command line options are supported, which can be listed by `-h`:

```
[root@stack2-jump rapid]# ./runrapid.py -h
usage: runrapid    [--version] [-v]
                   [--env ENVIRONMENT_NAME]
                   [--test TEST_NAME]
                   [--map MACHINE_MAP_FILE]
                   [--runtime TIME_FOR_TEST]
                   [--configonly False|True]
                   [--log DEBUG|INFO|WARNING|ERROR|CRITICAL]
                   [-h] [--help]
Command-line interface to runrapid
optional arguments:
  -v,  --version           Show program's version number and exit
  --env ENVIRONMENT_NAME   Parameters will be read from ENVIRONMENT_NAME. Default is rapid.env.
  --test TEST_NAME         Test cases will be read from TEST_NAME. Default is basicrapid.test.
  --map MACHINE_MAP_FILE   Machine mapping will be read from MACHINE_MAP_FILE. Default is machine.
map.
  --runtime                Specify time in seconds for 1 test run
 --configonly           If this option is specified, only upload all config files to the VMs, do not
run the tests
  --log                    Specify logging level for log file output, default is DEBUG
  --screenlog              Specify logging level for screen output, default is INFO
  -h, --help               Show help message and exit.
```

A typical `runrapid.py` script execution looks like Figure 5.10.



```
[root@stack2-jump rapid]# ./runrapid.py --runtime 5
Using 'rapid.env' as name for the environment
Using 'basicrapid.test' for test case definition
Using 'machine.map' for machine mapping
Runtime: 5
Connected to PROX on 192.168.0.104
Connected to PROX on 192.168.0.105
warmuptest
flowsizetest
+-------------------------------------------------------------------------------------------------------------------------------------------------------+
| UDP,    64 bytes, different number of flows by randomizing SRC & DST UDP port                                                                            |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
| Flows  |  Speed requested   | core generated   | Sent by Gen NIC| Forward by SUT |      core received       | Avg. Latency  | Max. Latency  | Packets Lost  | Loss Ratio  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
|  16384 |   0.9%  0.131 Mpps |      0.131 Mpps  |     0.131 Mpps |     0.131 Mpps | 0.1Gb/s      0.131 Mpps  |       669 us  |      4200 us  |             0 |      0.00%  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
+-------------------------------------------------------------------------------------------------------------------------------------------------------+
| UDP,   256 bytes, different number of flows by randomizing SRC & DST UDP port                                                                            |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
| Flows  |  Speed requested   | core generated   | Sent by Gen NIC| Forward by SUT |      core received       | Avg. Latency  | Max. Latency  | Packets Lost  | Loss Ratio  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
|  16384 |   1.4%  0.063 Mpps |      0.063 Mpps  |     0.063 Mpps |     0.063 Mpps | 0.1Gb/s      0.063 Mpps  |       680 us  |      4254 us  |             0 |      0.00%  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
+-------------------------------------------------------------------------------------------------------------------------------------------------------+
| UDP,   512 bytes, different number of flows by randomizing SRC & DST UDP port                                                                            |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
| Flows  |  Speed requested   | core generated   | Sent by Gen NIC| Forward by SUT |      core received       | Avg. Latency  | Max. Latency  | Packets Lost  | Loss Ratio  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
|  16384 |   2.9%  0.069 Mpps |      0.069 Mpps  |     0.069 Mpps |     0.069 Mpps | 0.3Gb/s      0.069 Mpps  |       666 us  |      4190 us  |             0 |      0.00%  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
+-------------------------------------------------------------------------------------------------------------------------------------------------------+
| UDP,  1024 bytes, different number of flows by randomizing SRC & DST UDP port                                                                            |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
| Flows  |  Speed requested   | core generated   | Sent by Gen NIC| Forward by SUT |      core received       | Avg. Latency  | Max. Latency  | Packets Lost  | Loss Ratio  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
|  16384 |   8.1%  0.096 Mpps |      0.096 Mpps  |     0.096 Mpps |     0.096 Mpps | 0.8Gb/s      0.096 Mpps  |       884 us  |      5552 us  |             0 |      0.00%  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
+-------------------------------------------------------------------------------------------------------------------------------------------------------+
| UDP,  1500 bytes, different number of flows by randomizing SRC & DST UDP port                                                                            |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
| Flows  |  Speed requested   | core generated   | Sent by Gen NIC| Forward by SUT |      core received       | Avg. Latency  | Max. Latency  | Packets Lost  | Loss Ratio  |
+--------+--------------------+------------------+----------------+----------------+--------------------------+---------------+---------------+---------------+-------------+
|  16384 | Speed 0 or close to 0 at speed: 0.07%
Waiting for child process 'PROX Testing on TestM1' to complete ...
Child process 'PROX Testing on TestM1' completed successfully
Child process 'PROX Testing on TestM2' completed successfully
[root@stack2-jump rapid]#
```

Figure 5.10 Runrapid.py Script Results

You can see that some preparation work was done before the actual test was started. First, the script read three files, `rapid.env`, `basicrapid.test`, and `machine.map`. The env file provides IP/MAC information of the gen and swap VM, and the `.test` file defines all detail behavior of the test.

1. Then, the script connects to both gen and swap VM.

2. The script starts some small amount of traffic as warmup, to test the reachability between the source and destination, and also to populate the MAC table or ARP table in devices along the path.

3. When everything is ready, the script starts the traffic and at the same time monitors the traffic's receiving rate in real time. Any packet drop rate higher than the defined threshold indicates the current traffic rate is too high to the DUT, so it will drop the rate in the next iteration. By binary search, it eventually finds the maximum throughput between the two systems within a given allowed packet loss and accuracy which are defined in the `*.test` files (for example, the `basicrapid.test` file for a simple test).

The script is highly configurable. The book's github site for this book provides a sample basicrapid.test used in our lab. You can start with it and fine tune based on your needs. For example, in section `[test2]` of the file you can change the rate of flow and packet size to define different test scenarios.

```
[test2]
test=flowsizetest
```

```
packetsizes=[64,256,512,1024,1500]
# the number of flows in the list need to be powers of 2, max 2^20
# Select from following numbers: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384,
32768, 65536, 131072, 262144, 524280, 1048576
flows=[16384, 65536]
```

Run PROX Manually

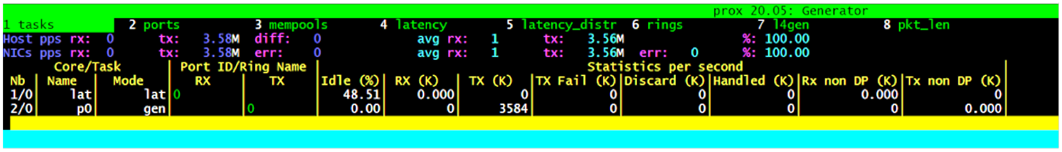
Okay, we just introduced rapid. The script supports very extensive options in the configuration files, which are beyond the scope of this book, but you should have a basic idea of how it works. Please remember that rapid and PROX and two different applications. A rapid script does all the magic and makes your life easier through the automation of PROX, and PROX is the foundation application that does the real work. In fact, PROX can run tests just fine without rapid. To launch PROX and start traffic in the gen VM's home folder (`root` in our case), run this command:

```
[root@stack2-gen ~]# /root/prox/build/prox -f /root/gen.cfg
```

PROX will parse its configuration file `/root/gen.cfg` and start to boot. From the booting messages in the screen you can learn its booting sequences:

- set up the DPDK environment (RTE EAL)
- initialize (rte) devices
- initialize mempools, port addresses, queue numbers, and rings on cores
- initialize DPDK ports
- initialize tasks
- start the test and display a ncurse based text UI

You will end up with a `ncurse` based UI like that shown in Figure 5.11.

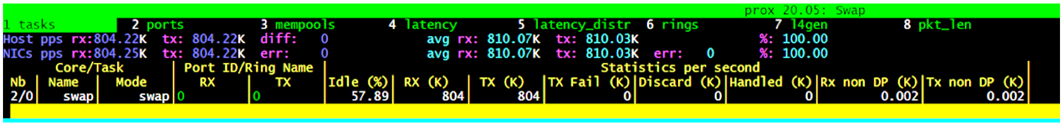


```
                                                                                       prox 20.05: Generator
1 tasks          2 ports          3 mempools       4 latency        5 latency_distr  6 rings          7 l4gen          8 pkt_len
Host pps rx:  0     tx:   3.58M diff:   0              avg rx:   1      tx:   3.56M              %: 100.00
NICs pps rx:  0     tx:   3.58M err:    0              avg rx:   1      tx:   3.56M  err:   0    %: 100.00
       Core/Task       | Port ID/Ring Name |                          Statistics per second
 Nb | Name |   Mode    |    RX    |   TX   |Idle (%)| RX (K) | TX (K) |TX Fail (K)|Discard (K)|Handled (K)|Rx non DP (K)|Tx non DP (K)|
 1/0|   lat|        lat|0         |        |   48.51|   0.000|       0|          0|          0|          0|        0.000|            0|
 2/0|    p0|        gen|          |0       |    0.00|       0|    3584|          0|          0|          0|            0|        0.000|
```

Figure 5.11 Gen Running UI

The display shows per task statistics that include estimated idleness, per second statistics for packets received, transmitted or dropped, per core cache occupancy, cycles per packet, etc. These statistics can help pinpoint bottlenecks in the system. This information can then be used to optimize the configuration. There are quite a few other features including debugging support, scripting, Open vSwitch support, and more. Refer to the PROX website for more details. For now, let's look at how the traffic flows.

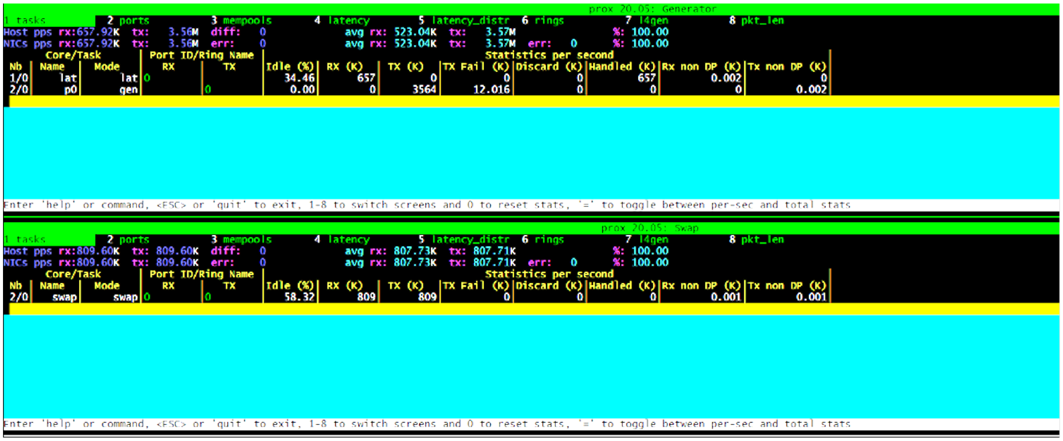
In Figure 5.11 you can only see traffic being sent, but nothing gets received yet. This is because we are now running PROX manually and we are only starting the gen side, which is the traffic sender only. You need to start the swap VM as well as a receiver, which will also loop the traffic back to the sender, so our first PROX application will see some RX statistics. Let's do that. On the compute where the swap VM is installed, execute the same prox command line, except this time pass a different configuration file named `swap.cfg`. See Figure 5.12.



```
                                                                                       prox 20.05: Swap
1 tasks          2 ports          3 mempools         4 latency          5 latency_distr  6 rings          7 l4gen           8 pkt_len
Host pps rx:804.22K  tx: 804.22K  diff:    0                avg rx: 810.07K  tx: 810.03K               %: 100.00
NICs pps rx:804.25K  tx: 804.22K  err:     0                avg rx: 810.07K  tx: 810.03K  err:    0    %: 100.00
       Core/Task             |  Port ID/Ring Name   |                              Statistics per second
 Nb  |  Name   |     Mode    |    RX    |     TX    |Idle (%)| RX (K)  |  TX (K)  |TX Fail (K)|Discard (K)|Handled (K)|Rx non DP (K)|Tx non DP (K)
 2/0 |    swap |        swap |0         |0          |   57.89|      804|       804|          0|          0|          0|        0.002|        0.002
```

Figure 5.12 Swap Running UI

You will end up with a similar ncurse-based text UI, after a similar booting process of the sender. Once the swap end of PROX is up and running, you will immediately see both RX and TX counters (Figure 5.13) keep updating on both sides of the traffic.



```
                                                                                       prox 20.05: Generator
1 tasks          2 ports          3 mempools         4 latency          5 latency_distr  6 rings          7 l4gen           8 pkt_len
Host pps rx:657.92K  tx:    3.56M  diff:    0                avg rx: 523.04K  tx:    3.57M               %: 100.00
NICs pps rx:657.92K  tx:    3.56M  err:     0                avg rx: 523.04K  tx:    3.57M  err:    0    %: 100.00
       Core/Task             |  Port ID/Ring Name   |                              Statistics per second
 Nb  |  Name   |     Mode    |    RX    |     TX    |Idle (%)| RX (K)  |  TX (K)  |TX Fail (K)|Discard (K)|Handled (K)|Rx non DP (K)|Tx non DP (K)
 1/0 |     lat |         lat |0         |           |   34.46|      657|         0|          0|          0|        657|        0.002|            0
 2/0 |      p0 |         gen |          |0          |    0.00|        0|      3564|     12.016|          0|          0|            0|        0.002

Enter 'help' or command, <ESC> or 'quit' to exit, 1-8 to switch screens and 0 to reset stats, '=' to toggle between per-sec and total stats
```

```
                                                                                       prox 20.05: Swap
1 tasks          2 ports          3 mempools         4 latency          5 latency_distr  6 rings          7 l4gen           8 pkt_len
Host pps rx:809.60K  tx: 809.60K  diff:    0                avg rx: 807.73K  tx: 807.71K               %: 100.00
NICs pps rx:809.60K  tx: 809.60K  err:     0                avg rx: 807.73K  tx: 807.71K  err:    0    %: 100.00
       Core/Task             |  Port ID/Ring Name   |                              Statistics per second
 Nb  |  Name   |     Mode    |    RX    |     TX    |Idle (%)| RX (K)  |  TX (K)  |TX Fail (K)|Discard (K)|Handled (K)|Rx non DP (K)|Tx non DP (K)
 2/0 |    swap |        swap |0         |0          |   58.32|      809|       809|          0|          0|          0|        0.001|        0.001

Enter 'help' or command, <ESC> or 'quit' to exit, 1-8 to switch screens and 0 to reset stats, '=' to toggle between per-sec and total stats
```

Figure 5.13 Gen and Swap UI

That concludes our discussion of PROX and rapid as testing tools. We'll use these tools extensively in the rest of this book to generate different kinds of traffic in each test. With the traffic running, you can dig deeper in order to understand the rules about how vRouter works. Now let's introduce some of the commonly used tools that are designed for, or especially useful for, verifications in the DPDK vRouter environment.

Chapter 6

Contrail DPDK vRouter Toolbox and Case Study

DPDK vRouter Tool Box

You've read a lot of details about DPDK and Contrail DPDK vRouter implementations. You should understand that its main benefit is boosting performance, but it does come with its own pros and cons. One problem commonly raised is the lack of tools during the troubleshooting process, especially in the case of traffic loss problems. Within the traditional Linux world, there are tons of well-known tools to trace the packet, from displaying packet statistics in and out of NIC, to showing drop counters, to performing packet capture for deeper level packet decoding. Examples of these tools are ifconfig, ip, bmon, tcpdump, tshark, etc. With DPDK, however, none of the traditional tools can be used directly, and the reason is obvious: whichever interface bound to DPDK becomes invisible to the Linux stack, hence they are also hidden from the perspective of these tools. In production, you need some new tools developed to fill this gap, so that you can narrow the packet loss related issues when the outage is ongoing. Fortunately, today's Contrail DPDK vRouter is equipped with quite a few such tools. In this section we'll look at some of them.

Contrail-tools Docker: vRouter Tool Box

Contrail-tools is a Docker container located in the compute node, where all of the vRouter tools and utilities are available. Apparently, from the user perspective, this is more convenient than distributing tools in multiple containers. This design was introduced a few releases before Contrail networking R2008. As more and more existing tools migrated into it and new tools were added, this container became a centralized tool box for whenever you want to check any running states of the vRouter data plane. Let's first take a look at how to open this box.

To enter the container, just run the `contrail-tools` script (same name as of the Docker) in a compute node:

```
[root@a7s3 ~]# contrail-tools
Unable to find image 'svl-artifactory.juniper.net/contrail-nightly/contrail-tools:2008.108' locally
2008.108: Pulling from contrail-nightly/contrail-tools
f34b00c7da20: Already exists
b3779b5a313a: Already exists
4b95f42cde64: Already exists
8b329f8ee1e6: Already exists
2986115b3d27: Already exists
10c5940c4895: Already exists
dec794e181cd: Already exists
226c056c5788: Already exists
d391962e0038: Pull complete
Digest: sha256:2d68d8cd010ba76c265c3b7458fcf12c459d46ec71357b45118dfc4610f40338
Status: Downloaded newer image for svl-artifactory.juniper.net/contrail-nightly/contrail-
tools:2008.108
(contrail-tools)[root@a7s3 /]$
```

Now you are inside of the container. From here you can test all the old vRouter tools you may have been familiar with, for example, printing the packet dropping statistics:

```
(contrail-tools)[root@a7s3 /]$ dropstats | grep -iEv  0$|^$
Flow Action Drop              1792
Flow Queue Limit Exceeded     305
Invalid NH                    12
No L2 Route                   1
```

We use grep to remove all counters with a zero value. When you are done, just exit the Docker and it will be killed:

```
(contrail-tools)[root@a7s3 /]$ exit
exit
[root@a7s3 ~]#
```

You can also pass the tool command as parameters to the script, execute the command, get its output, and exit the Docker all with one go:

```
[root@a7s3 ~]# contrail-tools dropstats | grep -iE route
No L2 Route                   68129939
[root@a7s3 ~]#
```

As of this writing, there are nearly twenty tools available in this container. Let's take a look at what's in the package.

First, in the container locate the package name:

```
[root@a7s3 ~]# contrail-tools
lcontrail-tools)[root@a7s3 /]$ rpm -qa | grep contrail-tool
contrail-tools-2008-108.el7.x86_64
```

Then, based on the package name, you can list all available tools in it:

```
(contrail-tools)[root@a7s3 /]$ repoquery -l contrail-tools-2008-108.el7.x86_64 | grep bin
/usr/bin/dpdkinfo
/usr/bin/dpdkvifstats.py
/usr/bin/dropstats
/usr/bin/flow
/usr/bin/mirror
/usr/bin/mpls
/usr/bin/nh
/usr/bin/pkt_droplog.py
/usr/bin/qosmap
/usr/bin/rt
/usr/bin/sandump
/usr/bin/vif
/usr/bin/vifdump
/usr/bin/vrfstats
/usr/bin/vrftable
/usr/bin/vrinfo
/usr/bin/vrmemstats
/usr/bin/vrouter
/usr/bin/vxlan
```

In previous chapters you've read about the `dpdk_nic_bind.py` script, which is a tool to bind a specific driver for a NIC. In the rest of this section, we'll introduce some more tools that are especially useful in the DPDK environment.

VIF Command and Scripts

The first command from our Contrail DPDK tool box is the `vif` command. Before talking about it, let's see how to list all interfaces in the compute running DPDK vRouter. Let's first try the Linux ip or ifconfig command in our DPDK compute running PROX gen VM:

```
[root@a7s3 ~]# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 0c:c4:7a:4c:16:c2 brd ff:ff:ff:ff:ff:ff
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 0c:c4:7a:4c:16:c3 brd ff:ff:ff:ff:ff:ff
8: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:56:4f:cc:6e brd ff:ff:ff:ff:ff:ff
25: vhost0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UNKNOWN mode DEFAULT group default qlen 1000
    link/ether 90:e2:ba:c3:af:20 brd ff:ff:ff:ff:ff:ff
```

You can see some interfaces were output:

- the loopback interface (lo)
- management interface (eno1)
- vhost0 interface
- docker interface (docker0)
- physical NIC which is not in use (eno2)

However, some of the most important interfaces are not shown at all:

- The physical fabric interface: the bond interface in our setup
- The VM virtual interfaces: the tapxxx interfaces

If you compare this with what you'd see with the same IP command in a kernel mode vRouter compute without DPDK, there's a big difference:

```
[root@a7s5-kiran ~]# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eno1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 0c:c4:7a:47:d7:b4 brd ff:ff:ff:ff:ff:ff
3: eno2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether 0c:c4:7a:47:d7:b5 brd ff:ff:ff:ff:ff:ff
4: enp2s0f0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_
UP> mtu 1500 qdisc mq master bond0 state UP mode DEFAULT group default qlen 1000
    link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff
5: enp2s0f1: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_
UP> mtu 1500 qdisc mq master bond0 state UP mode DEFAULT group default qlen 1000
    link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff
6: bond0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_
UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff
12: docker0: <NO-
CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:d6:c6:2c:12 brd ff:ff:ff:ff:ff:ff
41: pkt1: <UP,LOWER_UP> mtu 65535 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/void c2:6e:97:ef:cd:b2 brd 00:00:00:00:00:00
42: pkt3: <UP,LOWER_UP> mtu 65535 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/void 8e:44:4e:2e:28:0c brd 00:00:00:00:00:00
43: pkt2: <UP,LOWER_UP> mtu 65535 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/void a6:2a:01:7c:db:65 brd 00:00:00:00:00:00
44: vhost0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UNKNOWN mode DEFAULT group default qlen 1000
    link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff
45: bond0.101@bond0: <BROADCAST,MULTICAST,UP,LOWER_
UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default qlen 1000
    link/ether 00:1b:21:bb:f9:46 brd ff:ff:ff:ff:ff:ff
46: pkt0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_
fast state UNKNOWN mode DEFAULT group default qlen 1000
    link/ether 5e:a0:f8:77:25:97 brd ff:ff:ff:ff:ff:ff
49: tap0160123b-14: <BROADCAST,MULTICAST,UP,LOWER_
UP> mtu 1500 qdisc mq state UP mode DEFAULT group default qlen 1000
    link/ether fe:01:60:12:3b:14 brd ff:ff:ff:ff:ff:ff
```

Here except for the lo and management interface, and whatever we saw from the DPDK compute, you can also see these other important interfaces:

- bond interface and its sub-interface: bond0, bond0.101
- bond interface's member interfaces: enp2s0f0, enp2s0f1
- VM tap interface: tap0160123b-14
- pkt0 interface

NOTE The pkt1, pkt2, and pkt3 interfaces are created by vRouter but not used in DPDK setup.

The reason you can see these differences, as we've mentioned many times throughout this book, is that when DPDK is in charge of the NIC card the Linux kernel is mostly bypassed. The NIC card's feature and functions are exposed by another special driver directly connected to the user space PMD driver running in the DPDK layer, so the traditional applications that rely on the interfaces sitting in the Linux kernel to do their job are no longer useful.

We'll talk more about this later. For now, let's look at the `vif` command with `-l|--list` and `--get` option. The `vif --list` lists all interfaces located in the vRouter and `--get` just retrieves one of them. Here is the capture from the same DPDK compute:

```
[root@a7s3 ~]# contrail-tools vif --get 3
Vrouter Interface Table

......
vif0/3      PMD: tap41a9ab05-64 NH: 32
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104
            Vrf:3 Mcast Vrf:3 Flags:PL3L2DMonEr QOS:-1 Ref:12
            RX queue  packets:2306654691 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:2306869103  bytes:285898139558 errors:0
            TX packets:47613036  bytes:5739655392 errors:0
            ISID: 0 Bmac: 02:41:a9:ab:05:64

[root@a7s3 ~]# contrail-tools vif -l
Vrouter Interface Table
......

vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
            Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18
            RX device packets:106218495224  bytes:12108991404264 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:02:00.0  Status: UP  Driver: net_ixgbe
            Slave Interface(1): 0000:02:00.1  Status: UP  Driver: net_ixgbe
            Vlan Id: 101  VLAN fwd Interface: vfw
            RX packets:53109240518  bytes:5842056828972 errors:0
            TX packets:53459418469  bytes:5880886194306 errors:0
            Drops:291
            TX device packets:106919210258  bytes:12189494593618 errors:0
```

```
vif0/1      PMD: vhost0 NH: 5
            Type:Host HWaddr:90:e2:ba:c3:af:20 IPaddr:8.0.0.4
            Vrf:0 Mcast Vrf:65535 Flags:L3DEr QOS:-1 Ref:13
            RX device packets:436036  bytes:400358720 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:436036  bytes:400358720 errors:0
            TX packets:447092  bytes:88525732 errors:0
            Drops:3
            TX device packets:447092  bytes:88518904 errors:0

vif0/2      Socket: unix
            Type:Agent HWaddr:00:00:5e:00:01:00 IPaddr:0.0.0.0
            Vrf:65535 Mcast Vrf:65535 Flags:L3Er QOS:-1 Ref:3
            RX port   packets:71548 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            RX packets:71548  bytes:6153128 errors:0
            TX packets:14936  bytes:1359697 errors:0
            Drops:0

vif0/3      PMD: tap41a9ab05-64 NH: 38
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104
            Vrf:2 Mcast Vrf:2 Flags:L3L2DEr QOS:-1 Ref:12
            RX queue  packets:17708866065 errors:3874701360
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 3874691664 9696
            RX packets:17708865121  bytes:1062531327800 errors:0
            TX packets:17563478684  bytes:1053808124972 errors:0
            ISID: 0 Bmac: 02:41:a9:ab:05:64
            Drops:3874701393

vif0/4      PMD: tapd2d7bb67-c1 NH: 35
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.0.104
            Vrf:3 Mcast Vrf:3 Flags:PL3L2DEr QOS:-1 Ref:12
            RX queue  packets:3060 errors:205
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 205 0
            RX packets:5478  bytes:528770 errors:0
            TX packets:5402  bytes:423320 errors:0
            Drops:445
```

Here, the vRouter interfaces are:

- vif0/0: this connects to the bond interface
- vif0/1: this connects to `vhost0`, the interface in Linux kernel
- vif0/2: this connects to the `pkt0` interface toward vRouter agent
- vif0/3: this is the vRouter interface connecting the data interface of our PROX VM: `tap41a9ab05-64`
- vif0/4: this is the vRouter interface connecting the control and management interface of our PROX VM: `tapd2d7bb67-c1`

Now you should understand the importance of the `vif` command, especially in the DPDK vRouter. It shows interfaces from vRouter's perspective, and reveals the one-to-one connection mapping between vRouter and fabric or the VM tap interface. The latter would otherwise be invisible.

Besides that, it also displays key information. The `vrf` numbers and packet counters are the most commonly used data points, but among various counters we usually focus on the `RX/TX packets/bytes` counters, which display data received or sent in packets or bytes. Depending on your environment, sometimes you may also see non-zero numbers in the RX/TX queue `packets/errors` counter that gives inter-lcore packet statistics. This usually happens when two lcores are involved in the packet forwarding path. Let's use this command intensively in the rest of this chapter and analyze the lcores to understand some of the important vRouter working mechanisms.

The `vif` tool also supports some other options such as `--help` to display a brief list of all currently supported options:

```
[root@a7s3 ~]# contrail-tools vif --help
Usage: vif [--create <intf_name> --mac <mac>]
           [--add <intf_name> --mac <mac> --vrf <vrf>
                --type [vhost|agent|physical|virtual|monitoring]
                --transport [eth|pmd|virtual|socket]
                --xconnect <physical interface name>
                --policy, --vhost-phys, --dhcp-enable]
                --vif <vif ID> --id <intf_id> --pmd --pci]
           [--delete <intf_id>|<intf_name>]
           [--get <intf_id>][--kernel][--core <core number>][--rate] [--get-drop-stats]
           [--set <intf_id> --vlan <vlan_id> --vrf <vrf_id>]
           [--list][--core <core number>][--rate]
           [--sock-dir <sock dir>]
           [--clear][--id <intf_id>][--core <core_number>]
           [--help]
```

We won't cover every option and its usage and usually you don't need to know any except `--get` and `-l|--list`. But there is one more, (`--add`), which we'll talk about shortly. The `--clear` option will reset all counters, and this is handy to set a quick clean baseline for later observations, which we'll give an example of later.

MORE? For other options, refer to the Juniper documentation at: https://www.juniper.net/documentation/en_US/contrail20/topics/task/configuration/vrouter-cli-utilities-vnc.html.

Now let's look at two useful scripts developed based on the `vif` command: `dpdkvifstats.py` and `vifdump`.

dpdkvifstats.py script

We've seen that the `vif` command displays all interfaces and their traffic statistics (RX/TX packets/bytes/errors, RX queue packets/errors, etc.) in the form of a list. During testing or troubleshooting, you can collect this data to evaluate the vRouter's forwarding performance, its running status, whether it is losing packets or not, etc. In production, you always need to examine the traffic passing through a compute. It's the same thing in a lab, once you start traffic from the PROX or any other traffic generator, the first thing you want to check is the traffic rate on the interfaces.

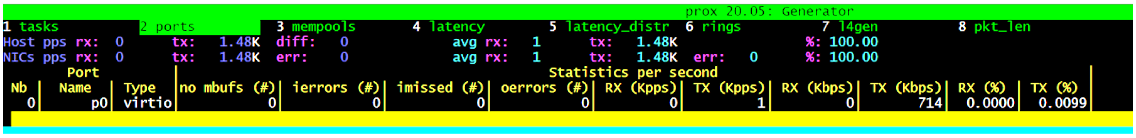
In fact, there are at least two common tasks in practice:

- monitor the traffic forwarding rate (instead of only number of packets)
- compare statistics between different vif interfaces

Starting from R2008, a Python script named `dpdkvifstat.py` is provided, which collects the statistics from `vif` output, calculates the changing rate of all counters in `pps` (packet per second) and bps (bit per second), based on both per-lcore and total statistics. It then displays the result in a table format. This makes the output prettier and the comparison across vif interfaces much easier to read.

In fact the `vif` command also provides `--list --rate` options to display traffic rates. However, it is lacking itemized per-lcore statistics and the display is not easy to collect in a file.

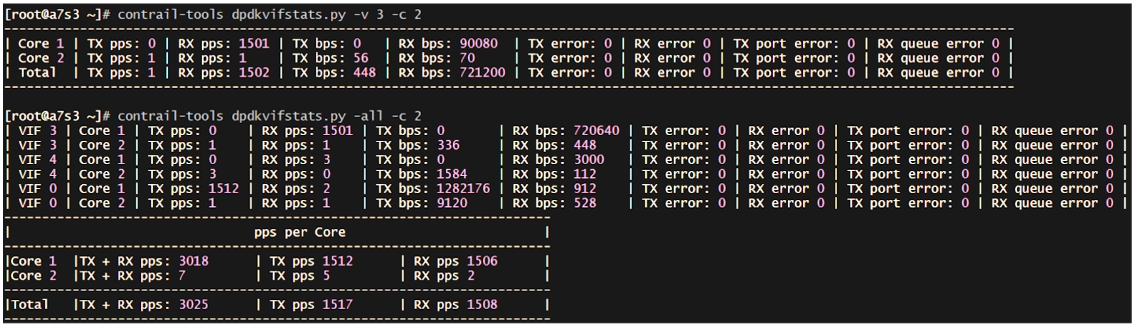
To demonstrate how the script works (see Figure 6.1), in our testbed we have configured PROX to send traffic at a constant speed of 125000 bytes per second (Bps) with minimum packet size of 60 bytes. That calculates to about 1.48K packet per second.



```
                                                                                   prox 20.05: Generator
1 tasks          2 ports        3 mempools      4 latency       5 latency_distr  6 rings         7 l4gen         8 pkt_len
Host pps rx:  0      tx:   1.48K diff:   0            avg rx:   1     tx:   1.48K                 %: 100.00
NICs pps rx:  0      tx:   1.48K err:    0            avg rx:   1     tx:   1.48K  err:   0       %: 100.00
        Port                                                          Statistics per second
 Nb |   Name  |  Type  |no mbufs (#)| ierrors (#)| imissed (#)| oerrors (#)| RX (Kpps)| TX (Kpps)| RX (Kbps)| TX (Kbps)| RX (%)  | TX (%)  |
  0 |      p0 | virtio |           0|           0|           0|           0|         0|         1|         0|       714| 0.0000  | 0.0099  |
```

Figure 6.1 PROX Gen Sending Traffic

Let’s take a look at the `dpdkvifstats.py` script output in Figure 6.2.



```
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 3 -c 2
----------------------------------------------------------------------------------------------------------------------------
| Core 1 | TX pps: 0 | RX pps: 1501 | TX bps: 0   | RX bps: 90080  | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
| Core 2 | TX pps: 1 | RX pps: 1    | TX bps: 56  | RX bps: 70     | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
| Total  | TX pps: 1 | RX pps: 1502 | TX bps: 448 | RX bps: 721200 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
----------------------------------------------------------------------------------------------------------------------------

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 2
| VIF 3 | Core 1 | TX pps: 0    | RX pps: 1501 | TX bps: 0       | RX bps: 720640 | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
| VIF 3 | Core 2 | TX pps: 1    | RX pps: 1    | TX bps: 336     | RX bps: 448    | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
| VIF 4 | Core 1 | TX pps: 0    | RX pps: 3    | TX bps: 0       | RX bps: 3000   | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
| VIF 4 | Core 2 | TX pps: 3    | RX pps: 0    | TX bps: 1584    | RX bps: 112    | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
| VIF 0 | Core 1 | TX pps: 1512 | RX pps: 2    | TX bps: 1282176 | RX bps: 912    | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
| VIF 0 | Core 2 | TX pps: 1    | RX pps: 1    | TX bps: 9120    | RX bps: 528    | TX error: 0 | RX error 0 | TX port error: 0 | RX queue error 0 |
--------------------------------------------------------------------------
|                               pps per Core                             |
--------------------------------------------------------------------------
|Core 1  |TX + RX pps: 3018        | TX pps 1512        | RX pps 1506      |
|Core 2  |TX + RX pps: 7           | TX pps 5           | RX pps 2         |
--------------------------------------------------------------------------
|Total   |TX + RX pps: 3025        | TX pps 1517        | RX pps 1508      |
--------------------------------------------------------------------------
```

Figure 6.2 Dpdkvifstats.py

We typically run the script two times. First, to show traffic rate for vif0/3 (`-v`), then to show traffic rate for all (`-a`) vif interfaces for comparison purposes. In both executions, per-lcore statistics of a specific interface are given separately. With the `-v` option, the total value of the interface is also given, which is in addition of counters from all cores. This gives a per-interface statistic. With `-a`, the script also calculates RX/TX/RX+TX traffic rate for each lcore across all interfaces in the end. This gives the overall lcore forwarding load in the DPDK vRouter.

To understand the output, let's first review the DPDK vRouter CPU cores allocation. In Chapter 3, you learned about DPDK vRouter architectures and you learned how the packet processing works. Basically: vRouter creates the same number of lcores and DPDK queues as the number of CPUs allocated to it.

For testing purposes, in this compute we've allocated two CPU cores to vRouter DPDK forwarding lcores. CPU allocation to DPDK vRouter forwarding lcores is configurable via vRouter configuration files. Refer to Chapter 4 for CPU allocation details. For each vRouter interface, two DPDK queues are created, each served by a forwarding lcore in DPDK process. That is why in the output for each vif interface there are two lines statistics, for Core 1 and Core 2, respectively.

This capture shows vRouter interface `vif 0/3` processed `1501` pps traffic in the first forwarding lcore, that is `720640` bits per second for 60 bytes packet size. These show the majority of the traffic forwarded out of fabric interface `vif 0/0`, with a similar rate of `1512` pps. These overlay packets received from the VM will be tunneled in extra underlay encapsulations, MPLSoUDP in this case, so the `vif 0/0` bps number (`1282176`) will be a little bit bigger comparing with the number on the VM interface.

This script conveniently gives a straightforward overview about the current traffic profile from vRouter's perspective. To compare with the original `vif` output which the script is based on, let's check what the raw data looks like without the dpdkvif-stats.py script:

```
[root@a7s3 ~]# vif --clear; sleep 1; vif --get 3 --core 10; vif --get 3 --core 11

Vif stats cleared successfully on all cores for all interfaces
......

vif0/3      PMD: tap41a9ab05-64 NH: 34
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104
            Vrf:2 Mcast Vrf:2 Flags:PL3L2DEr QOS:-1 Ref:12
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Core 10 RX packets:1488  bytes:89280 errors:0
            Core 10 TX packets:0  bytes:0 errors:0
            ISID: 0 Bmac: 02:41:a9:ab:05:64
            Drops:131

......
vif0/3      PMD: tap41a9ab05-64 NH: 34
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104
            Vrf:2 Mcast Vrf:2 Flags:PL3L2DEr QOS:-1 Ref:12
            Core 11 RX queue  packets:1496 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Core 11 RX packets:0  bytes:0 errors:0
            Core 11 TX packets:0  bytes:0 errors:0
            ISID: 0 Bmac: 02:41:a9:ab:05:64
            Drops:131
```

We captured the interface data, waited for one second, and then captured it again. After that we can calculate the differences of all the counters between the two captures to get the increasing rate of each counter:

- pps - packets per second: 1488 pps
- Bps - bytes per second: 1488 * 60 = 89280 Bps
- bps - bit per second: 89280 * 8 = 714240 bps

These numbers are consistent with what is seen in the `dpdkvifstats.py` script. But to monitor multiple vif interfaces you have to repeat these steps multiple times. Compare this with having a handy script doing everything for you!

As mentioned, `dpdkvifstats.py` script is useful to quickly retrieve a snapshot of current traffic profile. When everything goes well it is fine. In the case of traffic loss, you often need to first capture the packets themselves, then from one of the packet captures you can decode the payload and take a deeper look to analyze the issue. Now, you may say, oh you mean `tcpdump`! Well, yes and no. Please remember the fact that we are in a setup where the NIC card is invisible to most of the Linux applications, including `tcpdump`! Next let's go over another DPDK vRouter packet capture script: `vifdump`.

vifdump Script

In many Linux machines, `tcpdump` comes with the OS as part of a standard package. With it you can capture whatever packets sensed by a NIC, which can be either physical NIC or virtual NIC, like a `tuntap` interface. Both NICs are visible to the kernel. In DPDK environments, the difficulty of an interface not being visible to the kernel makes `tcpdump` unworkable, unless you just want it to read packets from a file.

Fortunately, we now know that each interface related to the vRouter data plane connects to a unique vRouter interface (`vif`). We can make use of this fact and create an alternative. `vifdump` is a shell script and when invoked it uses the --add option of the `vif` command to create a monitoring tun interface in the Linux kernel, and internally the vRouter will clone all data passing through the monitored vif interface to this kernel interface. `vifdump` will then start up the `tcpdump` program to capture the packets from the monitoring tun interface. From a user's perspective, the script works the same way as with `tcpdump`. Here are two captures on vif0/3 toward VM, which is our PROX gen, and on `vif0/0` toward the fabric interface:

```
[root@a7s3 ~]# contrail-tools vifdump -i 3 -n -c 3
vif0/3      PMD: tap41a9ab05-64 NH: 32
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mon3, link-type EN10MB (Ethernet), capture size 262144 bytes
13:12:31.286528 IP 192.168.1.104.filenet-cm > 192.168.1.105.filenet-nch: UDP, length 82
13:12:31.286532 IP 192.168.1.104.filenet-rmi > 192.168.1.105.filenet-pch: UDP, length 82
13:12:31.286540 IP 192.168.1.104.filenet-rpc > 192.168.1.105.filenet-pa: UDP, length 82
3 packets captured
401 packets received by filter
271 packets dropped by kernel
vifdump: deleting vif 4348...
```

```
[root@a7s3 ~]# contrail-tools vifdump -i 0 -n -c 3
vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on mon0, link-type EN10MB (Ethernet), capture size 262144 bytes
13:12:23.796516 IP 8.0.0.4.55184 > 8.0.0.2.4789: VXLAN, flags [I] (0x08), vni 8
IP 192.168.1.104.filenet-pa > 192.168.1.105.filenet-nch: UDP, length 82
13:12:23.796522 IP 8.0.0.4.54530 > 8.0.0.2.4789: VXLAN, flags [I] (0x08), vni 8
IP 192.168.1.104.filenet-rmi > 192.168.1.105.filenet-pa: UDP, length 82
13:12:23.796531 IP 8.0.0.4.63363 > 8.0.0.2.4789: VXLAN, flags [I] (0x08), vni 8
IP 192.168.1.104.filenet-nch > 192.168.1.105.filenet-pch: UDP, length 82
3 packets captured
334 packets received by filter
271 packets dropped by kernel
vifdump: deleting vif 4351...
[root@a7s3 ~]#
```

The shell script also uses UNIX trap to monitor signals and deletes the monitoring interface when the signals appear. The most used signal is `SIGINT` triggered by the keyboard's `ctrl-c` to stop the capture. That is why we see the `vifdump: deleting vif 4351…` message at the end of each capture.

`dpdkvifstats.py` and `vifdump` are two scripts developed based on the `vif` command. With these tools we can collect general packet RX/TX counters and packet contents. In the next section, we'll take a look at another powerful debug tool that is useful in the DPDK environment: `dpdkinfo`.

dpdkinfo Command

We've talked about `vif` and `dpdkvifstats.py` tools. Now let's introduce a relatively new tool that can be used to investigate lower level details of DPDK interfaces. It's `dpdkinfo` and it was introduced in Contrail 2008. With it, Contrail operators can collect more information about DPDK vRouter fabric interface internal status, connectivity (physical NIC bond), DPDK library information, and some other statistics.

Let's first run the tool with -h to get a brief menu of it:

```
(contrail-tools)[root@a7s3 /]$ dpdkinfo -h
Usage: dpdkinfo
       --help
       --version|-v                  Show DPDK Version
       --bond|-b                     Show Master/Slave bond information
       --lacp|-l     <all/conf>      Show LACP information from DPDK
       --mempool|-m  <all/<mempool-name>>  Show Mempool information
       --stats|-n    <eth>           Show Stats information
       --xstats|-x   <=all/=0(Master)/=1(Slave(0))/=2(Slave(1))>
                                     Show Extended Stats information
       --lcore|-c                    Show Lcore information
       --app|-a                      Show App information
    Optional: --buffsz      <value>  Send output buffer size (less than 1000Mb)
```

From this help information we can see it provides information about DPDK interface in multiple areas. In this rest of this section, let’s take a look at some of the most useful options:

- --version|-v
- --bond|-b
- --lacp|-l
- --stats|-n
- --xstats|-x
- --lcore|-c

There are some other options like `--app|-a, --mempool|-m` that we won’t introduce in this book, and the list of supported functions may grow in each future release. But you will get the basic idea of its usage and you can refer to the official documents for up-to-date information.

version

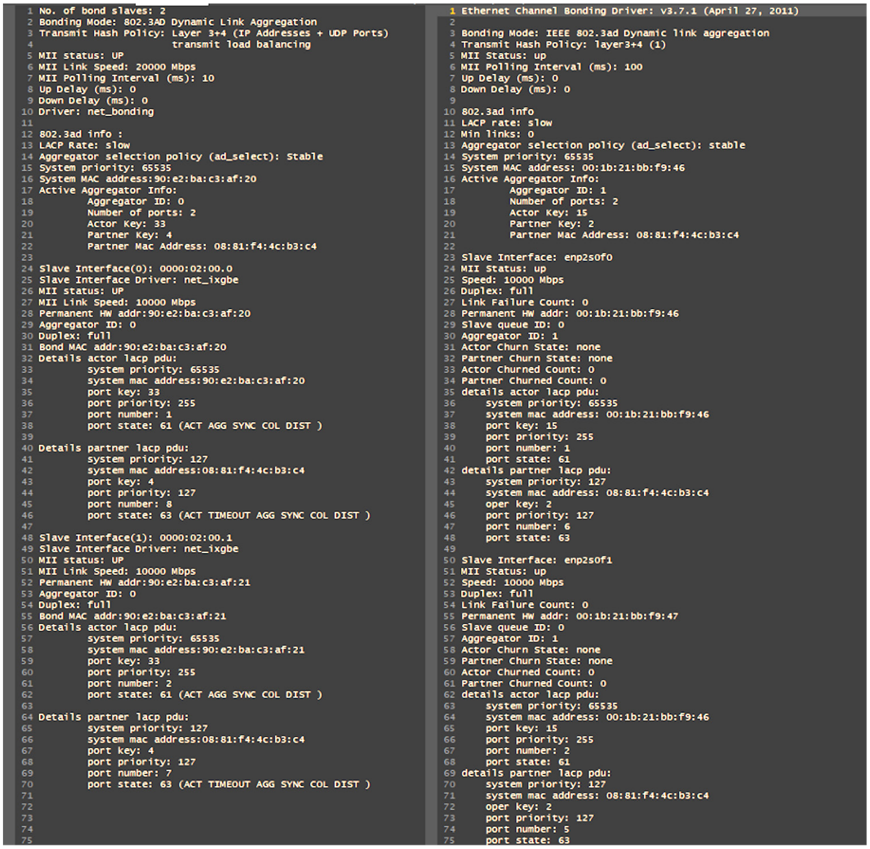
The `-v` or `--version` option reports the basic version information of DPDK release in use:

```
(contrail-tools)[root@a7s3 /]$ dpdkinfo -v
DPDK Version: DPDK 19.11.0
vRouter version: {build-info: [{build-time: 2020-09-04 10:38:22.330666, build-
hostname: 6fb64a1f86b9, build-user: root, build-version: 2004}]}
```

bond and LACP Status

The `-b` or `--bond` option displays detailed information about the bond interface managed by DPDK. The output is organized in a similar form as what you would see for the bond status managed by Linux kernel. Compare the output in Figure 6.3 with `cat /proc/net/bonding/bond0` output from a compute running kernel mode vRouter.

Basically, you can have same information as a Linux kernel bond0, such as bonding mode, transmit hash policy, system MAC and aggregator information, etc. In this example the current bonding mode is `802.3AD dynamic link aggregation`, indicating the LACP protocol is configured between compute and peer devices (in our environment it’s a TOR switch). The `Transmit Hash Policy` shows `Layer 3+4 (IP Addresses + UDP Ports) transmit load balancing`, which the bond allows for traffic to a particular network peer to span multiple clients for load balancing purpose. This is achieved by calculating a hash value for each packet from the IP addresses and UDP ports in the outer header of the packet, and then distributing the packet based on the hash value.



```
No. of bond slaves: 2
Bonding Mode: 802.3AD Dynamic Link Aggregation
Transmit Hash Policy: Layer 3+4 (IP Addresses + UDP Ports)
                      transmit load balancing
MII status: UP
MII Link Speed: 20000 Mbps
MII Polling Interval (ms): 10
Up Delay (ms): 0
Down Delay (ms): 0
Driver: net_bonding

802.3ad info :
LACP Rate: slow
Aggregator selection policy (ad_select): Stable
System priority: 65535
System MAC address:90:e2:ba:c3:af:20
Active Aggregator Info:
        Aggregator ID: 0
        Number of ports: 2
        Actor Key: 33
        Partner Key: 4
        Partner Mac Address: 08:81:f4:4c:b3:c4

Slave Interface(0): 0000:02:00.0
Slave Interface Driver: net_ixgbe
MII status: UP
MII Link Speed: 10000 Mbps
Permanent HW addr:90:e2:ba:c3:af:20
Aggregator ID: 0
Duplex: full
Bond MAC addr:90:e2:ba:c3:af:20
Details actor lacp pdu:
        system priority: 65535
        system mac address:90:e2:ba:c3:af:20
        port key: 33
        port priority: 255
        port number: 1
        port state: 61 (ACT AGG SYNC COL DIST )

Details partner lacp pdu:
        system priority: 127
        system mac address:08:81:f4:4c:b3:c4
        port key: 4
        port priority: 127
        port number: 8
        port state: 63 (ACT TIMEOUT AGG SYNC COL DIST )

Slave Interface(1): 0000:02:00.1
Slave Interface Driver: net_ixgbe
MII status: UP
MII Link Speed: 10000 Mbps
Permanent HW addr:90:e2:ba:c3:af:21
Aggregator ID: 0
Duplex: full
Bond MAC addr:90:e2:ba:c3:af:21
Details actor lacp pdu:
        system priority: 65535
        system mac address:90:e2:ba:c3:af:21
        port key: 33
        port priority: 255
        port number: 2
        port state: 61 (ACT AGG SYNC COL DIST )

Details partner lacp pdu:
        system priority: 127
        system mac address:08:81:f4:4c:b3:c4
        port key: 4
        port priority: 127
        port number: 7
        port state: 63 (ACT TIMEOUT AGG SYNC COL DIST )
```

```
Ethernet Channel Bonding Driver: v3.7.1 (April 27, 2011)

Bonding Mode: IEEE 802.3ad Dynamic link aggregation
Transmit Hash Policy: layer3+4 (1)
MII Status: up
MII Polling Interval (ms): 100
Up Delay (ms): 0
Down Delay (ms): 0

802.3ad info
LACP rate: slow
Min links: 0
Aggregator selection policy (ad_select): stable
System priority: 65535
System MAC address: 00:1b:21:bb:f9:46
Active Aggregator Info:
        Aggregator ID: 1
        Number of ports: 2
        Actor Key: 15
        Partner Key: 2
        Partner Mac Address: 08:81:f4:4c:b3:c4

Slave Interface: enp2s0f0
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 00:1b:21:bb:f9:46
Slave queue ID: 0
Aggregator ID: 1
Actor Churn State: none
Partner Churn State: none
Actor Churned Count: 0
Partner Churned Count: 0
details actor lacp pdu:
    system priority: 65535
    system mac address: 00:1b:21:bb:f9:46
    port key: 15
    port priority: 255
    port number: 1
    port state: 61
details partner lacp pdu:
    system priority: 127
    system mac address: 08:81:f4:4c:b3:c4
    oper key: 2
    port priority: 127
    port number: 6
    port state: 63

Slave Interface: enp2s0f1
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 00:1b:21:bb:f9:47
Slave queue ID: 0
Aggregator ID: 1
Actor Churn State: none
Partner Churn State: none
Actor Churned Count: 0
Partner Churned Count: 0
details actor lacp pdu:
    system priority: 65535
    system mac address: 00:1b:21:bb:f9:46
    port key: 15
    port priority: 255
    port number: 2
    port state: 61
details partner lacp pdu:
    system priority: 127
    system mac address: 08:81:f4:4c:b3:c4
    oper key: 2
    port priority: 127
    port number: 5
    port state: 63
```

Figure 6.3 *DPDK Info -b vs. cat /proc/net/bonding/bond0*

The command output also displays each member link's information, its current driver, MAC address, up/down status, etc.

Since LACP is running, LACP parameters for each member link are displayed. Another way to show this information is with `-l|--lacp` option:

```
[root@a7s3 ~]# contrail-tools dpdkinfo -l all
LACP Rate: slow

Fast periodic (ms): 900
Slow periodic (ms): 29000
Short timeout (ms): 3000
Long timeout (ms): 90000
Aggregate wait timeout (ms): 2000
Tx period (ms): 500
Update timeout (ms): 100
Rx marker period (ms): 2000
```

```
Slave Interface(0): 0000:02:00.0
Details actor lacp pdu:
        port state: 61 (ACT AGG SYNC COL DIST )

Details partner lacp pdu:
        port state: 63 (ACT TIMEOUT AGG SYNC COL DIST )

Slave Interface(1): 0000:02:00.1
Details actor lacp pdu:
        port state: 61 (ACT AGG SYNC COL DIST )

Details partner lacp pdu:
        port state: 63 (ACT TIMEOUT AGG SYNC COL DIST )

LACP Packet Statistics:
                 Tx      Rx
0000:02:00.0    13414   413
0000:02:00.1    13414   414
```

Here, you can get more insight of LACP running stats, including all LACP timers and PDU statistics about number of packets exchanged with the peer device. Of course, here the counters are LACP PDU only. If you need all packets received and sent through the bond interface, you can use `-n|--stats` option.

bond Packet Counters

The `-n|--stats` option is useful to look into the packet statistics of the bond interface. So far we've seen at least two ways of retrieving packet counters from a vif interface:

- vif --get X
- dpdkvifstats.py -v X

The DPDK bond interface is represented by the vRouter interface `vif0/0`, so you may think setting X to 0 in the above commands achieves the same effect. The problem is none of these tools print packet statistics for each member link of the bond. Let's take a look at an example here:

```
[root@a7s3 ~]# contrail-tools dpdkinfo --stats eth
Master Info:
RX Device Packets:28360664, Bytes:3233321316, Errors:0, Nombufs:0
Dropped RX Packets:0
TX Device Packets:28361174, Bytes:3234763122, Errors:0
Queue Rx: [0]28360664
      Tx: [0]28361174
      Rx Bytes: [0]3233321316
      Tx Bytes: [0]3234760294
      Errors:
---------------------------------------------------------------------

Slave Info(0000:02:00.0):
RX Device Packets:1421, Bytes:129257, Errors:0, Nombufs:0
Dropped RX Packets:0
TX Device Packets:28358167, Bytes:3234235595, Errors:0
```

```
Queue Rx: [0]1421
      Tx: [0]28358167
      Rx Bytes: [0]129257
      Tx Bytes: [0]3234232767
      Errors:
---------------------------------------------------------------------

Slave Info(0000:02:00.1):
RX Device Packets:28359275, Bytes:3233195707, Errors:0, Nombufs:0
Dropped RX Packets:0
TX Device Packets:3039, Bytes:531175, Errors:0
Queue Rx: [0]28359275
      Tx: [0]3039
      Rx Bytes: [0]3233195707
      Tx Bytes: [0]531175
      Errors:
---------------------------------------------------------------------
```

With the `--stats eth` option, dpdkinfo displays traffic distribution among all member links of a DPDK bond interfaces. For example, in this instance we are seeing the first member link (PCI bus 0000:02:00.0) receive 1421 packets, while the second member link (PCI bus 0000:02:00.1) receives 28359275 packets. It is obvious that the second member link carries most of the traffic. Maybe you are wondering why we ended up with imbalanced traffic distributions, because previously we've mentioned that `Transmit Hash Policy` is set to load balancing across member links. The reason is that in this test environment we are sending just one UDP flow!

With more flows we'll see how the balance happens. Let's send ten flows, but before that let's clear the current counters to make our second comparison easier:

```
[root@a7s3 ~]# contrail-tools vif --clear

Vif stats cleared successfully on all cores for all interfaces
```

Now, start the rapid script to send 64 flows, and check same `dpdkinfo` command output again:

```
[root@a7s3 ~]# contrail-tools dpdkinfo -n eth
Master Info:
RX Device Packets:471211, Bytes:53724144, Errors:0, Nombufs:0
Dropped RX Packets:0
TX Device Packets:471189, Bytes:53719798, Errors:0
Queue Rx: [0]471211
      Tx: [0]471190
      Rx Bytes: [0]53724144
      Tx Bytes: [0]53719884
      Errors:
---------------------------------------------------------------------

Slave Info(0000:02:00.0):
RX Device Packets:228370, Bytes:26033818, Errors:0, Nombufs:0
```

```
 Dropped RX Packets:0
 TX Device Packets:220073, Bytes:25090326, Errors:0
 Queue Rx: [0]228370
       Tx: [0]220076
       Rx Bytes: [0]26033818
       Tx Bytes: [0]25090640
       Errors:
 ---------------------------------------------------------------------

 Slave Info(0000:02:00.1):
 RX Device Packets:242872, Bytes:27693860, Errors:0, Nombufs:0
 Dropped RX Packets:0
 TX Device Packets:251148, Bytes:28633120, Errors:0
 Queue Rx: [0]242872
       Tx: [0]251158
       Rx Bytes: [0]27693860
       Tx Bytes: [0]28634260
       Errors:
 ---------------------------------------------------------------------
```

From the member link packet statistics, you can see that the traffic is balanced on both links.

Now that you understand the `-stats|-n` option provides the insight of member link usage reflected by a few RX/TX counters. Base on this information you can determine the load balance status of a DPDK bond interface. So far, all of the packet counters we've seen, no matter under master or members, are almost the same ones as what are provided by the vif command. In practice, if you need to get more extensive statistics, there is another option: `xstats|-x`. Let's check it out:

```
[root@a7s3 ~]# contrail-tools dpdkinfo -xall | grep -v : 0
Master Info:
Rx Packets:                         Rx Bytes:
    rx_good_packets: 852475379          rx_good_bytes: 97185979648
    rx_q0packets: 852475379             rx_q0bytes: 97185979648
Tx Packets:                         Tx Bytes:
    tx_good_packets: 852853117          tx_good_bytes: 97253818091
    tx_q0packets: 852853127             tx_q0bytes: 97253769503
Errors:
Others:
 ------------------------------------------------------------------

Slave Info(0):0000:02:00.0          Slave Info(1):0000:02:00.1
Rx Packets:                         Rx Packets:
  rx_good_packets: 412875343          rx_good_packets: 439600104
  rx_q0packets: 412875343             rx_q0packets: 439600104
  rx_size_64_packets: 5939            rx_size_64_packets: 19
  rx_size_65_to_127_packets: 412869003 rx_size_65_to_127_packets: 439553375
  rx_size_128_to_255_packets: 191     rx_size_128_to_255_packets: 42367
  rx_size_256_to_511_packets: 206     rx_size_256_to_511_packets: 1173
  rx_broadcast_packets: 5882          rx_size_512_to_1023_packets: 1242
  rx_multicast_packets: 6124          rx_size_1024_to_max_packets: 1922
  rx_total_packets: 412875340         rx_multicast_packets: 396
Tx Packets:                           rx_total_packets: 439600098
  tx_good_packets: 399807799        Tx Packets:
  tx_q0packets: 399807802             tx_good_packets: 453045397
  tx_total_packets: 399807792         tx_q0packets: 453045399
  tx_size_64_packets: 3552            tx_total_packets: 453045389
  tx_size_65_to_127_packets: 399717757 tx_size_65_to_127_packets: 453035768
  tx_size_128_to_255_packets: 59597   tx_size_128_to_255_packets: 6448
  tx_size_256_to_511_packets: 10695   tx_size_256_to_511_packets: 9
  tx_size_512_to_1023_packets: 831    tx_size_512_to_1023_packets: 1680
  tx_size_1024_to_max_packets: 15360  tx_size_1024_to_max_packets: 1484
  tx_multicast_packets: 6365          tx_multicast_packets: 6365
  tx_broadcast_packets: 2941        Rx Bytes:
Rx Bytes:                              rx_good_bytes: 50119065424
   rx_good_bytes: 47066921976          rx_q0bytes: 50119065424
   rx_q0bytes: 47066921976             rx_total_bytes: 50119064740
   rx_total_bytes: 47066921752      Tx Bytes:
Tx Bytes:                              tx_good_bytes: 51649995369
   tx_good_bytes: 45603831138          tx_q0bytes: 51649996187
   tx_q0bytes: 45603781752          Errors:
Errors:                             Others:
Others:                                rx_l3_l4_xsum_error: 439588641
   rx_l3_l4_xsum_error: 412856784      out_pkts_untagged: 474447816
   out_pkts_untagged: 549754060
 ------------------------------------------------------------------
```

As you can see, the output is *very* extensive – perhaps ten times more than what `vif`, `dpdkvifstats.py` and `dpdkinfo -n eth` provide. In fact, to shorten the output, we've removed all counters with a zero value in them, and also edited the output format to compact all texts into two columns. If you go through it quickly, you will be able to tell the fact that the majority part of the traffic is composed of packets with sizes between 65 to 127 bytes, and that is what we are sending from the rapid script. Increasing traffic packet size from rapid will end up with a different result:

```
[root@a7s3 ~]# contrail-tools dpdkinfo -xall | grep -v : 0
Master Info:
....
 --------------------------------------------------------------------
Slave Info(0):0000:02:00.0           Slave Info(1):0000:02:00.1
Rx Packets:                          Rx Packets:
    rx_good_packets: 7902180            rx_good_packets: 7896450
    rx_q0packets: 7902180               rx_q0packets: 7896450
    rx_size_64_packets: 302             rx_size_64_packets: 1
    rx_size_65_to_127_packets: 1731     rx_size_65_to_127_packets: 389
    rx_size_128_to_255_packets: 7900126 rx_size_128_to_255_packets: 7895820
    rx_size_256_to_511_packets: 15      rx_size_256_to_511_packets: 66
    rx_size_512_to_1023_packets: 3      rx_size_512_to_1023_packets: 69
    rx_size_1024_to_max_packets: 3      rx_size_1024_to_max_packets: 105
    rx_broadcast_packets: 299           rx_multicast_packets: 20
    rx_multicast_packets: 312           rx_total_packets: 7896450
    rx_total_packets: 7902180           Tx Packets:
Tx Packets:                             tx_good_packets: 8272747
    tx_good_packets: 7536810            tx_q0packets: 8272747
    tx_q0packets: 7536810               tx_total_packets: 8272747
    tx_total_packets: 7536810           tx_size_65_to_127_packets: 179
    tx_size_64_packets: 181             tx_size_128_to_255_packets: 8272496
    tx_size_65_to_127_packets: 290      tx_size_256_to_511_packets: 17
    tx_size_128_to_255_packets: 7535143 tx_size_512_to_1023_packets: 53
    tx_size_256_to_511_packets: 223     tx_size_1024_to_max_packets: 2
    tx_size_512_to_1023_packets: 90     tx_multicast_packets: 324
    tx_size_1024_to_max_packets: 883  Rx Bytes:
```

```
    tx_multicast_packets: 323           rx_good_bytes: 1405706413
    tx_broadcast_packets: 150           rx_q0bytes: 1405706413
Rx Bytes:                               rx_total_bytes: 1405706413
    rx_good_bytes: 1406393359         Tx Bytes:
    rx_q0bytes: 1406393359               tx_good_bytes: 1472542701
    rx_total_bytes: 1406393359           tx_q0bytes: 1472542701
Tx Bytes:                                Errors:
    tx_good_bytes: 1342701308         Others:
    tx_q0bytes: 1342698774               rx_l3_l4_xsum_error: 7895846
Errors:                                  out_pkts_untagged: 3532820029
Others:
    rx_l3_l4_xsum_error: 7901213
    out_pkts_untagged: 3249154601
 --------------------------------------------------------------------
```

We won't discuss all the counters listed in this output, for now, just add dpdkinfo with these two options `-n|stats` and `-x|xstats` in your DPDK vRouter troubleshooting toolkits. Considering using them to collect information whenever you run into traffic loss issues during your lab test or production deployment.

Next, we'll explore another interesting option: `-c|--lcore`.

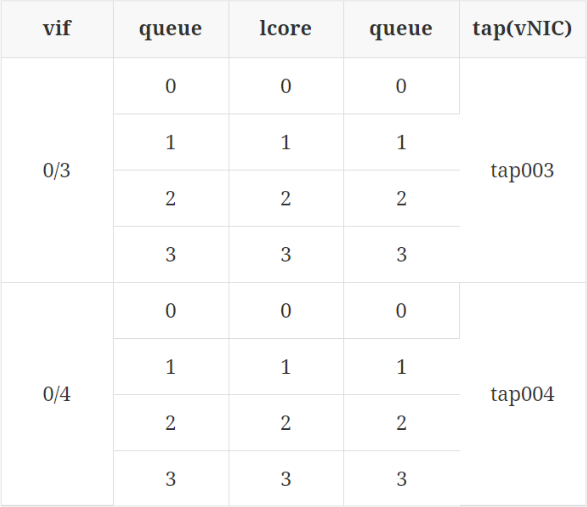
lcore

There are several key concepts we've been trying to illustrate in this book. Among others, at least three of them are often mentioned together: `lcore`, `interface`, and `queue`, but before we start introducing the fourth, the `-c|--lcore` option, let's briefly review these concepts:

- `lcore`: lcore is a thread in vRouter DPDK process running in user space.
- `interface`: This is the endpoint of connections between the vRouter and the other VM, or between vRouter and the outside of the compute. At the vRouter and VM end, the interfaces are called vif and tap interfaces, respectively. There are also bond0 physical interface in DPDK user space and vhost0 interface in Linux kernel. The former is the physically NIC bundle connecting to the peer device, and the latter gives the host an IP address and through which the vRouter agent can exchange control plane messages with the controller.
- `Queue`: For each interface there are some queues created. They are essentially allocated memory to hold the packets.

The CPU cores connect all these objects together. As of the writing of this book, the implementation is to have one-to-one mapping between the number of CPU cores allocated to vRouter and the number of interface queues. For example, if four CPUs are allocated to the DPDK vRouter forwarding threads (the `lcores`), then four lcores will be created, and four DPDK interface queues will be created for each vif interface. The same rule applies to the VM. You assign four CPU cores to a VM, then by default, Nova will create four queues for a tap interface in the VM. That said, of course, `multiple queue` as a feature needs to be turned on in Nova. We can illustrate with Table 6.1.

Table 6.1 Queues Created for Vif Interfaces



vif	queue	lcore	queue	tap(vNIC)
0/3	0	0	0	tap003
	1	1	1	
	2	2	2	
	3	3	3	
0/4	0	0	0	tap004
	1	1	1	
	2	2	2	
	3	3	3	

This is just a simple example. In production deployment there are a lot more conditions to consider, and a lot of confusion arises. Common questions are:

- What if the tap interface queue number is different than the vif queue number? What will happen when there are eight lcores but one of our VMs is running four queues in its tap interface?
- Will vif0/3 queue0 always be served by lcore0, instead of other lcores? If not, how to determine which vif queue goes to which lcore? Is there a chance that imbalanced lcores to queue mapping happens, so that some lcores are overloaded and some lcores are relatively idle?

To answer these questions, we need a tool to reveal the secret of actual mapping between lcores and queues from different vif interfaces. This is the moment for `-c|--lcore` option of `dpdkinfo` to show its power. Again, let's start with an example:

```
[root@a7s3 ~]# contrail-tools dpdkinfo -c
No. of forwarding lcores: 2
No. of interfaces: 4
Lcore 0:
        Interface: bond0.101            Queue ID: 0
        Interface: vhost0               Queue ID: 0

Lcore 1:
        Interface: bond0.101            Queue ID: 1
        Interface: tap41a9ab05-64       Queue ID: 0
```

Let's start from the first line. In this example, we have allocated two CPU cores to the DPDK vRouter forwarding lcores, so we have two forwarding lcores running in total.

Then, the second line provides the number of vRouter interfaces in the compute. We have four of them in total. One vif0/4 connecting to VM tap interface `tap41a9ab05-64`, and three mandatory, vif0/0, vif0/1, and vif0/2, connecting to bond, vhost0, and pkt0, respectively. Here, we have created just one VM (actually this is nothing but the PROX gen VM we've created earlier) with only one tap interface.

Let's focus on the third line onward. The output is listing all forwarding lcores that are currently configured in vRouter, and for each lcore it lists interfaces that each lcore is associated with, in another words, interfaces this core is serving.

Please note that there are some inconsistencies in terms of the lcore numbering in different tools:

- In `dpdkvifstats.py` script, the forwarding lcore number starts from one, so Core 1 refers to the first forwarding lcore.
- In `dpdkinfo -c` output, forwarding lcore number starts from zero, so Lcore 0 refers to the first forwarding lcore.
- In `vif` output, forwarding lcore number starts from ten, so --core 10 refers to the first forwarding lcore.

This can cause confusion in our discussions. To make it consistent, in the rest of this chapter we'll use the first forwarding lcore, `fwd lcore#10`, or simply `lcore#10`; the second forwarding `lcore`, `fwd lcore#11`, or simply `lcore#11`, and so on, to indicate Lcore 0, Lcore 1 in dpdkinfo-c output, Core 1, Core 2 in dpdkvifstats.py script output, and Core 10, Core 11 in `vif` output, respectively. Here's a better visualization:

vif	dpdkinfo -c	dpdkvifstats.py	meaning
Core 10	Lcore 0	Core 1	1st forwarding lcore: lcore#10
Core 11	Lcore 1	Core 2	2nd forwarding lcore: lcore#11

Okay, as you may have realized, in the VM interface we use just one queue, which means the multiple queue feature on the VM interface is not enabled. Therefore the VM tap interface has only one queue connecting to its peering vRouter interface. Correspondingly, only one queue in the vRouter interface is needed and only one lcore is required to serve the packet forwarding in the vif interface.

First, let's look at the `bond0` and `vhost0` interfaces. The bond0 are the physical interfaces, and will always have `multiple queues` enabled, that is why it has two queues, and both lcores serve it. The `vhost0` interface is a control plane Linux interface. At the time of writing of this book, the implementation is to hard-code vhost0 with one queue only. The first forwarding thread `lcore#10` got it. This is not the focus in this section, but it's helpful to understand the whole output.

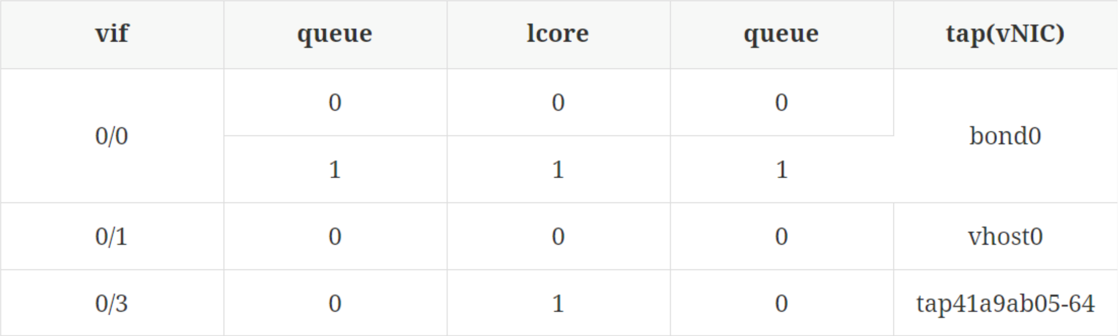
Finally, let's look at the last line, the VM tap interface. From the output, you can see it is the second forwarding lcore (`lcore#11`) being assigned to this VM interface. You're probably wondering whether it's just randomly chosen out of the two lcores or if some algorithm is used. It is not like that. Currently the allocation basically follows a simple method: *The least used lcore, in terms of the number of interface queues it is serving, will be assigned to serve the next interface queue.*

Based on what we just explained, `lcore#10` took two interfaces (`bond0.101` and `vhost0`) while `lcore#11` took just one (`bond0.101`), so it's lcore#11's turn to take the next interface and queue.

The vNIC queues are assigned to logical cores in the following algorithm: the forwarding core that is currently polling the least number of queues is selected, with a tie won by the core with the lowest number (the first forwarding core lcore#10).

You'll see more examples in later sections, when we'll test out the tie breaker and other things. For now, this mapping looks like this Table 6.2.

Table 6.2 vNIC Queues Assigned to Logical Cores



vif	queue	lcore	queue	tap(vNIC)
0/0	0	0	0	bond0
	1	1	1	
0/1	0	0	0	vhost0
0/3	0	1	0	tap41a9ab05-64

Now that we've gone through the `dpdkinfo` command and demonstrated some of the most commonly used options, you can quickly display a lot of useful information about DPDK and DPDK vRouter running status. We'll review this again later in our test case studies. The information is important before working on any deployment or troubleshooting task in the setup. However, when things go wrong, instead of just relying on the DPDK commands output, you may also want to check into the log messages to verify the current running status is what you expected it to be.

Next we'll take a look at DPDK vRouter log messages.

DPDK vRouter Log Files

Contrail's DPDK vRouter data plane log file is named `contrail-vrouter-dpdk.log`. Depending on the version or installation methods, it can be located in different folders or even with a totally different name. For example:

- In latest TripleO deployment: `/var/log/containers/contrail/dpdk/contrail-vrouter-dpdk.log`
- In latest Ansible deployment: `/var/log/contrail/contrail-vrouter-dpdk.log`
- In older 3.x Ubuntu deployment: `/var/log/contrail.log`

This log file contains lots of good information that is helpful to understand the current running status. Of course, understanding the log messages is important during a troubleshooting process.

DPDK vRouter Parameters

Each time the vRouter is started, the main configuration parameters are listed in the log file during the vRouter initialization stage. You can see the DPDK library version that has be used to build the DPDK vRouter binary program:

```
2020-09-15 20:27:22,381 VROUTER: vRouter version: {build-info:
[{build-time: 2020-09-15 01:07:25.101398, build-hostname:
contrail-build-r2008-rhel-115-generic-20200914170527.novalocal, build-user:
contrail-builder, build-version: 2008}]}
2020-09-15 20:27:22,382 VROUTER: DPDK version: DPDK 19.11.0
2020-09-15 20:27:23,046 VROUTER: Log file : /var/log/contrail/contrail-vrouter-dpdk.log
2020-09-15 20:27:23,046 VROUTER: Bridge Table limit:          262144
2020-09-15 20:27:23,046 VROUTER: Bridge Table overflow limit: 53248
2020-09-15 20:27:23,046 VROUTER: Flow Table limit:            524288
2020-09-15 20:27:23,046 VROUTER: Flow Table overflow limit:   105472
2020-09-15 20:27:23,046 VROUTER: MPLS labels limit:           5120
2020-09-15 20:27:23,046 VROUTER: Nexthops limit:              32768
2020-09-15 20:27:23,046 VROUTER: VRF tables limit:            4096
2020-09-15 20:27:23,046 VROUTER: Packet pool size:            16384
2020-09-15 20:27:23,046 VROUTER: PMD Tx Descriptor size:      128
2020-09-15 20:27:23,046 VROUTER: PMD Rx Descriptor size:      128
2020-09-15 20:27:23,046 VROUTER: Maximum packet size:         9216
2020-09-15 20:27:23,046 VROUTER: Maximum log buffer size:     200
2020-09-15 20:27:23,046 VROUTER: VR_DPDK_RX_RING_SZ:          2048
2020-09-15 20:27:23,046 VROUTER: VR_DPDK_TX_RING_SZ:          2048
2020-09-15 20:27:23,046 VROUTER: VR_DPDK_YIELD_OPTION:        0
2020-09-15 20:27:23,046 VROUTER: VR_SERVICE_CORE_MASK:        0x10
2020-09-15 20:27:23,046 VROUTER: VR_DPDK_CTRL_THREAD_MASK:    0x10
2020-09-15 20:27:23,046 VROUTER: Unconditional Close Flow on TCP RST: 0
2020-09-15 20:27:23,046 VROUTER: EAL arguments:
2020-09-15 20:27:23,046 VROUTER:            -n  4
2020-09-15 20:27:23,046 VROUTER:  --socket-mem  1024
```

You can see the complete list of vRouter start up parameters on this Contrail vRouter, for example:

- `build-version` 2008
- it's running DPDK Version `19.11.0`
- `Nexthops limit` parameter is configured as 32768, decreased from the default value (65536)
- CPU core #4 is pinned to be used by control and service thread (`VR_SERVICE_CORE_MASK: 0x10`)

We can compare this information with what we can display with these command line tools and see if they are consistent:

- `contrail-version`
- `dpdkinfo -v`
- `vrouter --info`
- `taskset`

Any inconsistency will provide a clue to proceed in that area.

Polling Core Allocation

In Chapter 3 we introduced the DPDK vRouter process. It is a multiple thread application, and the threads fall into different categories based on their roles. This is also reflected by some log entries. Before diving into the logs, let's do a quick review of the three thread categories:

- Control threads: These are generated by DPDK libraries and are used during Contrail vRouter startup for DPDK initialization. Control threads are not our focus in this book.
- Service threads: These two service threads are totally hard-coded named lcore0 through lcore9. Each lcore has its own role. For example, lcore9 serves Netlink connections between agent and vRouter data plane. Details of each lcore's rule is out of this book's scope. You just need to know that they are used to serve communication between the vRouter agent and vRouter forwarding plane.
- Forwarding threads: After service threads, from lcore10 and onward, the forwarding threads are the horsepower that performs the packet forwarding tasks and determines the performance of the DPDK vRouter. *This has been the main focus of our book*.

NOTE In service threads, `lcore3` to `lcore7` are never used in Contrail DPDK vRouter.

Okay, now let's take a look at an interesting log entry:

```
2020-09-16 09:06:50,886 VROUTER: --lcores  (0-2)@(10,34),(8-9)@(10,34),*10@2,11@4,12@6,13@8*
```

Here, the string `--lcores` means a service thread, or a forwarding thread. Following this string are a few coupled numbers connected by @ - NUMBER@NUMBER, which are separated by commas. How to decode these? Well, to understand this you need to understand CPU pinning. To achieve maximum performance, we're pinning each of the service and forwarding threads (or `lcores`) with a few specific CPU cores, so each thread will be served by dedicated CPUs that are isolated from any other system tasks. So this log reads:

- Service threads, that is lcore0 to lcore2, and lcore8-lcore9 in the message, are all pinned to two CPU cores: core#10 and CPU core#34. The pinning is configured by the `SERVICE_CORE_MASK` parameter.
- Forwarding threads, lcore10 to lcore13, are allocated and are pinned to CPU core#2, core#4, core#6 and core#8, respectively. This is configured from the `CPU_LIST` parameter.

Internal Load Balancing

In some situations the polling core performs a new hash calculation to distribute the polled packets to another processing core. This is a DPDK pipeline model implemented in the vRouter.

This distribution behavior can be observed in the following messages in DPDK log file:

```
2020-01-07 13:08:01,403 VROUTER: Lcore 10: distributing MPLSoGRE packets to [11,12,13]
2020-01-07 13:08:01,403 VROUTER: Lcore 11: distributing MPLSoGRE packets to [10,12,13]
2020-01-07 13:08:01,403 VROUTER: Lcore 12: distributing MPLSoGRE packets to [10,11,13]
2020-01-07 13:08:01,404 VROUTER: Lcore 13: distributing MPLSoGRE packets to [10,11,12]
```

Here the logs show `MPLSoGRE`, but it actually applies to both MPLSoGRE or VxLAN packets. Historically, this only happened when MPLSoGRE was supported. So it remains like that in the software code. Here it means both MPLSoGRE and VxLAN packets will be distributed via hashing by the polling core. See Figure 6.4.

Each time a new virtual interface is connected to the vRouter, a vif port is created on the vRouter with the same number of queues as the number of polling CPUs (specified in CPU_LIST parameter). Each queue is created and handled by only one of the vRouter polling cores. So, for each vif we have a one-to-one mapping between vRouter polling cores and RX queues. This mapping can be seen from the `dpdkinfo -c` command output we've introduced. The same can be observed in the DPDK vRouter logs:

```
2019-09-24 16:36:50,011 VROUTER: Adding vif 8 (gen. 37) virtual device tap66e68bc1-a9
....
2019-09-24 16:36:50,012 VROUTER: lcore 12 RX from HW queue 0
2019-09-24 16:36:50,012 VROUTER: lcore 13 RX from HW queue 1
2019-09-24 16:36:50,012 VROUTER: lcore 10 RX from HW queue 2
2019-09-24 16:36:50,012 VROUTER: lcore 11 RX from HW queue 3
```

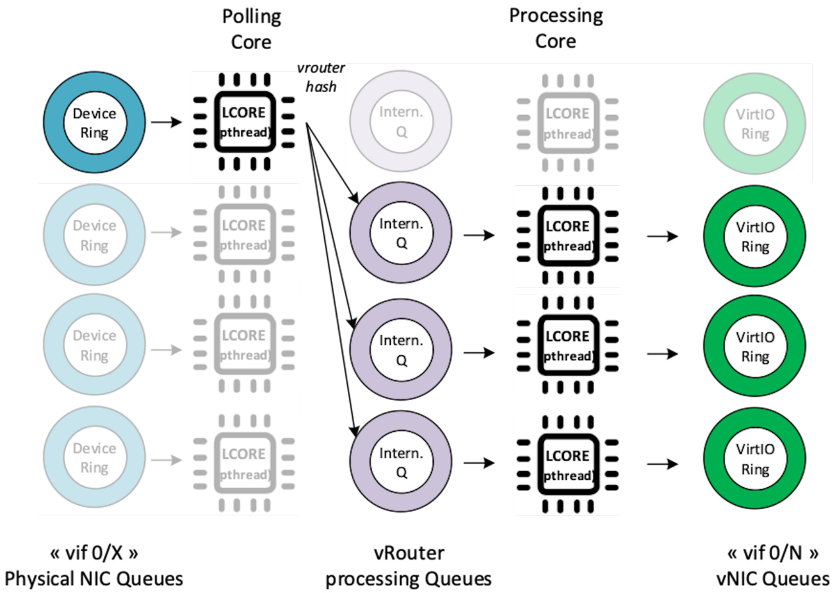


Figure 6.4 Virtual Interface Queues

Here, the vif interface 0/8 is created in order to connect the virtual NIC tap66e68bc1-a9 to the vRouter. Because four forwarding lcores are configured, this vif is created with 4 queues, namely q0 to q3, which are respectively handled by polling cores 12,13,10, and 11.

When a polling queue is enabled on the vRouter, a ring activation message is generated in the Contrail DPDK log file.

The vrings correspond to both transmit and receive queues, as seen in Figure 6.5:

- The transmit queues are the even numbers. Divide them by two to get the queue number: vring 0 is TX queue 0, vring 2 is TX queue 1, … and so on.
- The receive queues are the odd numbers. Divide them by two (discard the remainder) to get the queue number: vring 1 is RX queue 0, vring 3 is RX queue 1, … and so on.
- Ready state 1 = enabled, ready state 0 = disabled.

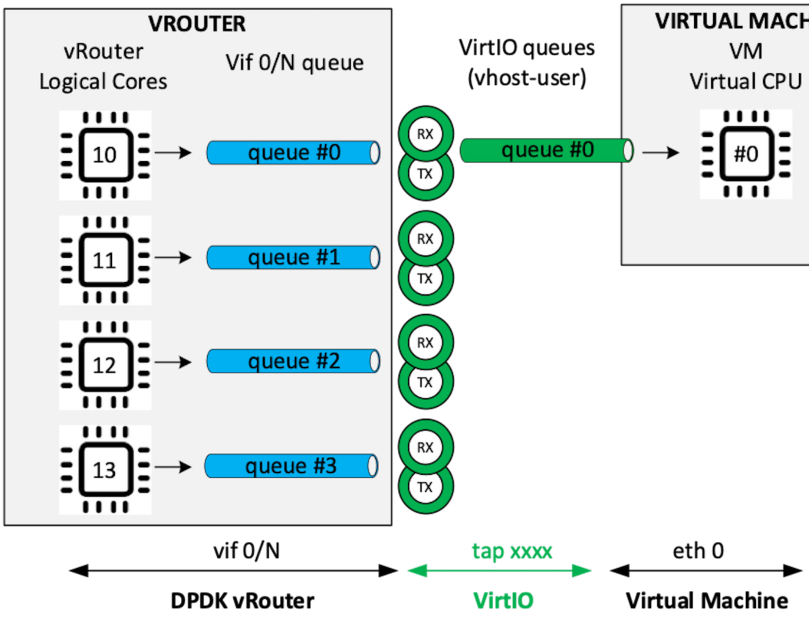


Figure 6.5 Vrings Correspond to Both Transmit and Receive Queues

In this next example, only one RX (and TX) queue is enabled on the vRouter vif interface. A single queue virtual machine interface is connected to the vRouter port:

```
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 0 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 1 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 2 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 3 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 4 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 5 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 6 ready state 0
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 7 ready state 0
```

And in the next example, four RX (and TX) queues are enabled on the vRouter vif interface, but a virtual machine interface having more than four queues is connected to the vRouter port:

```
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 0 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 1 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 2 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 3 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 4 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 5 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 6 ready state 1
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: setting vring 7 ready state 1
2019-09-24 16:37:46,693 UVHOST: vr_uvhm_set_vring_enable: Can not disable TX queue 4 (only 4 queues)
```

```
2019-09-24 16:37:46,693 UVHOST: Client _tap66e68bc1-a9: handling message 18
2019-09-24 16:37:46,693 UVHOST: vr_uvhm_set_vring_enable: Can not disable RX queue 4 (only 4 queues)
```

As there are more than four queues on the virtual machine interface, some queues must not be enabled on the virtual machine NIC. Unfortunately, these queues can ’t be disabled on the virtual machine. Therefore, this setup is faulty. See Figure 6.6.

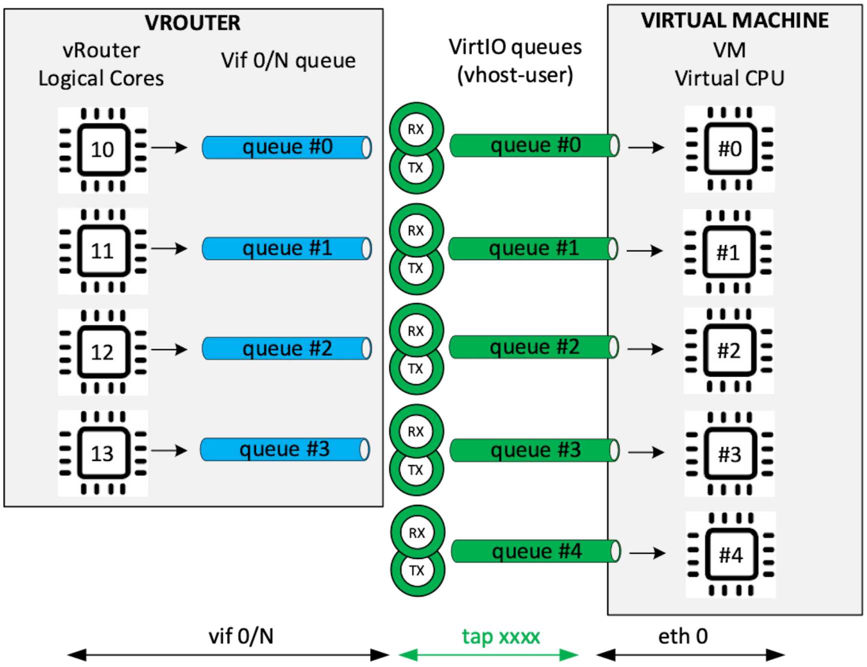


Figure 6.6 A Faulty vRouter and VM Queuing Setup

DPDK vRouter Case Studies

In previous sections, we’ve introduced some DPDK tools and explained some important log entries to help collect DPDK vRouter running status. Let’s drill down into these tools.

Single Queue

Having understood the lcore mapping basics, let’s test some traffic.

One Way Single Flow: VM to Fabric

To make it very simple, we are sending single uni-directional UDP flow from the PROX gen VM. You can list current flows you have in vRouter to confirm:

```
 [root@a7s3 ~]# contrail-tools flow -l
 Flow table(size 161218560, entries 629760)

 ......
     Index                Source:Port/Destination:Port                      Proto(V)
 -----------------------------------------------------------------------------------
     40196<=>436016       192.168.0.106:59514                                 6 (3)
                          192.168.0.104:22
 (Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):36,  Stats:503/35823,
  SPort 56703, TTL 0, Sinfo 8.0.0.3)

    436016<=>40196        192.168.0.104:22                                    6 (3)
                          192.168.0.106:59514
 (Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):27,  Stats:511/71619,
  SPort 49812, TTL 0, Sinfo 4.0.0.0)

    62792<=>172020       192.168.0.106:48664                                 6 (3)
                         192.168.0.104:8474
(Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):36,  Stats:3828/296117,
 SPort 63470, TTL 0, Sinfo 8.0.0.3)

   172020<=>62792        192.168.0.104:8474                                  6 (3)
                         192.168.0.106:48664
(Gen: 1, K(nh):27, Action:F, Flags:, TCP:SSrEEr, QOS:-1, S(nh):27,  Stats:2739/274615,
 SPort 52648, TTL 0, Sinfo 4.0.0.0)

     38232<=>257372       192.168.1.105:32768                                17 (2)
                          192.168.1.104:32770
 (Gen: 5, K(nh):30, Action:F, Flags:, QOS:-1, S(nh):37,  Stats:0/0,  SPort 61739,
  TTL 0, Sinfo 0.0.0.0)

    257372<=>38232        192.168.1.104:32770                                17 (2)
                          192.168.1.105:32768
 (Gen: 5, K(nh):30, Action:F, Flags:, QOS:-1, S(nh):30,  Stats:390003/48360372,
  SPort 62464, TTL 0, Sinfo 3.0.0.0)
```

Here, you can see six vRouter flows, which are in fact three groups. The first two groups with index pairs `40196/436016` and `62792/172020`, are generated by the control messages from the rapid jump VM into the PROX gen VM. The last group of flows with index pairs `38232/257372` is our single flow test traffic. The stats `39003/48360372` show the traffic flow is sent from gen VM (`192.168.1.104:32770`) to swap VM (`192.168.1.105:32768`).

In Contrail vRouter, flows are generated in pairs. For any traffic, even if it is one direction only, vRouter will generate a reverse flow for it. This is because in the real world most of the traffic is bidirectional, so having a separate entry built for each direction is required. In this case, from PROX, we are generating uni-directional traffic so only the flow of that direction has packet stats. The pairing flow entry is generated as well, but packet statistics show nothing.

Let’s clear the vif counters and collect the statistics using `dpdkvifstats.py` tool:

```
[root@a7s3 ~]# contrail-tools vif --clear

Vif stats cleared successfully on all cores for all interfaces

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 3 -c 2
 -------------------------------------------------------------------
| Core 1 | TX pps: 0 | RX pps: 1504 | TX bps: 0   | RX bps: 90240
| Core 2 | TX pps: 1 | RX pps: 1    | TX bps: 42  | RX bps: 56
| Total  | TX pps: 1 | RX pps: 1505 | TX bps: 336 | RX bps: 722368
 -------------------------------------------------------------------

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 0 -c 2
 ---------------------------------------------------------------------
| Core 1 | TX pps: 1512 | RX pps: 2 | TX bps: 166320  | RX bps: 132
| Core 2 | TX pps: 1    | RX pps: 1 | TX bps: 112     | RX bps: 110
| Total  | TX pps: 1513 | RX pps: 3 | TX bps: 1331456 | RX bps: 1936
 ---------------------------------------------------------------------
```

From the first capture on the vRouter interface connecting to the PROX gen VM tap interface (`-v 3`), we are seeing that lcore#10 received the traffic – you can tell from the RX speed 1504 pps showing in Core 1 only. The second capture on the vRouter interface toward the bond interface (`-v 0`) confirms the same – it is the same lcore#10 (Core 1 here) that is sending the traffic to the bond interface, at speeds of 1512 pps, almost the same as the speed it received the traffic from the VM tap interface. The forwarding path is illustrated here:

```
VM: tap41a9ab05-64 => vif0/3 => lcore#10 => vif0/0 => bond0
```

This seems to be weird, doesn’t it? Remember, previously based on the core-interface mapping given by `dpdkinfo -c`, we already know it was the lcore#11 serving our VM interface, not the other one. Accordingly, in the `dpdkvifstats.py` output, that should be Core 2 instead of Core 1. Let’s revisit the mapping:

```
[root@a7s3 ~]# contrail-tools dpdkinfo -c
No. of forwarding lcores: 2
No. of interfaces: 4
Lcore 0:
        Interface: bond0.101            Queue ID: 0
        Interface: vhost0               Queue ID: 0

Lcore 1:
        Interface: bond0.101            Queue ID: 1
        Interface: tap41a9ab05-64       Queue ID: 0
```

So, we’re right. The flow that is expected should be something like this:

```
VM: tap41a9ab05-64 => vif0/3 => lcore#11 => vif0/0 => bond0
```

Well, if you remember what you read in Chapter 3, you probably will know the answer. When a packet flows from the PROX gen VM to the bond, vRouter uses a `pipeline model` to process the packet. What that really means is, the interface’s serving lcore, that is the second forwarding lcore, in our case based on `dpdkinfo -c` output, will poll it out of the vif interface. In Chapter 3, when we introduced the vRouter packet forwarding process, we mentioned that when traffic flows from the vif connecting VM tap interface to vif0/0, all packets will be distributed by the polling lcore to other lcores for processing. The distribution is calculated based on the hash of the packet header.

Apparently, the polling core here, based on the mapping above, is `lcore#11`, and the only other lcore is the first forwarding lcore `lcore#10`. So packets from VM got polled by the lcore#11 and then distributed to the `lcore#10`, which then forwarded them to the fabric interface vif0/0. Currently `dpdkvifstats.py` does not tell us much about these details, but if you collect `vif` output, you'll see additional clues:

```
[root@a7s3 ~]# contrail-tools vif --get 3 --core 10
Vrouter Interface Table

......
vif0/3      PMD: tap41a9ab05-64 NH: 38
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104
            Vrf:2 Mcast Vrf:2 Flags:L3L2DEr QOS:-1 Ref:12
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Core 10 RX packets:31272  bytes:1876320 errors:0
            Core 10 TX packets:0  bytes:0 errors:0
            Drops:18660668

[root@a7s3 ~]# contrail-tools vif --get 3 --core 11
Vrouter Interface Table

......
vif0/3      PMD: tap41a9ab05-64 NH: 38
            Type:Virtual HWaddr:00:00:5e:00:01:00 IPaddr:192.168.1.104
            Vrf:2 Mcast Vrf:2 Flags:L3L2DEr QOS:-1 Ref:12
            Core 11 RX queue  packets:35384 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Core 11 RX packets:26  bytes:1092 errors:0
            Core 11 TX packets:24  bytes:1008 errors:0
            Drops:18660668
```

There is an RX queue counter, `Core 11 RX queue  packets:35384`, that gives a very important clue about this *inter-core load-balancing*. `Core 11`, our second forwarding lcore, polled the packet first from vif0/3 into its RX queue. Instead of processing the packet, it distributed the packets onto the first forwarding lcore, `Core10`, which, then processed them. That is why same number of packets are counted as `RX packets` in `Core 10`. Therefore, the full forwarding path of this traffic flow is like this:

```
                            (polling lcore) (processing lcore)
VM: tap41a9ab05-64 => vif0/3 => lcore#11 => lcore#10 => vif0/0 => bond0
```

For the sake of completeness, we also captured the `vif` command on fabric interface vif0/0:

```
[root@a7s3 ~]# contrail-tools vif --get 0 --core 10
Vrouter Interface Table

......
vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
            Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18
            Core 10 RX device packets:199  bytes:49057 errors:0
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:02:00.0  Status: UP  Driver: net_ixgbe
            Slave Interface(1): 0000:02:00.1  Status: UP  Driver: net_ixgbe
            Vlan Id: 101  VLAN fwd Interface: vfw
            Core 10 RX packets:131  bytes:37595 errors:0
            Core 10 TX packets:48756  bytes:5362888 errors:0
            Drops:0
            Core 10 TX device packets:49024  bytes:5730372 errors:0

[root@a7s3 ~]# contrail-tools vif --get 0 --core 11
Vrouter Interface Table

......
vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
            Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:02:00.0  Status: UP  Driver: net_ixgbe
            Slave Interface(1): 0000:02:00.1  Status: UP  Driver: net_ixgbe
            Vlan Id: 101  VLAN fwd Interface: vfw
            Core 11 RX packets:67  bytes:9860 errors:0
            Core 11 TX packets:181  bytes:162062 errors:0
            Drops:0
```

Here after the first forwarding lcore serving vif0/0 processed the packets, it sent them out of vif0/0, which is reflected as `TX packets`.

One important thing to point out here is that what we've tested and demonstrated here is the DPDK vRouter's default behavior with the current parameters. Please keep in mind that vRouter is very configurable. There is one vRouter configuration option introduced in release R2008 which will change this default `pipeline model` behavior. This option is `--vr_no_load_balance`, and you can verify the current running vRouter-DPDK process command line in your setup with `ps` command. With that configured, vRouter will change to the so-called `run to complete model`, which means that the same lcore that polled the packet will continue to process/forward it. This requires reboot of DPDK vRouter, and we'll let you test the scenarios in your own lab.

This concludes the analysis of traffic forwarding in the direction of the VM to fabric. Next let's take a look at the returning direction: from fabric (vif0/0) to VM (vif0/3).

Returning Traffic: Fabric to VM

Now let's do the returning traffic. We configure the swap VM in such a way that it loops whatever it receives back to the sender. Here is the capture:

```
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 0 -c 2
 ---------------------------------------------------------------------------------
 | Core 1 | TX pps: 85844 | RX pps: 16    | TX bps: 14936710  | RX bps: 1940     ..
 | Core 2 | TX pps: 1     | RX pps: 85846 | TX bps: 88        | RX bps: 14937132 ..
 | Total  | TX pps: 85845 | RX pps: 85862 | TX bps: 119494384 | RX bps: 119512576..
 ---------------------------------------------------------------------------------
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -v 3 -c 2
 ---------------------------------------------------------------------------------
 | Core 1 | TX pps: 0     | RX pps: 85274 | TX bps: 0        | RX bps: 10574058 ..
 | Core 2 | TX pps: 85278 | RX pps: 1     | TX bps: 10574431 | RX bps: 56       ..
 | Total  | TX pps: 85278 | RX pps: 85275 | TX bps: 84595448 | RX bps: 84592912 ..
 --------------------------------------------------------------------------------
```

Here, we are looking at the returning traffic from the fabric back to the PROX gen VM.

Let's focus on seeing the RX in vif 0/0 and TX in vif0/3, and the data shows lcore#11 received the packets from vif0/0 and forwarded them out of vif0/3, the forwarding path is illustrated below:

```
              RX                            TX
fabric: bond0 => vif0/0 => lcore#11 => vif0/3 => tap41a9ab05-64 => VM
```

To confirm if this lcore#11 that is "forwarding" packets is also the one that did the "polling", we'll need to look at the vif capture and looking for the "RX queue packets" counter as what we've seen before:

```
[root@a7s3 ~]# contrail-tools vif --get 0 --core 10
Vrouter Interface Table

......
vif0/0     PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
          Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0
          Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18
          Core 10 RX device packets:3481584  bytes:619708685 errors:0         RX queue errors to lcore
0 0 0 0 0 0 0 0 0 0 0 0
          Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
           Slave Interface(0): 0000:02:00.0  Status: UP  Driver: net_ixgbe
            Slave Interface(1): 0000:02:00.1  Status: UP  Driver: net_ixgbe
            Vlan Id: 101  VLAN fwd Interface: vfw
            Core 10 RX packets:676  bytes:106243 errors:0
            Core 10 TX packets:3482241  bytes:605899226 errors:0
            Drops:99
            Core 10 TX device packets:3482474  bytes:619966089 errors:0

[root@a7s3 ~]# contrail-tools vif --get 0 --core 11
Vrouter Interface Table

......
vif0/0      PCI: 0000:00:00.0 (Speed 20000, Duplex 1) NH: 4
            Type:Physical HWaddr:90:e2:ba:c3:af:20 IPaddr:0.0.0.0
            Vrf:0 Mcast Vrf:65535 Flags:TcL3L2VpVofEr QOS:-1 Ref:18
            RX queue errors to lcore 0 0 0 0 0 0 0 0 0 0 0 0
            Fabric Interface: eth_bond_bond0  Status: UP  Driver: net_bonding
            Slave Interface(0): 0000:02:00.0  Status: UP  Driver: net_ixgbe
            Slave Interface(1): 0000:02:00.1  Status: UP  Driver: net_ixgbe
            Vlan Id: 101  VLAN fwd Interface: vfw
            Core 11 RX packets:3594939  bytes:625517508 errors:0
            Core 11 TX packets:166  bytes:133391 errors:0
            Drops:99
```

There isn't an "RX queue packets" counter like the one we saw in the traffic on the other direction (from VM to fabric interface). Therefore, in this direction of traffic sent from fabric to VM, we don't see any "inter-core load balancing" behavior like we elaborated on previously.

You may notice that there is a "Core 10 RX device packets" and a "Core 10 TX device packets" counter displayed on lcore#10 only. As of the time of writing of this book, these counters show the total packets received and sent by the NIC, so they have nothing to do with the inter-core load balancing behavior.

This is because this test vRouter is using VxLAN encapsulations for the fabric interface, which follows the run-to-completion model explained in Chapter 3. Therefore, one forwarding lcore polls for packets, makes forwarding decision, and forwards the packet out of the other interface. This "rule" applies to the traffic flows from physical NIC toward the VM, regardless of number of lcores enabled. In the test below, we are seeing that with four lcores enabled this conclusion remains unchanged:

```
   [root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 4
   | VIF 3   |Core 1  | TX pps: 0        | RX pps: 0        | TX bps: 0        | RX bps: 0        | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
   | VIF 3   |Core 2  | TX pps: 1        | RX pps: 1        | TX bps: 448      | RX bps: 448      | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
   | VIF 3   |Core 3  | TX pps: 0        | RX pps: 75778    | TX bps: 0        | RX bps: 36373760  | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
   | VIF 3   |Core 4  | TX pps: 75776    | RX pps: 0        | TX bps: 36372480  | RX bps: 0        | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
   | VIF 0   |Core 1  | TX pps: 10       | RX pps: 1        | TX bps: 12424    | RX bps: 448      | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
   | VIF 0   |Core 2  | TX pps: 0        | RX pps: 0        | TX bps: 0        | RX bps: 176      | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
   | VIF 0   |Core 3  | TX pps: 76810    | RX pps: 1        | TX bps: 67593384  | RX bps: 816      | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
   | VIF 0   |Core 4  | TX pps: 1        | RX pps: 76839    | TX bps: 912      | RX bps: 67619992  | TX 
error: 0        | RX error 0        | TX port error: 0        | RX queue error 0        |
    ------------------------------------------------------------------------
    |                               pps per Core                           |
    ------------------------------------------------------------------------
    |Core 1   |TX + RX pps: 19        | TX pps 10         | RX pps 9         |
    |Core 2   |TX + RX pps: 2         | TX pps 1          | RX pps 1         |
    |Core 3   |TX + RX pps: 152589    | TX pps 76810      | RX pps 75779     |
    |Core 4   |TX + RX pps: 152627    | TX pps 75788      | RX pps 76839     |
    ------------------------------------------------------------------------
    |Total    |TX + RX pps: 305237    | TX pps 152609     | RX pps 152628    |
    ------------------------------------------------------------------------
```

This concludes our analysis to the bidirectional single flow traffic. As you can see, one benefit of having traffic generator/swapper built into lab environment is that you can fine tune the generator to send traffic in a very specific pattern, so that you can take a deep look at the counters and analyze the vRouter traffic forwarding behavior. This is very helpful for learning purposes, but in production, you probably never expect to have such a luxury since the traffic pattern in the field is usually much more complex. But don't worry, you can add more and more complexities to our traffic pattern so eventually you will see something close to what you would see in real life.

Next, let’s add more flows in our testbed and check out the results.

Multiple Flows

Now we are sending 64 flows from PROX gen VM. To confirm the flow numbers let’s use the `flow -s` command in `contrail-tools`:

```
[root@a7s3 ~]# contrail-tools flow -s
Flow Statistics
---------------
    Total  Entries  --- Total =     132, new =       0
    Active Entries  --- Total =     132, new =       0
    Hold   Entries  --- Total =       0, new =       0
    Fwd flow Entries  - Total =     132
    drop flow Entries - Total =       0
    NAT flow Entries  - Total =       0

    Rate of change of Active Entries
    --------------------------------
        current rate      =        0
        Avg setup rate    =        0
        Avg teardown rate =        0
    Rate of change of Flow Entries
    ------------------------------
        current rate      =        0
```

You can see 132 flows entries meaning 66 groups of flows in our test. The additional two groups of flows are the control flows between the jump VM and gen VM. Good, let’s collect the traffic statistics:

```
[root@a7s3 ~]# contrail-tools vif --clear

Vif stats cleared successfully on all cores for all interfaces

[root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 2
| VIF 3 | Core 1 | TX pps: 1     | RX pps: 85248 | TX bps: 448       | RX bps: 84566016
| VIF 3 | Core 2 | TX pps: 1     | RX pps: 1     | TX bps: 336       | RX bps: 560
| VIF 0 | Core 1 | TX pps: 85842 | RX pps: 15    | TX bps: 119490528 | RX bps: 14744
| VIF 0 | Core 2 | TX pps: 0     | RX pps: 0     | TX bps: 0         | RX bps: 0
------------------------------------------------------------------------
|                                pps per Core                          |
------------------------------------------------------------------------
|Core 1  |TX + RX pps: 171133    | TX pps 85858     | RX pps 85275     |
|Core 2  |TX + RX pps: 2         | TX pps 1         | RX pps 1         |
------------------------------------------------------------------------
|Total   |TX + RX pps: 171135    | TX pps 85859     | RX pps 85276     |
------------------------------------------------------------------------
```

Still, the `lcore#10` processed the packets and forwarded them out of vif0/0. If you compare this result with our first test, where we have just one uni-directional flow, there is simply no difference. Shouldn’t we expect to see some load balance between lcores since we have more flows now? We should, but only when the VM tap interface has multiple queues. With just one queue, the mapping between our tap interface and lcores never changes. In this case it's always lcore#11 polling the traffic and distributing to `lcore#10`, hence we'll always see packet being forwarded by `lcore#10` instead of `lcore#11`, regardless of number of flows and traffic volumes.

On the other direction, if we enable the returning traffic, we'll see on VIF0 (vif0/0) the two lcores' traffic are `RX pps: 41547`, and `RX pps: 44257`, which is well balanced because we have two queues enabled on the vif0/0:

```
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 2
| VIF 3 | Core 1 | TX pps: 41249 | RX pps: 85182 | TX bps: 40919336  | RX bps: 84500544
| VIF 3 | Core 2 | TX pps: 43936 | RX pps: 1     | TX bps: 43584072  | RX bps: 336
| VIF 0 | Core 1 | TX pps: 85765 | RX pps: 41547 | TX bps: 119382912 | RX bps: 57825008
| VIF 0 | Core 2 | TX pps: 3     | RX pps: 44257 | TX bps: 18216     | RX bps: 61604304
------------------------------------------------------------------------
|                                pps per Core                          |
------------------------------------------------------------------------
|Core 1  |TX + RX pps: 253763    | TX pps 127025    | RX pps 126738    |
|Core 2  |TX + RX pps: 88197     | TX pps 43939     | RX pps 44258     |
------------------------------------------------------------------------
|Total   |TX + RX pps: 341960    | TX pps 170964    | RX pps 170996    |
------------------------------------------------------------------------
```

With a single queue in the VM tap interface, it's hard to achieve good load balance between lcores on the vRouter interface facing the VM. Sometimes you need to enable multiple queues to make better use of all your DPDK forwarding lcores.

This concludes our analysis on one single queue test, and we'll go ahead to test multiple queues.

Multiple Queues

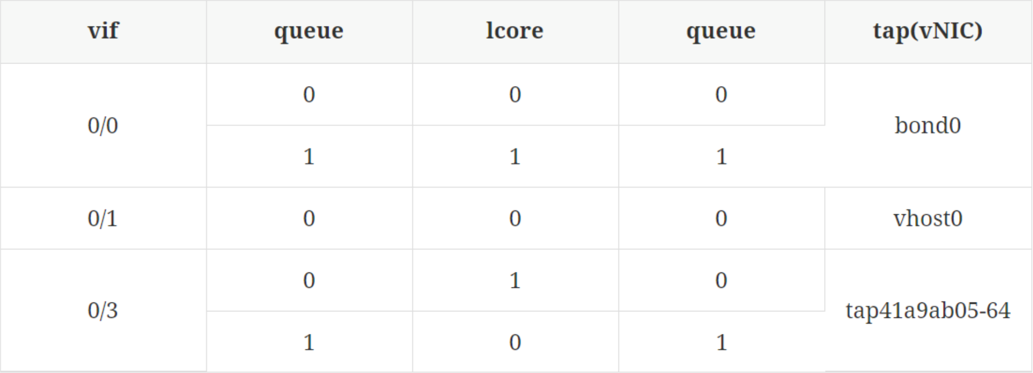
Finally, let's look at a multiple queue example. Based on the previous setup, this time we added one more queue in the tap interface of VM gen and then collect the core interface mapping (see Figure 6.7):

```
[root@a7s3 ~]# contrail-tools dpdkinfo -c
No. of forwarding lcores: 2
No. of interfaces: 5
Lcore 0:
        Interface: bond0.101           Queue ID: 0
        Interface: vhost0              Queue ID: 0
        Interface: tap41a9ab05-64      Queue ID: 1

Lcore 1:
        Interface: bond0.101           Queue ID: 1
        Interface: tap41a9ab05-64      Queue ID: 0
```

So most items remain the same, except we have one more queue added on tap interface and the vRouter interface to which it attaches. Correspondingly, one core is allocated to serve this new queue. Before this new queue was created, we already knew that each of our lcores was serving the same amount of queues; therefore, as a tie breaker, which we've mentioned when we introduced `dpdkinfo -c` previously, the first forwarding lcore, `lcore#10` with our notation, is allocated for the new queue.

Figure 6.7 Table View of these Mappings



vif	queue	lcore	queue	tap(vNIC)
0/0	0	0	0	bond0
	1	1	1	
0/1	0	0	0	vhost0
0/3	0	1	0	tap41a9ab05-64
	1	0	1	

Let's check the traffic distribution between lcores with multiple queues on VM tap interface:

```
[root@a7s3 ~]# contrail-tools dpdkvifstats.py -all -c 2
| VIF 3 | Core 1 | TX pps: 41319 | RX pps: 42606 | TX bps: 40988672 | RX bps: 42264712
| VIF 3 | Core 2 | TX pps: 43889 | RX pps: 42604 | TX bps: 43537008 | RX bps: 42262288
| VIF 0 | Core 1 | TX pps: 42923 | RX pps: 41540 | TX bps: 59748824 | RX bps: 57815160
| VIF 0 | Core 2 | TX pps: 42918 | RX pps: 44320 | TX bps: 59741640 | RX bps: 61693328
---------------------------------------------------------------------------
|                                 pps per Core                            |
---------------------------------------------------------------------------
|Core 1  |TX + RX pps: 168416    | TX pps 84258      | RX pps 84158      |
|Core 2  |TX + RX pps: 173731    | TX pps 86807      | RX pps 86924      |
---------------------------------------------------------------------------
|Total   |TX + RX pps: 342147    | TX pps 171065     | RX pps 171082     |
---------------------------------------------------------------------------
```

Now we have multiple queues on both the VM tap interface and the fabric interface. Traffic on all lcores is well balanced. Please keep this in mind as an ideal traffic profile that we expect the vRouter to have. In production, we usually deal with more complicated vRouter lcore configurations and traffic profiles, so the lcore balancing may not appear as perfect as what we are seeing in lab environment, but at least you have a good baseline in mind and know what to look for when the result is far worse than expected.