SAMPLE CHAPTER

OpenStack IN ACTION

V. K. Cody Bumgardner

FOREWORD BY Jay Pipes

MANNING

OpenStack in Action

by V. K. Cody Bumgardner

Chapter 4

brief contents

4 Understanding private cloud building blocks

This chapter covers

- Understanding OpenStack core project interoperability
- Exploring the relationship between vendor hardware and OpenStack
- Learning from a manual OpenStack install

In chapter 1 you were introduced to OpenStack. You learned how OpenStack fits into the cloud ecosystem, reasons for adopting the technology, and the focus of this book. In chapter 2 you went from those high-level concepts directly into a hands-on test-drive of the OpenStack framework using DevStack. In chapter 3 you worked through some examples that you might encounter working as an OpenStack operator and gained further insight into the structure of the framework.

In this chapter, we'll shift back to the high-level concepts. If the first chapter was to introduce and inform you, the second to get you excited about the technology, and the third to make you operationally comfortable, the fourth gives you a foundational understanding of what's really going on inside the OpenStack framework.

This chapter won't be as thought-provoking as chapter 1 or as fun as chapters 2 and 3. But regardless of whether you're a system administrator, developer, IT architect, or even a CTO, this is the *most important* chapter for your understanding of the OpenStack framework. If you'll be dealing with OpenStack in the trenches, this chapter will build your OpenStack foundation, which will be deepened in chapters 5 through 8. If you'll be working with OpenStack on a high level, even if you're simply responsible for a vendor-managed solution, this chapter will help you understand the collection of interacting components that make up an OpenStack deployment.

What are you waiting for? Let's get started!

4.1 How are OpenStack components related?

Since the first public release of OpenStack in 2010, the framework has grown from a few core components to nearly ten. There are now hundreds of OpenStack-related projects, each with various levels of interoperability. These projects range from OpenStack library dependencies to projects where the OpenStack framework is the dependency.

In an effort to provide structure around the diverse set of projects, the OpenStack Foundation created several project designations, including core, incubated, library, gating, supporting, and related. These project designations and their descriptions can be found in table 4.1.

Table 4.1 Project designations

Project designation	Description
Core	Official OpenStack projects (most people use these)
Incubated	Core projects in development (on track to become core)
Library	Dependencies of core projects
Gating	Integration test suites and deployment tools
Supporting	Documentation and development infrastructure
Related	Unofficial projects (self-associated projects)

Incubated projects, once fully developed and accepted, will eventually function in the same way core projects do. Library functions will be abstracted (not observable) by core project interaction. Gating and supporting projects aren't used to provide resources in a deployed system, so you don't need to worry about those. That leaves the related projects, which as the name implies, have some affiliation with OpenStack, even if the affiliation is self-nominated.

4.1.1 Understanding component communication

Often when someone refers to "OpenStack," they're referring to projects with a "core" designation. Core projects can use the OpenStack trademark and must pass all "must-pass" tests, as defined by the OpenStack Foundation. Simply put, core components

are those that almost everyone will use in an OpenStack deployment. Projects like Compute, Networking, Storage, shared services, and Dashboard are examples of projects with a core designation, as shown in table 4.2.

Table 4.2 Core projects

Project	Codename	Description
Compute	Nova	Manages virtual machine (VM) resources, including CPU, memory, disk, and network interfaces
Networking	Neutron	Provides resources used by VM network interfaces, including IP addressing, routing, and software-defined networking (SDN)
Object Storage	Swift	Provides object-level storage accessible via RESTful APIs
Block Storage	Cinder	Provides block-level (traditional disk) storage to VMs
Identity Service (shared service)	Keystone	Manages role-based access control (RBAC) for OpenStack components; provides authorization services
Image Service (shared service)	Glance	Manages VM disk images; provides image delivery to VMs and snapshot (backup) services
Telemetry Service (shared service)	Ceilometer	Centralized collection for metering and monitoring OpenStack components
Orchestration Service (shared service)	Heat	Template-based cloud application orchestration for OpenStack environments
Database Service (shared service)	Trove	Provides users with relational and non-relational database services
Dashboard	Horizon	Provides a web-based GUI for working with OpenStack

In addition to various project designations, there are also several topologies in which you can deploy project components. If more of a specific resource (Storage, Compute, Networking, and so on) is required, more component-specific servers can be added. We'll discuss the project designations and their related components in section 4.1.2.

DASHBOARD AUTHENTICATION COMMUNICATION

Let's jump right in and take a look at how core components communicate. We'll walk through the process of accessing the OpenStack Dashboard, reviewing the VM creation options, and creating a virtual machine.

You must first provide the Dashboard with your login credentials and obtain an authentication token. The authentication token is saved as a cookie in your web browser and used with subsequent requests. As shown in figure 4.1, you obtain an authentication token from the Identity Service. While you can use the Dashboard (instead of the CLI or APIs) to navigate through the rest of this example, we won't show the interaction with the Dashboard. Even during the login process, the Dashboard just displays interactions between the web browser and the OpenStack APIs. We're primarily concerned with component interaction on the API level.

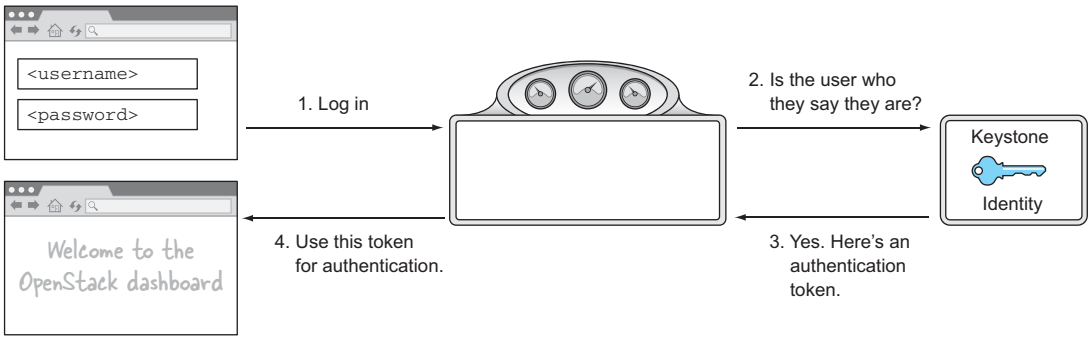

Figure 4.1 Dashboard login

Once you have your authentication token, you can take the second step and access the Compute component to create your virtual machine (VM).

RESOURCE QUERY AND REQUEST COMMUNICATION

As explained in chapter 3, OpenStack works on a tenant model. If the OpenStack deployment is a hotel of resources, you can think of tenants as rooms in the hotel. Each tenant (room) is assigned a resource quota (a number of towels, beds, and so on). OpenStack users (guests) are assigned to tenants (rooms) through the use of roles. The identity information is kept by the Identity component, and the quota information is maintained by the Compute component.

In the Dashboard, when you click Launch Instance, the Compute component is queried to determine what resources and configurations are available in your current tenant. Based on the available options, you describe the VM you want and submit the configuration for creation.

The communication between components during a VM creation request is shown in figure 4.2. Because the creation of a VM isn't instantaneous, the process is executed asynchronously, so after you submit a VM for provisioning, you're returned to the Dashboard. In the Dashboard, your browser will periodically update the VM status information.

RESOURCE PROVISIONING COMMUNICATION

When VM creation requests are submitted, the Compute service component will interact with other components to provision the requested VM. The first thing that happens is that the VM object record is registered with the Compute service component. This object record contains information about the VM status and configuration—the VM object isn't the VM instance, only a record describing the instance.

When components communicate in the VM creation process, they reference common objects, like this VM object. For instance, the Compute service component might request a storage assignment from the Storage service component. The Storage service component would then provision the requested storage and provide a reference to a `Storage` object, which would then be referenced in the VM object record.

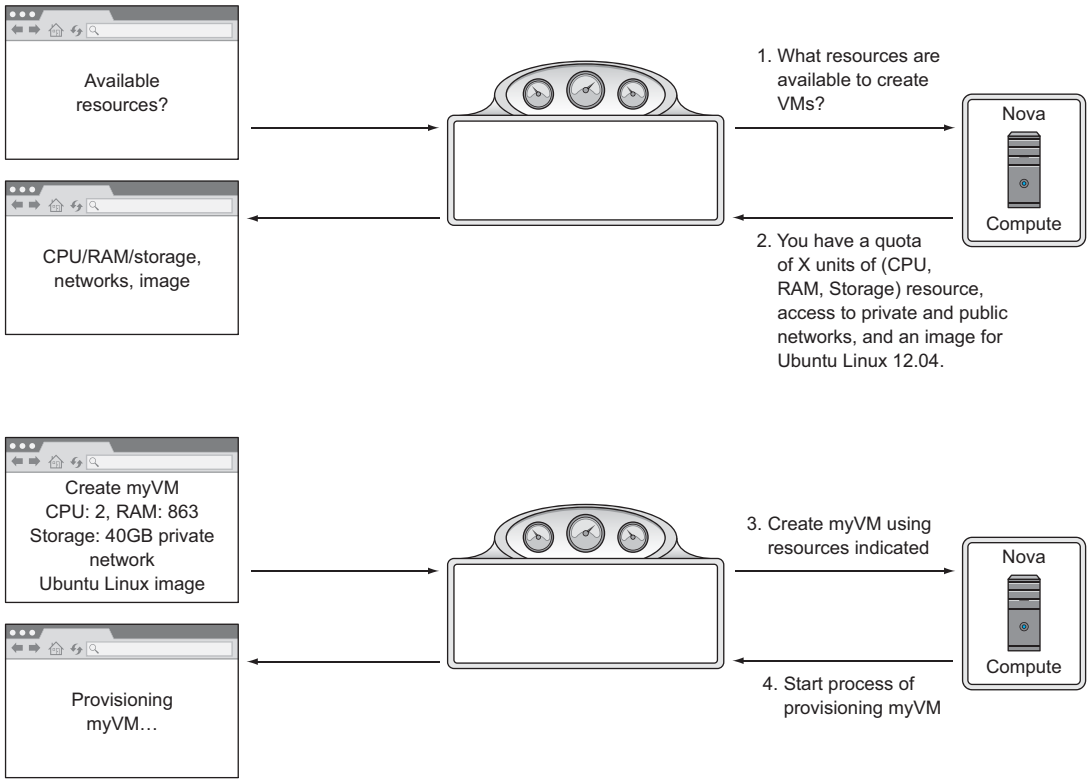

Figure 4.2 Resource query and request

As shown in figure 4.3, the Compute service component communicates with other core components to provision and assign resources to the VM object. Compute will first request infrastructure components like Storage and Networking. When the virtual infrastructure has been assigned to the VM and referenced in the VM object, the Image Service will prepare the virtual storage volume with the requested image or snapshot. At this point the VM creation process is complete and the Compute component can spawn the VM.

As demonstrated in the previous figures, core components work in concert to provide OpenStack services. OpenStack interactions, even those in the Dashboard, eventually find their way to the OpenStack APIs.

As you'll see next, related projects often use API calls exclusively to interact with OpenStack.

RELATED PROJECT COMMUNICATION

Let's take a look at how Ubuntu Juju, a related project, interacts with OpenStack. Juju is a cloud automation package that uses OpenStack for virtual infrastructure. Juju automates the deployment and configuration of applications on virtual infrastructure using application-specific charms.

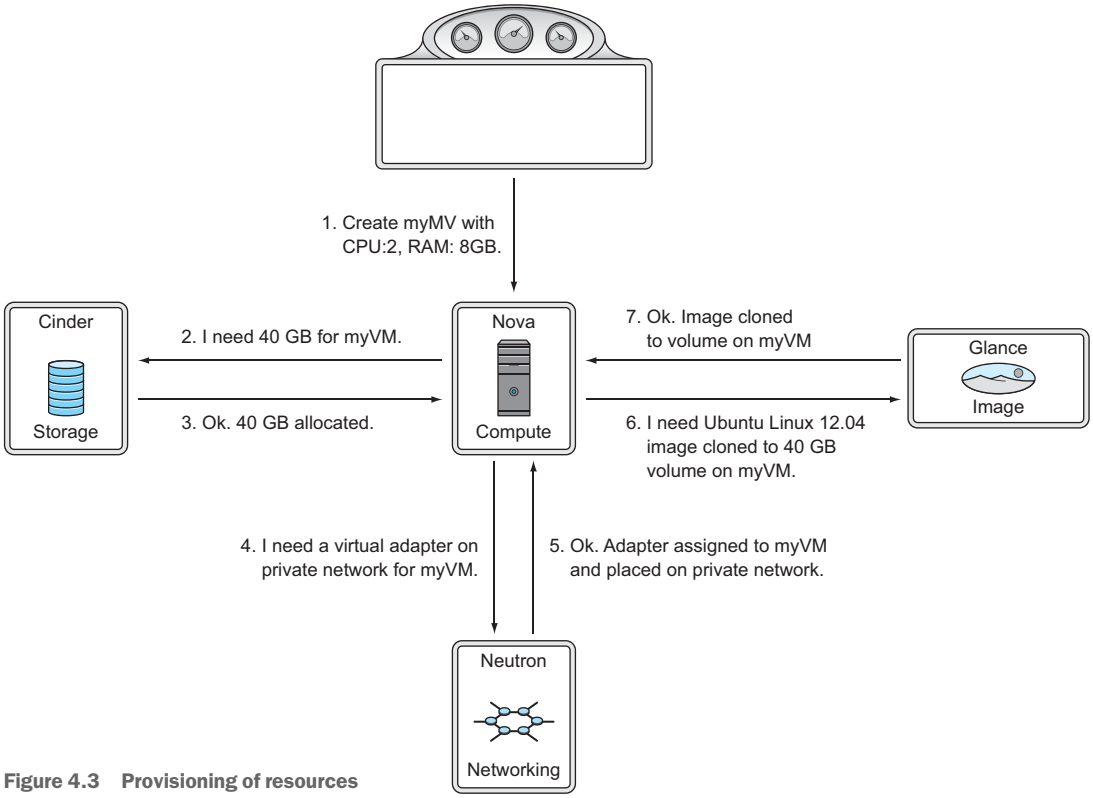

Figure 4.3 Provisioning of resources

For the lack of a better description, Juju *charms* are collections of installation scripts that define how services and applications integrate into virtual infrastructure. Because infrastructure, including networks and storage, can be provisioned programmatically using OpenStack, Juju can deploy entire application suites from a charm. Simply put, Juju turns newly provisioned VM instances into applications. We discuss this process in detail in later chapters, but essentially you tell an application charm how large you want your instances to be and how many instances you want, and it does the work to deploy your applications.

The first step in using Juju in your OpenStack deployment is to deploy what Juju calls its *bootstrap*, using the Juju CLI. The bootstrap is a VM that Juju uses to control its automation processes. The deployment of the bootstrap, from a component perspective, is similar to what you've seen in recent figures (see figures 4.1, 4.2, and 4.3). The difference here is that in place of the web browser making the request, it's the Juju application.

JUJU NODES FROM THE OPENSTACK PERSPECTIVE Juju nodes run the Ubuntu Linux operating system and include Juju-specific automation tools. From the OpenStack perspective, a Juju node is no different than any other VM provided by OpenStack. As a related project, Juju makes use of resources provided by OpenStack, but that's where the integration ends.

Once the bootstrap node has been started, Juju commands will be issued to the bootstrap node, not directly to OpenStack APIs. The reason for this is that the provisioning process is asynchronous, as mentioned earlier, and it's sometimes time-consuming. You don't want to maintain a connection from the desktop to the OpenStack deployment while a 20-VM application is deployed.

In chapter 12 you'll walk through deploying WordPress using Juju as an orchestration tool and OpenStack as the back end. Let's take a look at how Juju uses the bootstrap VM to orchestrate application deployment. Consider an example where you use Juju and OpenStack to deploy a load-balanced WordPress application with a clustered MySQL back end. In this case, you have three types of service nodes: load-balancing, WordPress (Apache/PHP), and MySQL DB. Using the Juju charm developed for WordPress, you describe the number of nodes for each service, their virtual size (CPU, RAM, and so on), and how the nodes relate. You submit this charm to your bootstrap node, which then interacts with OpenStack to provision the application. This process is shown in figure 4.4.

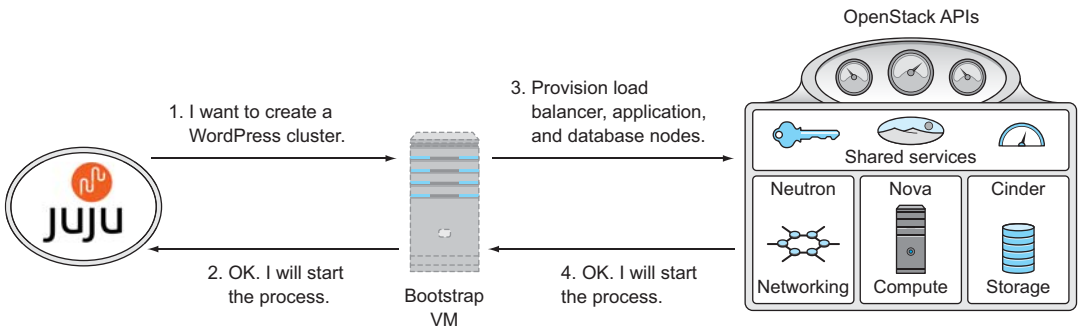

Figure 4.4 OpenStack interacting with a related project

Let's assume that OpenStack, through the direction of the bootstrap node, successfully provisions all the required virtual infrastructure. At this point you have a collection of VMs, but no applications. The bootstrap node polls OpenStack, watching for its requested VMs to come online. Once the VMs are online, it will start a process outside the OpenStack framework to complete the install. As shown in figure 4.5, the bootstrap node will communicate directly with the newly provisioned VMs. From this point forward, OpenStack simply provides the virtual infrastructure and is unaware of the application roles assigned to each VM.

We've now discussed how the components of OpenStack communicate on the logical level. In the figures, we've illustrated component communication, as if everything was communicating inside a single big node (physical instance). In practice, however, OpenStack components will be distributed across many physical commodity servers in a multi-node topology.

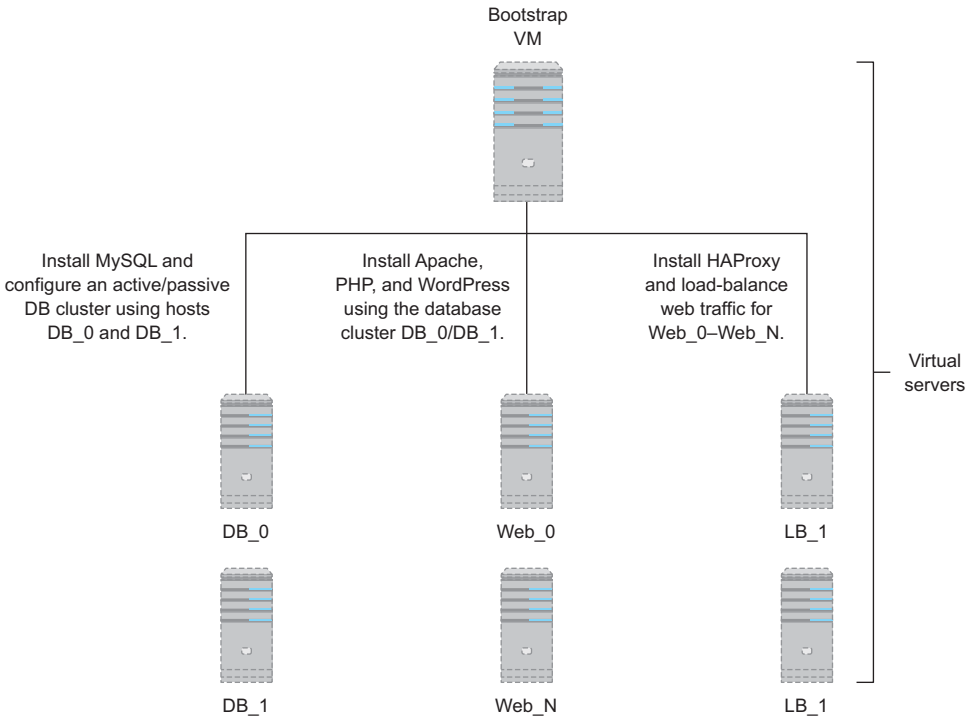

Figure 4.5 Juju bootstrap controls the VMs

4.1.2 Distributed computing model

Let's take a look at the OpenStack component distribution model. In distributed computing, there are several component distribution methods.

In a *mesh* distribution, control and data are distributed on the node level, and no central authority exists. This method is fully distributed, but maintaining concurrency across nodes is more difficult than in a central-control model. Mesh distributions are most often used when workloads are self-contained and require little coordination beyond collecting results.

On the other end of the spectrum, a *hub-and-spoke* distribution passes all control and data through a central node, like spokes around a hub. Hub-and-spoke topologies are generally limited in scale, due to the aggregation of both the control and data plane to a central node. Hub-and-spoke is most often used for workloads with a high degree of node-to-node communication and coordination.

The OpenStack distribution model shares characteristics of both mesh and hub-and-spoke distributions. Like mesh, once OpenStack provisions the virtual infrastructure, the infrastructure will continue to function without the involvement of a central controller. But like hub-and-spoke, component interaction is coordinated through a central API service. The node that maintains the API services is known as the OpenStack

controller. The controller coordinates component requests and serves as the primary interface for an OpenStack deployment.

GENERAL DISTRIBUTED COMPONENT MODEL

Briefly, let's suspend our thinking around the idea of OpenStack components, and focus on the hybrid mesh and hub-and-spoke distribution model implemented by OpenStack. Figure 4.6 illustrates the interaction of nodes in the OpenStack distribution model. The client contacts the controller to make service requests. The controller, while not an operational dependency of the nodes, is aware of the system-wide status and inventory. The controller selects the appropriate nodes for the job and distributes the request.

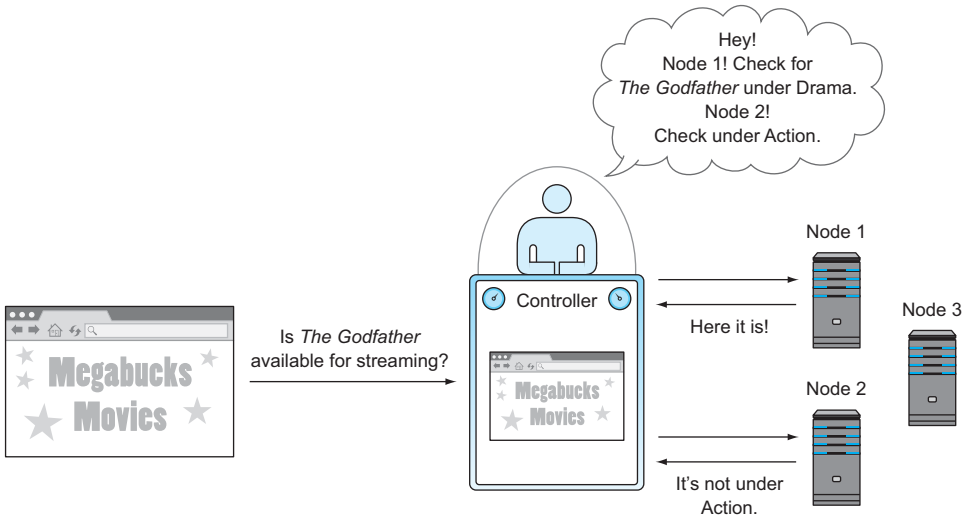

Figure 4.6 Distributed component model

OPENSTACK'S DISTRIBUTED COMPONENT MODEL

The general distributed component model presented in figure 4.6 is representative of the way OpenStack components communicate. Let's discuss one final abstract example of this model before we look at OpenStack specifics. Suppose a distributed component model, like the one shown in figure 4.6, was implemented in a content management system, like the ones used to stream movies on demand. Consider two movies streaming simultaneously to two users. The initial requests to stream a movie were made from the clients to a controller, and the controller directed two nodes to stream the two movies to the clients. Now, suppose that while the movies are streaming, the controller experiences a catastrophic failure. The movie streams wouldn't be interrupted, and neither the clients nor the nodes would be aware of this event. In this type of distributed model, new requests can't be fulfilled until a controller is available, but existing operations will continue.

Now let's think about how OpenStack components behave. This time we'll think about components in relation to the OpenStack distribution model. The control portion of the component will reside on the control node, and the provisioning components will be distributed on the resource nodes. Figure 4.7 introduces OpenStack components into the distributed model.

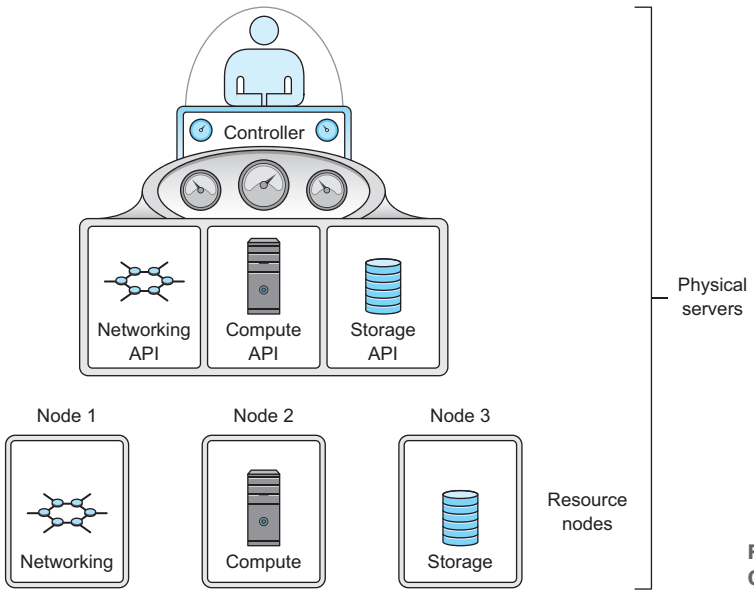

Figure 4.7 Distributed OpenStack model

Distributed component interaction in VM provisioning

In the OpenStack distributed model, many resource nodes can exist for a single controller. OpenStack components are actually collections of services. As previously stated, some services run on controller nodes and some on resource nodes. Depending on the component, there might be several services that run on the controller and several more on resource nodes. For the Compute component alone, there are six controller services. In comparison, the Compute resource nodes generally run a single Compute component.

Let's take a look at what happens when a VM request is made. Figure 4.8 illustrates the node-level interaction of distributed OpenStack components required to create a VM. From a component perspective, nothing has changed from the previous figures. What we want to demonstrate is how OpenStack components communicate when components are distributed on multiple nodes.

VM-level component communication

In a multi-node deployment, you'll have multiple nodes for each primary node type (compute, storage, network). The ratio of compute, network, and storage nodes will be dependent on your requirements for these resources. Specific node types might

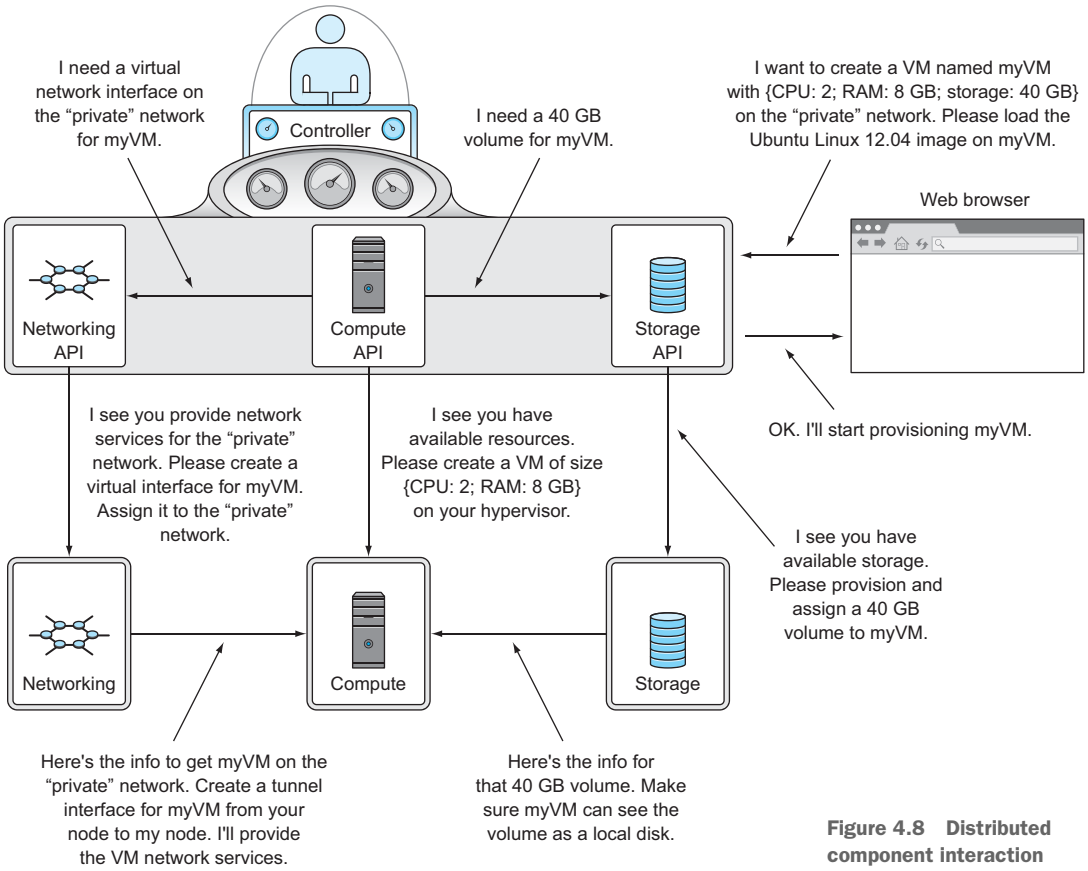

Figure 4.8 Distributed component interaction

additionally be connected to other vendor components, such as storage nodes to vendor storage systems and network nodes to vendor network devices. The way specific vendor resources are used by OpenStack is explained in section 4.2.

We've described OpenStack component relations from the component communication level and the distributed services level. Now we'll take a look at what's going on from the perspective of the VM.

Virtual machines, as the name implies, are virtualized representations of resources that would be available on a single physical machine. A VM runs an operating system (OS), just like a physical system, and any OS running on a general-purpose VM will expect virtual hardware to behave exactly like physical hardware resources. This is to say, the OS reads and writes to network and storage devices the same way it writes to CPU registers or RAM. When a physical machine runs a hypervisor, the hypervisor does the work of translating multiple virtual address spaces to a single physical address space. In a distributed OpenStack component, not only do you have virtual resources, but they're also distributed on separate physical nodes. You need to understand how the distribution of resources relates to what is seen by the VMs.

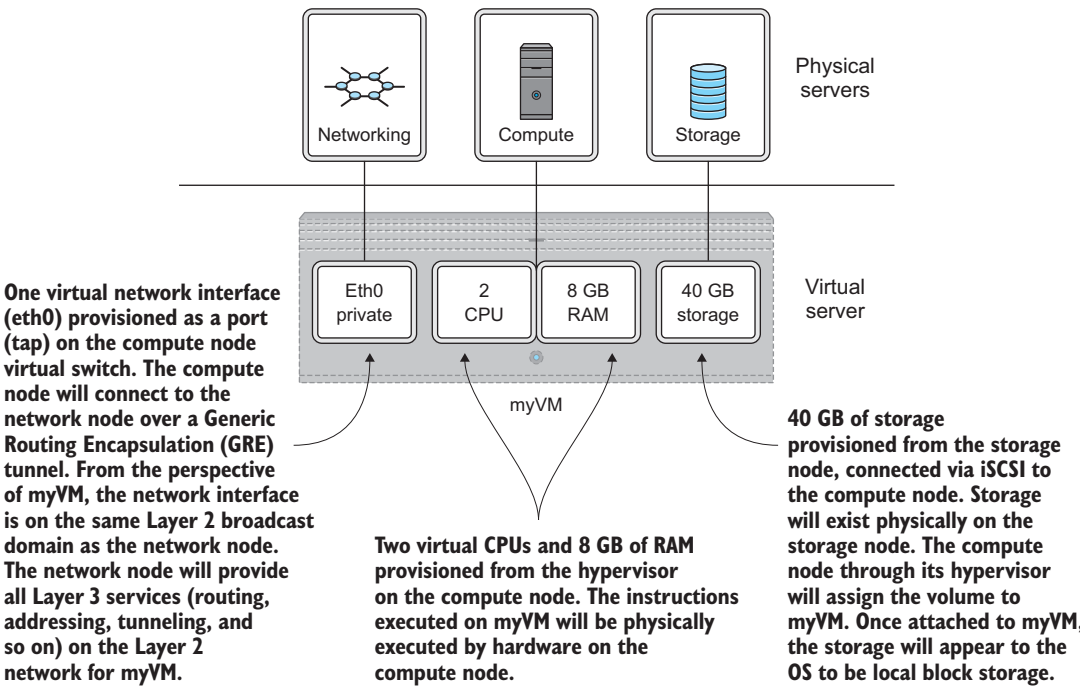

Figure 4.9 Component-VM relations

Although VM resources are provided by multiple component-specific nodes, from the perspective of the VM all resources are provided by a single piece of hardware. Figure 4.9 illustrates how resources from component-specific resource nodes are combined to create a single VM.

You can think of the VM as *living* on a specific compute node, but the actual data will live on a storage node, and data communicated (Layer 3) by the VM lives (passes through) the network node.

> **DISTRIBUTED VIRTUAL ROUTING (DVR)** Until recent releases of OpenStack, L3 network functions like routing were typically performed by a small number of dedicated network nodes. The Neutron/DVR subproject has emerged to manage the distribution of routing across compute and dedicated network nodes.

The OpenStack distributed architecture and component design allows for very efficient deployment of virtual infrastructure. The OpenStack framework provides you with the ability to manage many nodes across component-node types from a single system.

4.2 How is OpenStack related to vendor technologies?

For many years, the vendors that provided compute, storage, and network hardware focused on marketing faster and more capable hardware. More recently, though, hardware has been viewed as a commodity, software has become more interoperable,

and vendors have begun to provide services such as cloud computing instead of just hardware and software, offering consumers much more flexible choices.

One of the greatest benefits provided by the OpenStack framework is vendor neutrality. By interfacing with the OpenStack APIs, you are assured a minimum level of functionality regardless of the underlying hardware vendor you're using. OpenStack doesn't free you from vendors altogether—you still need underlying servers, storage, and network resources. But OpenStack allows you to make vendor choices based on performance and price without taking into account sunk costs on vendor-specific implementations and the lock-in of feature sets. Not only can you use existing hardware and software with OpenStack, future purchases can be based on what OpenStack provides, not vendor-specific features.

In this section, we'll discuss how OpenStack deals with vendor-specific integrations. The term *vendor* is used loosely in this context and can refer to either open source technologies or commercial products. In OpenStack it's up to the vendor or support community to develop the vendor-technology integration. Different OpenStack components have different ways of dealing with this integration, as you'll see in the following sections.

4.2.1 Using vendor storage systems with OpenStack

Let's look at the types of vendor storage supported by OpenStack Block Storage (Cinder) and see how this integration is achieved. Figure 4.10 shows a logical view of storage resource assignments and management.

This figure shows the CPU and RAM portions of a VM being provided by a commodity server. It also shows that the storage assigned to the VM is not on the commodity server; it's provided by a separate storage system. As you'll soon learn, there are many ways to provide that virtual block device to a VM.

> **STORAGE SYSTEM IN DEVSTACK** In chapter 2 you walked through deploying DevStack, but you didn't do any specific configuration for storage. In that single-node DevStack deployment, the storage resources were consumed from the same computer as the compute resources. However, in multi-node production deployments, compute and storage resources are isolated on specific storage and compute nodes and/or appliances.

The use of storage vendors and technologies isn't limited to OpenStack Block Storage (Cinder) and could even be used with OpenStack Object Storage (Swift). We'll look at Cinder because the storage it manages is used as part of a VM, and this chapter focuses on the integration between OpenStack and vendor components. This is not to say that OpenStack Object Storage is any less complex, just that it's more self-contained and isn't used directly by a running VM (and thus is less relevant in this chapter).

HOW STORAGE IS USED BY VMS

In OpenStack and other environments that provide infrastructure as a service (IaaS), virtual block storage devices are provisioned and assigned to VMs. The operating

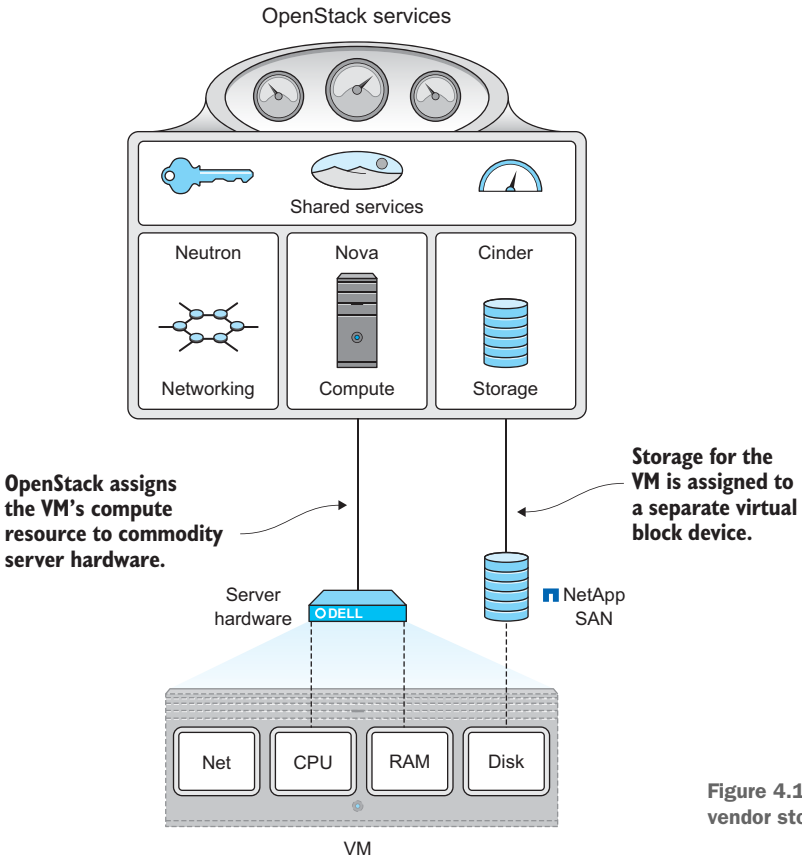

Figure 4.10 OpenStack and vendor storage system

systems running in the VMs manage the filesystems on their virtual block devices or volumes.

You might be wondering, "If the compute portion of a VM is provided by a server and the storage is provided by a separate server or storage appliance, how are they connected to provide a single VM?" The simple answer is that all resources eventually make their way to the VM as virtual hardware, which is then connected together on the hypervisor level. Take a look at figure 4.11, which shows a technical view of the logical view shown previously in figure 4.10.

In this figure, a vendor storage system is directly connected to a compute node (connected through Peripheral Component Interconnect Express (PCI-E), Ethernet, Fiber Channel (FC), Fiber Channel over Ethernet (FCoE), or vendor-specific communication link). The compute node and the storage system communicate using a *storage transport protocol* such as Internet Small Computer System Interface (iSCSI), Network File System (NFS), or a vendor-specific protocol. In short, storage can be provided to the compute node running the hypervisor using many different methods, and it's the compute node's job to present those resources to the virtual machine.

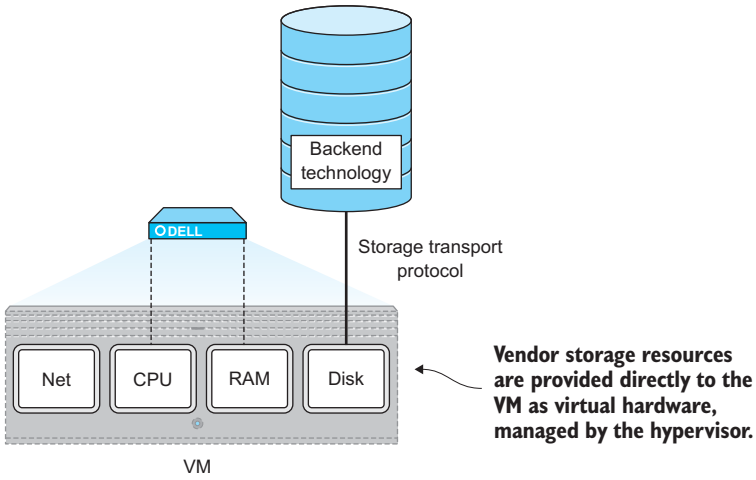

Figure 4.11 Vendor storage used by hypervisor

Looking once again at figure 4.11, you can see that regardless of how the storage is provided, the storage resources assigned to a specific VM end up managed as virtual hardware on the same node that provides CPU and RAM resources to that VM.

Let's summarize what you've learned so far about OpenStack and vendor storage systems:

- Operating systems use block storage devices for their filesystems.
- Hypervisors on compute nodes provide virtual block (OS-bootable) devices to VMs.
- There are many ways to provide storage resources to a compute node running a hypervisor.
- Vendor storage systems can be used to provide storage resources to compute nodes.
- OpenStack manages the relationship between the hypervisor, the compute node, and the storage system.

In the next section you'll learn just how OpenStack manages these resources.

How OpenStack supports vendor storage

You might be thinking, "OK, I understand how the storage is used, but how is it managed by OpenStack?" Cinder is a modular system allowing developers to create plug-ins (drivers) to support any storage technology and vendor. These modules might be developed by a product development team in a corporation or a community effort.

Figure 4.12 shows Cinder using a plug-in to manage a vendor storage system.

As you learned earlier in this chapter, individual OpenStack components have specific responsibilities. In the case of Cinder, its responsibility is to translate the request for storage from OpenStack Compute into an actionable request using the vendor-specific API of a storage system.

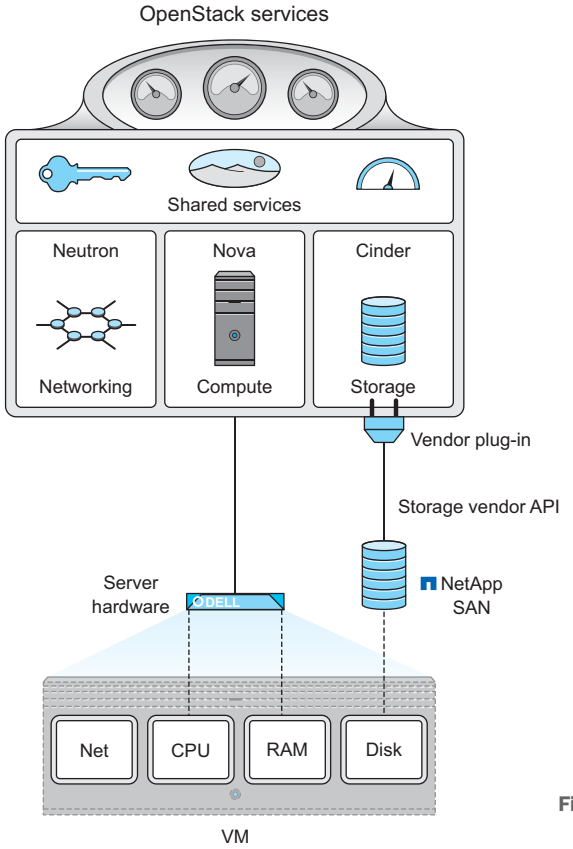

Figure 4.12 Cinder manages vendor storage.

Obviously, if you're going to translate one language or API to another, you need a minimum number of defined functions that can be related to each other. For each OpenStack release, there are a minimum number of required features and statistical reports for each plug-in. If plug-ins aren't maintained between releases, and additional functions and reports are required, they're deprecated in subsequent releases. The current lists of minimum features and reports (at the time of writing) are found in tables 4.3 and 4.4. The most current list of plug-in requirements can be found on the GitHub repository: http://docs.openstack.org/developer/cinder/devref/drivers.html. However, as of the time of this writing, the list of Cinder plug-in *minimum features* hasn't changed since the Icehouse release.

Table 4.3 Minimum features

Feature name	Description
Volume create/delete	Creates/deletes a volume for a VM on a backend storage system
Volume attach/detach	Attaches/detaches a volume to/from a VM on a backend storage system
Snapshot create/delete	Takes a running snapshot of a volume on a backend storage system

Table 4.3 Minimum features *(continued)*

Feature name	Description
Volume from snapshot	Creates a new volume from a previous snapshot on a backend storage system
Get volume stats	Reports the statistics on a specific volume
Image to volume	Copies image to a volume that can be used by a VM
Volume to image	Copies a volume used by a VM to a binary image
Clone volume	Clones one VM volume to another VM volume
Extend volume	Extends the size of a VM volume without destroying the data on the existing volume

Table 4.4 Minimum reporting statistics

Statistic name	Example	Description
driver_version	1.0a	Version of the vendor-specific driver for the reporting plug-in.
free_capacity_gb	1000	Amount of free space in gigabytes. If unknown or infinite, the keywords "unknown" or "infinite" are reported.
reserved_percentage	10	Percentage of space that is reserved but not yet used (thin provisioned volume allocation, not actual usage).
storage_protocol	iSCSI	Reports the storage protocol: iSCSI, FC, NFS, etc.
total_capacity_gb	102400	Amount of total capacity in gigabytes. If unknown or infinite, the keywords "unknown" or "infinite" are reported.
vendor_name	Dell	Name of the vendor that provides the backend storage system.
volume_backend_name	Equ_vol00	Name of the volume on the vendor backend. This is needed for statistical reporting and troubleshooting.

EXAMPLES OF VENDOR STORAGE IN OPENSTACK As previously stated, support for vendor storage is provided by plug-ins in Cinder. Plug-ins have already been developed by and for many vendors, including Coraid, Dell, EMC, GlusterFS, HDS, HP, Huawei, IBM, NetApp, Nexenta, Ceph, Scality, SolidFire, VMware, Microsoft, Zadara, and Oracle. In addition to commercial vendors, Cinder also supports storage provided by Linux Logical Volume Manager (LVM) and NFS mounts. An up-to-date Cinder support matrix can be found here: https://wiki.openstack.org/wiki/CinderSupportMatrix.

UNKNOWN OR INFINITE FREE SPACE In table 4.4, under *free_capacity_gb*, you'll notice that the values *unknown* and *infinite* can be used as free space values. Situations where these values are necessary might exist, but from a general operations perspective you should be aware that these are valid values for a storage driver.

4.2.2 Using vendor network systems with OpenStack

In OpenStack, it's common for compute resources to be provided by server hardware, storage resources by vendor storage systems, and networks by one or more vendors simultaneously. Obviously, if a VM is *running* on a specific server, that server is providing all of the computational resources (CPU, RAM, I/O, and so on) for that VM. Because a server can support more than one VM, this relationship is one-to-many from the perspective of the server and one-to-one from the perspective of the VM. That is to say that from a computation standpoint, the only resources consumed will come from the server hosting the VM.

As discussed in the previous section, although storage resources are technically removed from the compute node, from the perspective of the VM this is also a one-to-one relationship. In general, you'll have a single node running on a single volume that appears to the VM to be from a single container of virtual hardware.

Figure 4.13 shows the logical view of network resource assignments and management first introduced in chapter 1.

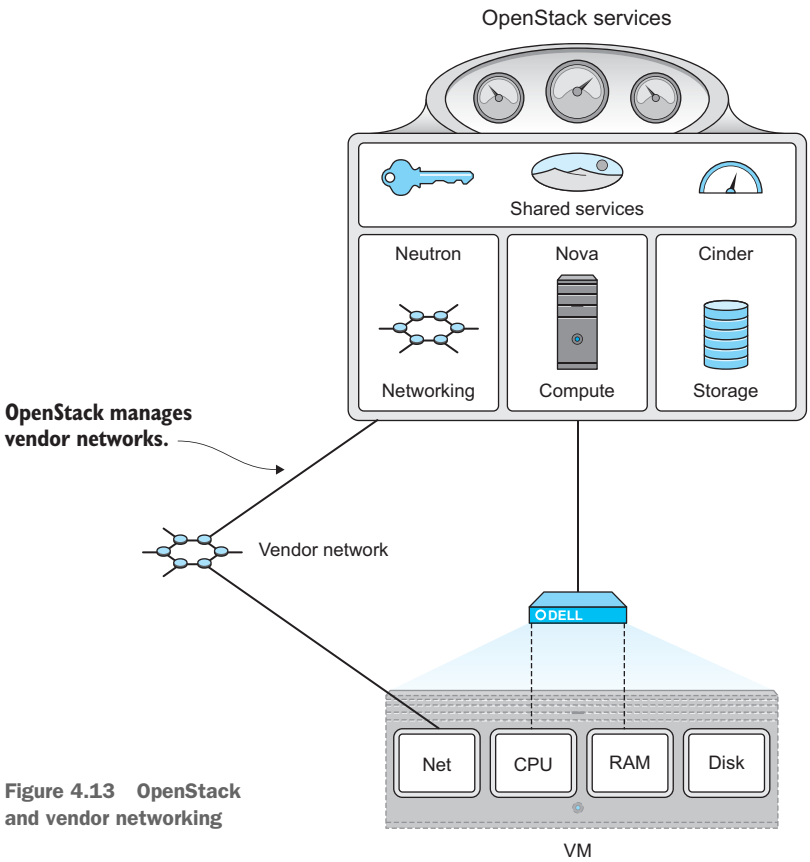

Figure 4.13 OpenStack and vendor networking

This figure represents a simplistic view of networking that suggests network resources are to be consumed in the same one-to-one way as compute and storage. Unfortunately, networking is not that simple. What the figure doesn't show are the layers of management that go into connecting two endpoints on a network. This section describes OpenStack Networking (Neutron) and how it manages vendor networks.

We'll look first at how VMs use networking.

How networking is used by VMs

Obviously, a network isn't very useful with a single VM, so you can expect that there will be, at a minimum, two VMs/nodes communicating. The way in which two nodes communicate depends on their relation to one another in the overall network. Table 4.5 summarizes several communication cases experienced in traditional virtual environments. These are described as traditional cases because software-defined networking (SDN), regardless of vendor, has blurred the lines of this paradigm.

Table 4.5 Node communication cases

Case	Description
Intra-host	Communication on the same VLAN (L2 network) on the same physical host
Inter-host-internal	Communication between nodes on the same VLAN, but different hosts
Inter-host-external	Communication between OpenStack hosts and endpoints on unknown external networks (internet)

In the intra-host case, traffic is kept on the physical host and never reaches the vendor network. The hypervisor can use its virtual switch (network) to pass traffic from one host to another.

In contrast, in both the inter-host-internal and inter-host-external cases, the hypervisor nodes and overall virtualization platform completely offload node communication to the vendor network.

Figure 4.14 shows the traditional method of communication for nodes on the same host. As of the time of writing, legacy Nova networking and the default distributed switch in VMware vSphere work this way.

The figure shows three nodes on the same physical host. The two nodes on VLAN_1 communicate inside the host and don't touch the vendor network. But communication between the two nodes on separate VLANs, VLAN_1 and VLAN_2, is offloaded to the vendor network. The vendor network is completely in charge of making sure this communication makes it to the intended destination, even when the endpoints are on the same node. The detail needed to cover how the networking works in these cases is beyond the scope of this chapter. What you need to understand is that OpenStack abstracts a great deal of complexity from the vendor network. Complex vendor-specific configurations are managed through plug-ins.

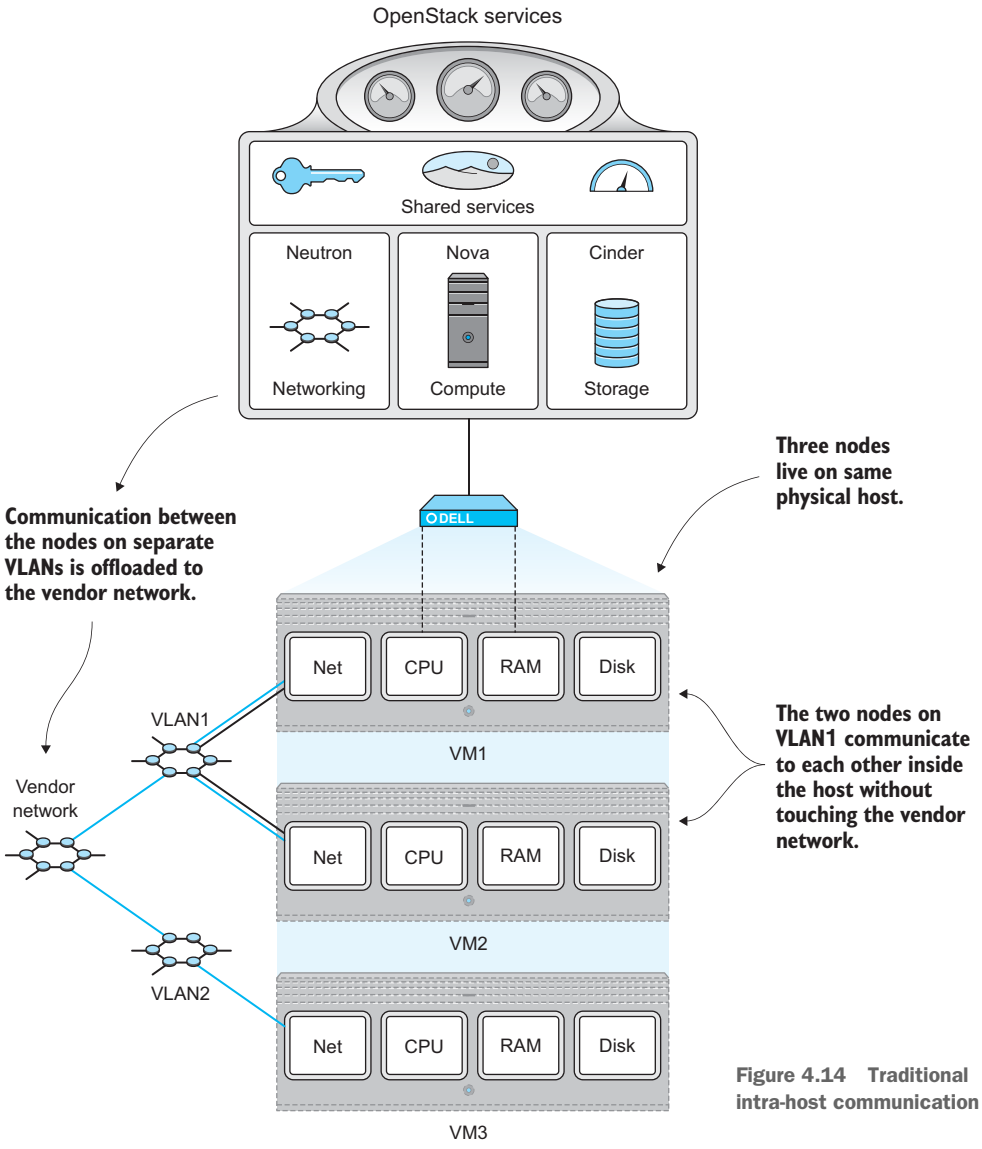

Figure 4.14 Traditional intra-host communication

By now it should be clear that vendor networking is more complicated than simply *provisioning* resources, which is what vendor storage systems do. Take a look at figure 4.15, which shows two hosts communicating using a vendor network. Of course, you could configure OpenStack to behave like a traditional virtualization framework and simply offload all the communication to the vendor network, but this is undesirable for a cloud platform. The details of why it's undesirable are beyond the scope of this chapter, but suffice it to say that this approach will not scale and will be a limiting factor in how you manage and provision resources.

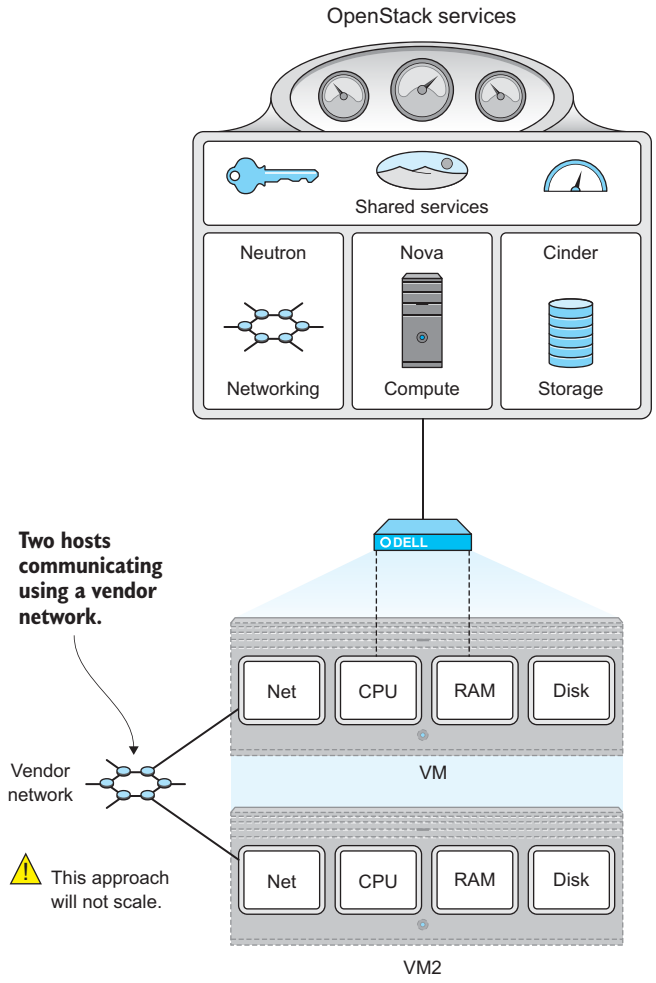

Figure 4.15 Vendor networking host-to-host

Suppose you want to manage the network in figure 4.16 with the same level of granularity you manage compute and storage resources. In this model, OpenStack Networking (Neutron) interfaces directly with vendor network components, which allows Neutron and its supported host to make their own network decisions.

Let's summarize what you've learned so far about OpenStack and vendor networking systems:

- Traditional hypervisors and virtualization frameworks unintelligently offload many functions to vendor networking.
- Traditional hypervisors and virtualization frameworks have little or no knowledge of how networking was performed, even for their own VMs.
- Managing vendor networking is more complicated than controlling a one-to-one relationship, like with vendor storage.
- Neutron is the codename for OpenStack Networking.

- Neutron integrates with vendor networking components to make networking decisions for OpenStack.

We'll take a look at how Neutron interfaces with vendor network components in the next section.

How OpenStack supports vendor networking

Just as Cinder uses vendor-specific plug-ins to communicate with vendor storage systems, Neutron uses plug-ins to manage vendor networking. As previously stated, plug-ins translate between OpenStack APIs and vendor-specific APIs. The relationship between Neutron and the vendor network is shown in figure 4.16.

You might wonder just what is being managed on the vendor network. The answer to this question is *it depends*. There are many networking vendors who produce many types of networking devices. These devices must interoperate at least on the level of network communication. After all, what good is a network if you can't communicate between networks and devices?

Software-defined networking (SDN) supports the idea of a separation of network management and communication functions. Because OpenStack Networking is a type

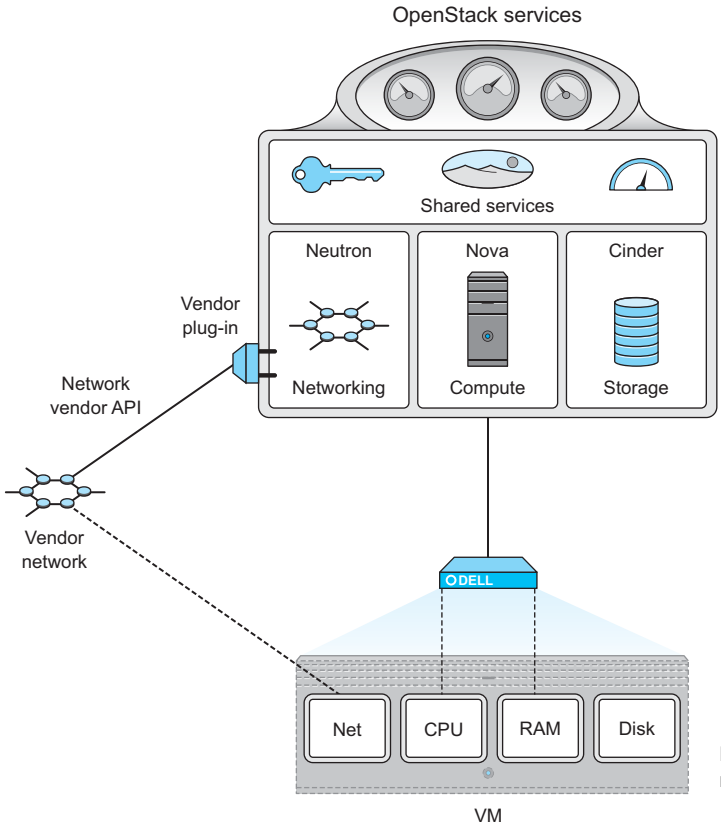

Figure 4.16 Neutron manages vendor networking.

of SDN, this so-called separation of the *control plane* and *data plane* is at the heart of OpenStack Networking when dealing with vendor hardware and software.

OPENSTACK NETWORKING ALSO PROVIDES L3 SERVICES In the context of vendor networking, OpenStack functions as a network controller. But it's worth noting that OpenStack networking does provide L3 services in the form of virtual routing, DHCP, and other services.

Figure 4.17 shows Neutron managing network devices in the control plane through the use of vendor-specific plug-ins. As you can see, the data plane never touches Neutron. In fact, Neutron might have no low-level insight into how the communication between the two nodes is happening. But Neutron knows that both nodes are on the specific network hardware that it manages, so Neutron can configure the endpoints to communicate, regardless of how the communication navigates the data plane.

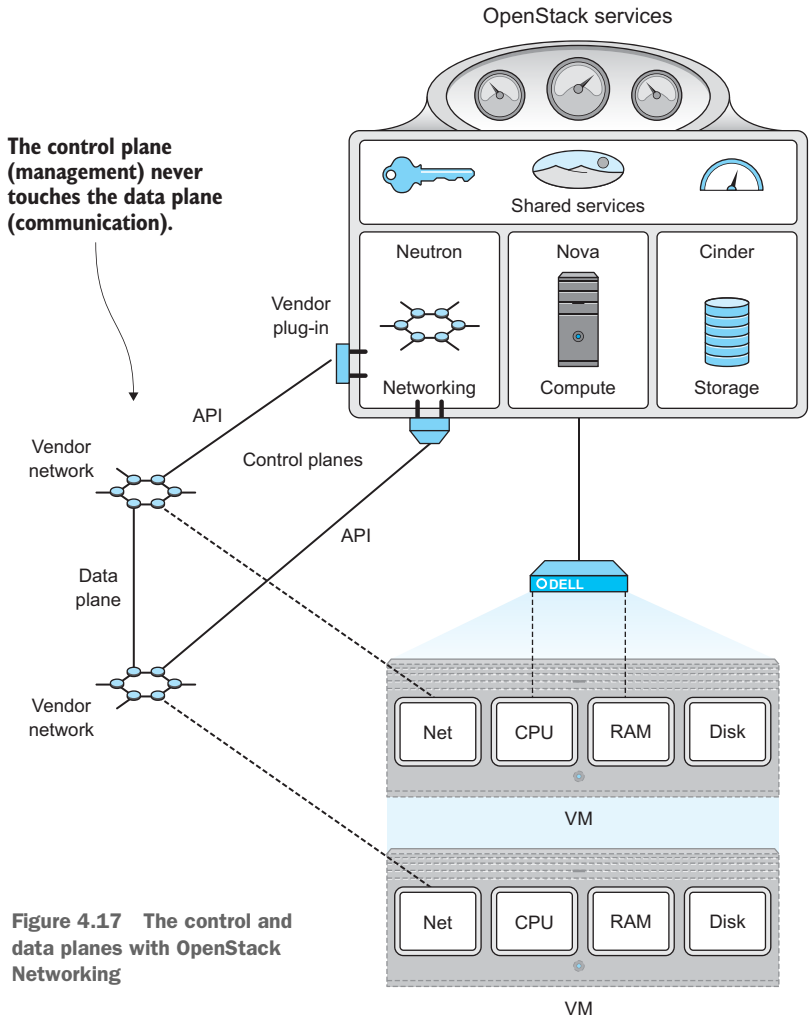

Figure 4.17 The control and data planes with OpenStack Networking

Understanding SDN and OpenStack Networking

This is a very complicated topic, and you'll probably want to go back and reread this section a few times. You aren't expected to fully understand SDN, but it's important that you understand the basic role of Neutron in relation to vendor networking. Chances are that your local network expert (unless this is you), doesn't know any more about SDN, and by relation OpenStack Networking, than you do. OpenStack/Neutron works on the control plane to manage communication between VMs that it manages, but it doesn't control the data plane related to communication between endpoints.

This is a new way of thinking about networking that really turns the traditional network world on its head. I've just introduced the topic in this section to give you some insight into how OpenStack can manage vendor networks without you having to hand over your enterprise or data center to OpenStack control. As previously stated, OpenStack can be configured to work very traditionally in terms of network integration, but the framework is well positioned to take advantage of the SDN model and technologies. The Open Networking Foundation (www.opennetworking.org) was founded to promote SDN and is a good starting point for gaining a deeper understanding of SDN.

In the next section, you'll learn about the types of vendor networking used in OpenStack.

Examples of vendor networking in OpenStack

In early versions of OpenStack, networking was provided in a traditional way, with networking being managed by OpenStack Compute (Nova). As demand for network control outside the scope of OpenStack Compute grew, OpenStack Networking (originally Quantum, and later Neutron) was developed as a separate project.

As previously described, Neutron manages vendor networking using vendor-specific plug-ins. As the community added more and more support for vendor networking, the need for further modularity through a standard plug-in module was identified. The benefits of modular plug-ins include reduced redundant code, easier vendor integration, and standardization of core network functions.

With the release of OpenStack Havana in late 2013, the Neutron Modular Layer 2 (ML2) plug-in was introduced. The ML2 plug-in is divided into *type* and *mechanism* drivers. Figure 4.18 shows the hierarchy of the ML2 plug-in with the type and mechanism drivers.

The *type* drivers, as the name suggests, are related to the type of network the plug-in manages. You can think of the type driver as how Neutron manages the endpoints. For example, Neutron could specify a tunnel be created between endpoints without knowing anything about the network between the endpoints. This gets us back to the discussion about the separation of control and data planes.

The *mechanism* drivers are responsible for managing the virtual and physical network devices that are attached to endpoints. These drivers create, update, and delete network and port resources based on the requirements of the type driver.

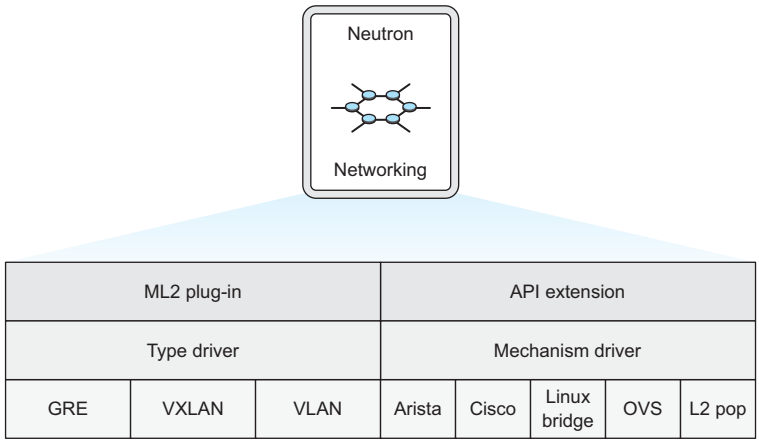

Figure 4.18 Network management with the Neutron ML2 plug-in

The goal of the ML2 plug-in is to replace many of the monolithic plug-ins that exist today.

> **EXAMPLES OF VENDOR NETWORKING IN OPENSTACK** Neutron plug-ins have been developed for many vendors, including Arista, Cisco, Nicira/VMware, NEC, Brocade, IBM, and Juniper. In addition, ML2 drivers have been developed for Big Switch/Floodlight, Arista, Mellanox, Cisco, Brocade, Nicira/VMware, and NEC.

The next section will touch on what you've learned in the first part of this book and what will be covered in the second part.

4.3 Why walk through a manual deployment?

In chapter 1 you were introduced to OpenStack. In that introduction you learned how OpenStack fits into the cloud ecosystem, why you might want to adopt the technology, and what the focus of this book will be. In chapter 2, motivated by the fantastic possibilities described in the first chapter, you took a limited test-drive of the OpenStack framework, working through some exercises that didn't require an in-depth knowledge of the framework. Chapter 3 presented more examples, but this time from an operational perspective, giving you further insight into the structure of the framework. Finally, in this chapter you learned how OpenStack works through its framework of components and interoperates with vendor hardware and software.

You've covered a great deal in four chapters. If you completed all of the exercises and have a working DevStack deployment, congratulate yourself! You might (unfortunately) already be considered an OpenStack expert in many organizations. But although the first part of this book may be sufficient to make you look like an expert, there's much more to learn before you take the leap to a multi-node production deployment.

Part 2 of this book covers deploying OpenStack manually, going through each command and configuration, and explaining both the steps involved and what they mean. If your view is more high-level, or you plan on relying on a vendor for your OpenStack support, you can skip to part 3, where we'll cover topics related to design, implementation, and even the economics of OpenStack production deployments. This being said, even if you expect a fully managed OpenStack solution to be provided by a vendor, there's certainly value in knowing what's going on under the covers. I recommend at least reviewing part 2, even if you don't plan on personally deploying a production OpenStack environment.

4.4 Summary

- OpenStack is a framework that consists of many projects.
- OpenStack project designations range from core (integral parts of OpenStack) to related (projects that have some relation).
- OpenStack works using a collection of distributed core components.
- Core components communicate with each other using their respective APIs.
- OpenStack can manage vendor-provided hardware and software.
- OpenStack manages vendor-provided hardware and software through component plug-ins.

CLOUD/SOFTWARE DEVELOPMENT

OpenStack IN ACTION

V. K. Cody Bumgardner

OpenStack is an open source framework that lets you create a private or public cloud platform on your own physical servers. You build custom infrastructure, platform, and software services without the expense and vendor lock-in associated with proprietary cloud platforms like Amazon Web Services and Microsoft Azure. With an OpenStack private cloud, you can get increased security, more control, improved reliability, and lower costs.

OpenStack in Action offers real-world use cases and step-by-step instructions on how to develop your own cloud platform. This book guides you through the design of both the physical hardware cluster and the infrastructure services you'll need. You'll learn how to select and set up virtual and physical servers, how to implement software-defined networking, and technical details of designing, deploying, and operating an OpenStack cloud in your enterprise. You'll also discover how to best tailor your OpenStack deployment for your environment. Finally, you'll learn how your cloud can offer user-facing software and infrastructure services.

What's Inside

- Develop and deploy an enterprise private cloud
- Private cloud technologies from an IT perspective
- Organizational impact of self-service cloud computing

No prior knowledge of OpenStack or cloud development is assumed.

Cody Bumgardner is the Chief Technology Architect at a large university where he is responsible for the architecture, deployment, and long-term strategy of OpenStack private clouds and other cloud computing initiatives.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/books/openstack-in-action

"An excellent primer on the complex world of cloud computing and the OpenStack software ecosystem."
—From the Foreword by Jay Pipes, Member, OpenStack Technical Committee

"Provides enough theory and practice to understand the subject matter with just the right level of detail."
—Hafizur Rahman
Kii Corporation

"A fundamental resource for learning, installing, and managing this exciting piece of cloud infrastructure."
—Michael Hamrah, Getty Images

"If you thought that AWS was the only player, you need to read this book."
—Kosmas Chatzimichalis
Mach 7x

Free eBook
SEE INSERT

ISBN 13: 978-1-61729-216-3
ISBN 10: 1-61729-216-8
55499
9 781617 292163

MANNING $54.99 / Can $63.99 [INCLUDING eBOOK]