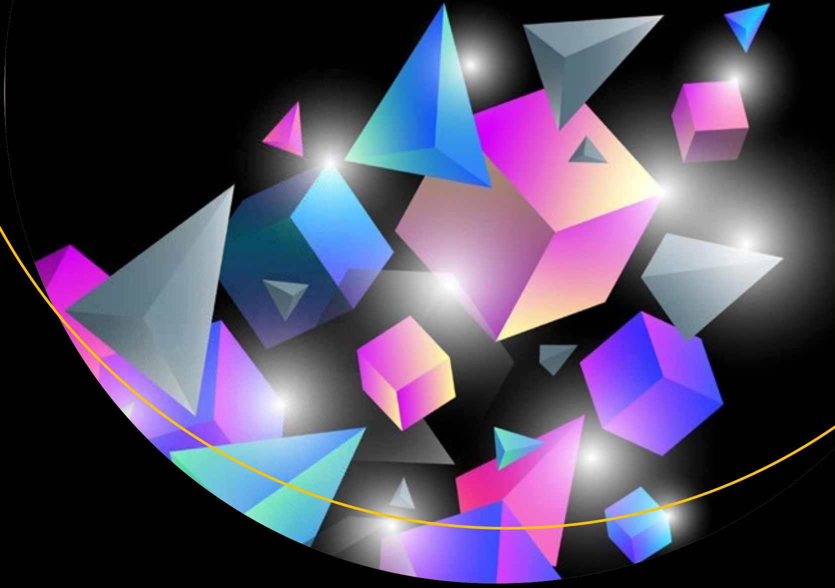

Foundations of Libvirt Development

How to Set Up and Maintain a Virtual Machine Environment with Python

W. David Ashley

Apress®

Foundations of Libvirt Development

How to Set Up and Maintain a Virtual Machine Environment with Python

W. David Ashley

Apress®

Foundations of Libvirt Development: How to Set Up and Maintain a Virtual Machine Environment with Python

W. David Ashley
Austin, TX, USA

ISBN-13 (pbk): 978-1-4842-4861-4 ISBN-13 (electronic): 978-1-4842-4862-1
https://doi.org/10.1007/978-1-4842-4862-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Divya Modi

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-4861-4. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

To all the teachers to whom I owe so much,
I hope I have made you proud. And to all of my students,
I hope I was able to pass some knowledge on to you.

Table of Contents

About the Author

W. David Ashley is a technical writer for SkillSoft where he specializes in open source, particularly Linux. As a member of the Linux Fedora documentation team, he recently led the libvert project documentation and wrote the Python programs included with it. He has developed in 20 different programming languages during his 35 years as a software developer and IT consultant, including more than 18 years at IBM and 12 years with American Airlines.

About the Technical Reviewer

Nishant Krishna is an entrepreneur, software architect, innovator, and inventor with 18+ years of experience working on architecture, anticounterfeiting technologies, EdTech, cloud and virtualization, Internet of Things (IoT) technologies, machine learning, security compliance, API development, and network management systems (NMSs). He can be reached at https://in.linkedin.com/in/nishantkrishna and https://twitter.com/nishantkrishna.

Acknowledgments

I would like to acknowledge the libvirt development team and Red Hat for their contributions to the Linux community. Without their in-depth help and encouragement, this book would not have been possible.

CHAPTER 1

Introduction

Welcome to the first chapter of *Foundations of Libvirt Development*. In this chapter, you'll learn what this book covers and the conventions I use throughout.

What This Book Covers

This book is about the libvirt APIs and how to work with it to control virtual machines under the QEMU/KVM system. libvirt has APIs that support many languages, but this work concentrates on the Python language exclusively.

libvirt can be used to support a number of virtual machine types, and the APIs are a common entry point to any type of virtual machine. The virtual machines that are supported include (but are not limited to) KVM, QEMU, Xen, Virtuozzo, VMware ESX, LXC, bhyve, and more. All of these can be controlled programmatically via the libvirt APIs. The libvirt APIs works the same way across all the supported platforms.

This book does not cover how to install QEMU/KVM on any platform. It is assumed that you either have that knowledge already or are working on a system where it is already installed. See the documentation for your operating system for instructions on how to install virtual machine support.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_1

The libvirt Python API is an object-oriented system. It provides classes for virtual connections, virtual machine (domain) interactions, virtual and real network support, storage support, and information retrieval from the main host system. All interaction with the system is through the supported classes, and thus you will need to understand object orientation under Python. Be sure you have this knowledge before attempting to use the Python libvirt API.

In addition to the API topics mentioned, the book covers general interactions such as creating/destroying entities like domains, networks, storage, and others. The book also covers topics such as obtaining information about objects, getting statistics about objects, changing the configuration of an object, starting/stopping an object, and adding/removing hardware from objects.

The libvirt API covers the entire life cycle of virtual objects, from creation to destruction. It contains everything needed to manage a virtual object during that life cycle. However, it doesn't contain APIs for managing the object from inside (i.e., there are no APIs that connect to the object's internal APIs). Thus, for domains (virtual machines), there is no way to submit jobs to the domain or monitor anything going on in the domain. That type of administration is left up to the administrator.

Book Conventions

This book uses several conventions to draw attention to specific pieces of information. The convention used depends on the type of information displayed.

Computer Commands

Computer commands are usually presented in bold text such as in the following example:

> Run the Unix command **ls** in a command shell to present a list of files and directories.

File Names

File names are usually presented in monospaced text such as in the following example:

> To see the contents of the file `report.txt`, use the command **cat report.txt**.

Programming Language Elements and Literals

Programming language elements include such things as variable names, literals, constants, symbols, tokens, functions, class names, and other objects.

Literal data is usually taken directly from a computer screen or a computer language literal value and presented in monospaced text such as in the following example:

> The following line is the output from running **ls**:

```
en-US Makefile publican.cfg
```

Computer Output and Source Code

Computer output data is usually taken directly from a computer screen and presented in monospaced text such as in the following example.

```
books         Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests   Desktop1  downloads      images  notes  scripts  svgs   libvirt
```

Source code listings are also set off from the rest of the text as follows:

```
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
conn.close()
exit(0)
```

Notes and Warnings

Finally, I use three visual styles to draw attention to information that might otherwise be overlooked.

Note Notes are tips, shortcuts, or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important Important boxes detail things that are easily missed such as configuration changes that apply only to the current session, or services that need restarting before an update will apply. Ignoring a box labeled "Important" will not cause data loss but may cause irritation and frustration.

Warning Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

CHAPTER 2

Architecture

This chapter describes the main principles and architecture choices behind the libvirt API and the Python libvirt module. These include the object model, the driver model, and management options. The details of these models and options are described in this chapter.

Object Model

The goal of both the libvirt API and the Python libvirt module is to provide all the functions necessary for deploying and managing virtual machines. This includes the management of both the core hypervisor functions and the host resources that are required by virtual machines, such as networking, storage, and PCI/USB devices. Most of the classes and methods exposed by libvirt have a pluggable internal back end, providing support for different underlying virtualization technologies and operating systems. Thus, the extent of the functionality available from a particular API or method is determined by the specific hypervisor driver in use and the capabilities of the underlying virtualization technology.

Hypervisor Connections

A *connection* is the primary, or top-level, object in the libvirt API and Python libvirt module. An instance of this object is required before attempting to use almost any of the classes or methods. A connection is

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_2

associated with a particular hypervisor, which may be running locally on the same machine as the libvirt client application or on a remote machine over the network. In all cases, the connection is represented by an instance of the `virConnect` class and identified by a URI. The URI scheme and path define the hypervisor to connect to, while the host part of the URI determines where it is located. Refer to the section called "URI Formats" in Chapter 3 for a full description of valid URIs.

An application is permitted to open multiple connections at the same time, even when using more than one type of hypervisor on a single machine. For example, a host may provide both KVM full machine virtualization and Linux containers. A connection object can be used concurrently across multiple threads. Once a connection has been established, it is possible to obtain handles to other managed objects or create new managed objects, as discussed in the next section, "Guest Domains."

Guest Domains

A *guest domain* can refer to either a running virtual machine or a configuration that can be used to launch a virtual machine. The connection object provides methods to enumerate the guest domains, create new guest domains, and manage existing domains. A guest domain is represented with an instance of the `virDomain` class and has a number of unique identifiers.

- **ID**: This is a positive integer, unique among the running guest domains on a single host. An inactive domain does not have an ID.
- **Name**: This is a short string, unique among all the guest domains on a single host, both running and inactive. To ensure maximum portability between hypervisors, it is recommended that names include only alphanumeric (`a–Z`, `0–9`), hyphen (`-`), and underscore (`_`) characters.

- **UUID**: This consists of 16 unsigned bytes, guaranteed to be unique among all guest domains on any host. RFC 4122 defines the format for UUIDs and provides a recommended algorithm for generating UUIDs with guaranteed uniqueness.

A guest domain may be transient or persistent. A transient guest domain can be managed only while it is running on the host. Once it is powered off, all traces of it will disappear. A persistent guest domain has its configuration maintained in a data store on the host by the hypervisor, in an implementation-defined format. Thus, when a persistent guest is powered off, it is still possible to manage its inactive configuration. A transient guest can be turned into a persistent guest while it is running by defining a configuration for it.

Virtual Networks

A virtual network provides a method for connecting the network devices of one or more guest domains within a single host. The virtual network can do either of the following:

- Remain isolated to the host.
- Allow routing of traffic off-node via the active network interfaces of the host OS. This includes the option to apply NAT to IPv4 traffic.

A virtual network is represented by an instance of the `virNetwork` class and has two unique identifiers.

- **Name**: This is a short string, unique among all the virtual networks on a single host, both running and inactive. For maximum portability between hypervisors, applications should use only alphanumeric (`a–Z`, `0–9`), hyphen (`-`), and underscore (`_`) characters in names.

- **UUID**: This consists of 16 unsigned bytes, guaranteed to be unique among all the virtual networks on any host. RFC 4122 defines the format for UUIDs and provides a recommended algorithm for generating UUIDs with guaranteed uniqueness.

A virtual network can be transient or persistent. A transient virtual network can be managed only while it is running on the host. When taken offline, all traces of it will disappear. A persistent virtual network has its configuration maintained in a data store on the host, in an implementation-defined format. Thus, when a persistent network is brought offline, it is still possible to manage its inactive configuration. A transient network can be turned into a persistent network on the fly by defining a configuration for it.

After the installation of libvirt, every host will get a single virtual network instance called `default`, which provides DHCP services to guests and allows NAT'd IP connectivity to the host's interfaces. This service is of most use to hosts with intermittent network connectivity such as laptops using wireless networking.

Refer to Chapter 6 for further information about using virtual network objects.

Storage Pools

The storage pool object provides a mechanism for managing all types of storage on a host, such as local disks, logical volume groups, iSCSI targets, Fibre Channel HBAs, and local/network file systems. A *pool* refers to a quantity of storage that can be allocated to form individual volumes. A storage pool is represented by an instance of the `virStoragePool` class and has a pair of unique identifiers.

- **Name**: This is a short string, unique among all the storage pools on a single host, both running and inactive. For maximum portability between hypervisors, applications should rely on using only alphanumeric (`a–Z`, `0–9`), hyphen (`-`), and underscore (`_`) characters in names.
- **UUID**: This consists of 16 unsigned bytes, guaranteed to be unique among all the storage pools on any host. RFC 4122 defines the format for UUIDs and provides a recommended algorithm for generating UUIDs with guaranteed uniqueness.

A storage pool can be transient or persistent. A transient storage pool can be managed only while it is running on the host, and when powered off, all traces of it will disappear. A persistent storage pool has its configuration maintained in a data store on the host by the hypervisor, in an implementation-defined format. Thus, when a persistent storage pool is deactivated, it is still possible to manage its inactive configuration. A transient pool can be turned into a persistent pool on the fly by defining a configuration for it.

Refer to Chapter 5 for further information about using storage pool objects.

Storage Volumes

The storage volume object allows you to allocate a block of storage within a pool, be it a disk partition, a logical volume, a SCSI/iSCSI LUN, or a file within a local/network file system. Once allocated, a volume can be used to provide disks to one (or more) virtual domains. A volume is represented by an instance of the `virStorageVol` class and has three unique identifiers.

- **Name**: This is short string, unique among all storage volumes within a storage pool. For maximum portability between implementations, applications should rely on using only alphanumeric (`a–Z`, `0–9`), hyphen (`-`), and underscore (`_`) characters in names. The name is not guaranteed to be stable across reboots or between hosts, even if the storage pool is shared between hosts.
- **Key**: This is a unique string, of arbitrary printable characters, intended to uniquely identify the volume within the pool. The key is intended to be stable across reboots and between hosts.
- **Path**: This is a file system path referring to the volume. The path is unique among all the storage volumes on a single host. If the storage pool is configured with a suitable target path, the volume path may be stable across reboots and between hosts.

Refer to Chapter 5 for further information about using storage volume objects.

Host Devices

Host devices provide a view to the hardware devices available on the host machine. This covers both the physical USB or PCI devices and the logical devices these provide, such as NICs, disks, disk controllers, sound cards, and so on. Devices can be arranged to form a tree structure, allowing relationships to be identified.

A host device is represented by an instance of the `virNodeDev` class and has one general identifier (a name), though specific device types may have their own unique identifiers. A name is a short string, unique among all the devices on the host. The naming scheme is determined by the host operating system. The name is not guaranteed to be stable across reboots.

Physical devices can be detached from the host OS driver, which implicitly removes all associated logical devices. If the device is removed, then it will also be removed from any domain that references it. Physical device information is also useful when working with the storage and networking APIs to determine what resources are available to configure. Host devices are currently not covered in this guide.

Driver Model

The libvirt library exposes a guaranteed stable API and ABI, both of which are decoupled from any particular virtualization technology. In addition, many of the APIs have associated XML schemata, which are considered part of the stable ABI guarantee. Internally, there are multiple implementations of the public ABI, each targeting a different virtualization technology. Each implementation is referred to as a *driver*. When obtaining an instance of the `virConnect` class, the application developer can provide a URI to determine which hypervisor driver is activated.

No two virtualization technologies have the same functionality. The libvirt goal is not to restrict applications to a lowest common denominator since this would result in an unacceptably limited API. Instead, libvirt attempts to define a representation of concepts and configuration that is hypervisor agnostic and adaptable to allow future extensions. Thus, if two hypervisors implement a comparable feature, libvirt provides a uniform control mechanism or configuration format for that feature.

If a libvirt driver does not implement a particular API, then it will return a `VIR_ERR_NO_SUPPORT` error code enabling this to be detected. There is also an API to allow applications to query certain capabilities of a hypervisor, such as the type of guest ABIs that are supported. Figure 2-1 shows all the possible driver models that can be referenced via the API.

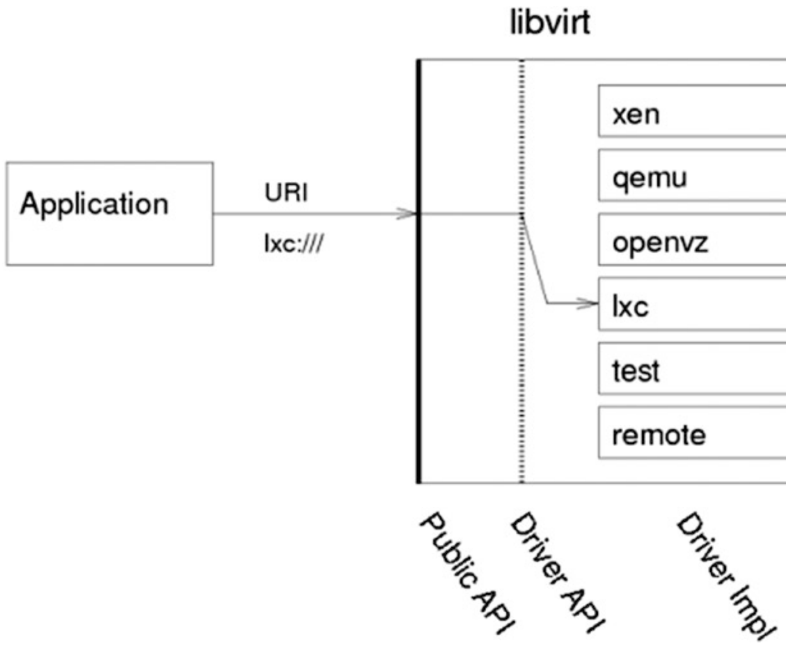

Figure 2-1. *Libvirt driver architecture*

Internally, a libvirt driver will attempt to utilize whatever management channels are available for the virtualization technology in question. For some drivers, this may require libvirt to run directly on the host being managed, talking to a local hypervisor, while others may be able to communicate remotely over an RPC service. For drivers that have no native remote communication capability, libvirt provides a generic secure RPC service. This is discussed in detail later in this chapter.

The following is a list of hypervisor drivers:

- **Xen**: The open source Xen hypervisor provides paravirtualized and fully virtualized machines. A single system driver runs in the Dom0 host talking directly to a combination of the hypervisor, xenstored, and xend. An example local URI scheme is `xen:///`.

- **QEMU**: This supports any open source QEMU-based virtualization technology, including KVM. A single privileged system driver runs in the host managing QEMU processes. Each unprivileged user account also has a private instance of the driver. An example privileged URI scheme is `qemu:///system`. An example unprivileged URI scheme is `qemu:///session`.
- **UML**: This is the User Mode Linux kernel, a pure paravirtualization technology. A single privileged system driver runs in the host managing UML processes. Each unprivileged user account also has a private instance of the driver. An example privileged URI scheme is `uml:///system`. An example unprivileged URI scheme is `uml:///session`.
- **OpenVZ**: This is the OpenVZ container-based virtualization technology, using a modified Linux host kernel. A single privileged system driver runs in the host talking to the OpenVZ tools. An example privileged URI scheme is `openvz:///system`.
- **LXC**: This is the native Linux container-based virtualization technology, available with Linux kernels since 2.6.25. A single privileged system driver runs in the host talking to the kernel. An example privileged URI scheme is `lxc:///`.
- **Remote**: This is a generic secure RPC service talking to a `libvirtd` daemon. It provides encryption and authentication using a choice of TLS, x509 certificates, SASL (GSSAPI/Kerberos), and SSH tunneling. The URIs follow the scheme of the desired driver, but with a hostname filled in and a data transport name

appended to the URI scheme. An example URI to talk to Xen over a TLS channel is `xen+tls://somehostname/`. An example URI to talk to QEMU over a SASL channel is `qemu+tcp:///somehost/system`.

- **Test**: This is a mock driver, providing a virtual in-memory hypervisor covering all the libvirt APIs. It facilitates the testing of applications using libvirt by allowing automated tests to run the exercises' libvirt APIs without needing to deal with a real hypervisor. An example default URI scheme is `test:///default`. An example customized URI scheme is `test:///path/to/driver/config.xml`.

Remote Management

While many virtualization technologies provide a remote management capability, libvirt does not assume this and provides a dedicated driver allowing for the remote management of any libvirt hypervisor driver. The driver has a variety of data transports providing considerable security for the data communication. The driver is designed such that there is 100 percent functional equivalence whether talking to the libvirt driver locally or via the RPC service.

In addition to the native RPC service included in libvirt, there are a number of alternatives for remote management that will not be discussed in this book. The `libvirt-qpid` project provides an agent for the QPid messaging service, exposing all libvirt managed objects and operations over the message bus. This keeps a fairly close, near one-to-one, mapping to the C API in libvirt. The `libvirt-CIM` project provides a CIM agent that maps the libvirt object model onto the DMTF virtualization schema.

Basic Usage

The server end of the RPC service is provided by the libvirtd daemon, which must be run on the host to be managed. In a default deployment, this daemon will be listening for connections only on a local UNIX domain socket. This allows for a libvirt client to use only the SSH tunnel data transport. With a suitable configuration of x509 certificates, or SASL credentials, the libvirtd daemon can be told to listen on a TCP socket for direct, nontunneled client connections.

As you can see from the previous example of libvirt driver URIs, the hostname field in the URI is always left empty for local libvirt connections. To make use of the libvirt RPC driver, only two changes are required to the local URI. At least one hostname must be specified, at which point libvirt will attempt to use the direct TLS data transport. An alternative data transport can be requested by appending its name to the URI scheme. The URIs formats will be described in detail in Chapter 6.

Data Transports

To cope with the wide variety of deployment environments, the libvirt RPC service supports a number of data transports, all of which can be configured with industry-standard encryption and authentication capabilities.

Here are the libvirt transports:

tls

This is a TCP socket running the TLS protocol on the wire. This is the default data transport if none is explicitly requested, and it uses a TCP connection on port 16514. At a minimum, it is necessary to configure the server with an x509 certificate authority and issue it a server certificate. The libvirtd server can, optionally, be configured to require clients to present x509 certificates as a means of authentication.

tcp

This is a TCP socket without the TLS protocol on the wire. This data transport should not be used on untrusted networks, unless the SASL authentication service has been enabled and configured with a plug-in that provides encryption. The TCP connection is made on port 16509.

unix

This is a local-only data transport, allowing users to connect to a libvirtd daemon running as a different user account. As it is accessible only on the local machine, it is unencrypted. The standard socket names are `/var/run/libvirt/libvirt-sock` for full management capabilities and `/var/run/libvirt/libvirt-sock-ro` for a socket restricted to read-only operations.

ssh

The RPC data is tunneled over an SSH connection to the remote machine. It requires that Netcat (`nc`) be installed on the remote machine and that libvirtd is running with the UNIX domain socket enabled. It is recommended that SSH be configured to not require password prompts to the client application. For example, if you're using SSH public key authentication, it is recommended that you run an ssh-agent to cache key credentials. GSSAPI is another useful authentication mode for the SSH transport allowing use of a pre-initialized Kerberos credential cache.

ebxt

This supports any external program that can make a connection to the remote machine by means that are outside the scope of libvirt. If none of the built-in data transports is satisfactory, this allows an application to provide a helper program to proxy RPC data over a custom channel.

Authentication Schemes

To cope with the wide variety of deployment environments, the libvirt RPC service supports a number of authentication schemes on its data transports, with industry-standard encryption and authentication capabilities. The choice of authentication scheme is configured by the administrator in the `/etc/libvirt/libvirtd.conf` file.

Here are the libvirt schemes:

sasl

SASL is an industry standard for pluggable authentication mechanisms. Each plug-in has a wide variety of capabilities, and discussion of their merits is outside the scope of this document. For the `tls` data transport, there is a wide choice of plug-ins, since TLS is providing data encryption for the network channel. For the `tcp` data transport, libvirt will refuse to use any plug-in that does not support data encryption. This effectively limits the choice to GSSAPI/Kerberos. SASL can optionally be enabled on the UNIX domain socket data transport if strong authentication of local users is required.

polkit

PolicyKit is an authentication scheme suitable for local desktop virtualization deployments, for use only on the UNIX domain socket data transport. It enables the libvirtd daemon to validate that the client application is running within the local X desktop session. It can be configured to allow access to a logged-in user automatically or can prompt them to enter their own password or the superuser (root) password.

x509

Although not strictly an authentication scheme, the TLS data transport can be configured to mandate the use of client x509 certificates. The server can then whitelist the client distinguished names to control access.

Generating TLS Certificates

libvirt supports TLS certificates for verifying the identity of the server and clients. There are two distinct checks involved.

1. The client checks that it is connecting to the correct server by matching the certificate the server sends with the server's hostname. This check can be disabled by adding `?no_verify=1`.
2. The server checks to ensure that only allowed clients are connected. This is performed using either of the following:

a. The client's IP address

b. The client's IP address and the client's certificate

Server checking can be enabled or disabled using the `libvirtd.conf` file.

For full certificate checking, you will need to have certificates issued by a recognized certificate authority (CA) for your server (or servers) and all clients. To avoid the expense of obtaining certificates from a commercial CA, there is the option to set up your own CA and tell your servers and clients to trust certificates issued by your own CA. To do this, follow the instructions in the next section.

It should be noted that using a certificate using the FODN/hostname where the host is not present in the DNS will cause problems unless the host is added to the hosts file. Alternatively, if you replace the hostname with the IP address, that too may cause problems.

Be aware that the default configuration for `libvirtd.conf` allows any client to connect, provided that they have a valid certificate issued by the CA for their own IP address. You may need to make this setting more or less permissive, depending upon your requirements.

Public Key Infrastructure Setup

The public key infrastructure (PKI) provides access control for a domain through the API (Table 2-1). For a full discussion of this facility, please refer to `https://libvirt.org/auth.html`. By default, any user needing access to domain clients will need to have a PKI certificate created for them by the administrator. Otherwise, the user will have no access to the domains via the API.

Table 2-1. *Public Key Setup*

Location	Machine	Description	Required Fields
`/etc/pki/CA/cacert.pem`	Installed on all clients and servers	CA's certificate	n/a
`/etc/pki/libvirt/private/serverkey.pem`	Installed on the server	Server's private key	n/a
`/etc/pki/libvirt/servercert.pem`	Installed on the server	Server's certificate signed by the CA	CommonName (CN) must be the hostname of the server as it is seen by clients.
`/etc/pki/libvirt/private/clientkey.pem`	Installed on the client	Client's private key.	n/a
`/etc/pki/CA/cacert.pem`	Installed on the client	Client's certificate signed by the CA	Distinguished Name (DN) can be checked against an access control list (`tls_allowed_dn_list`).

Summary

This chapter introduced the components that make up the architecture of libvirt. Much of the information in this chapter will be further expanded on in subsequent chapters. Later chapter will introduce the Python API and will show detailed functions that can communicate with these components.

CHAPTER 3

Connecting to Domains

As mentioned in the previous chapter, a *connection* is the foundation of every action and object in the libvirt system. Every entity that wants to interact with libvirt, be it virsh, virt-manager, or a program using the libvirt library, needs to first obtain a connection to the libvirt daemon on the host node it is interested in. A connection describes not only the type of virtualization technology that the agent wants to interact with (QEMU, XEN, UML, etc.) but also any authentication methods necessary to connect to that resource.

This chapter describes the connection object in detail, including how to open and close the object, how to connect to a particular driver and its associated domains, the URI formats that describe the connection, the host capabilities, and the host's physical hardware.

Overview

The first thing a libvirt agent must do is call the `virInitialize` function, or one of the Python libvirt connection functions, to obtain an instance of the `virConnect` class. This instance will be used in subsequent operations. The Python libvirt module provides three different functions for connecting to a resource.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_3

```
conn = libvirt.open(name)
conn = libvirt.openAuth(uri, auth, flags)
conn = libvirt.openReadOnly(name)
```

In all three cases, there is a name parameter that refers to the URI of the hypervisor to connect to. Chapter 2 provided full details on the various URI formats that are acceptable. If the URI is None, then libvirt will apply some heuristics and probe for a suitable hypervisor driver. While this may be convenient for developers doing ad hoc testing, it is strongly recommended that applications do not rely on probing logic since it may change at any time. Applications should always explicitly request which hypervisor connection is desired by providing a URI.

The difference between the three methods is the way in which they authenticate and the resulting authorization level they provide.

open

The open function will attempt to open a connection for full read-write access. It does not have any scope for authentication callbacks to be provided, so it will succeed only for connections where authentication can be done based on the credentials of the application. Listing 3-1 shows how to connect to the qemu system to obtain a connection object that will be used next to close the connection.

Listing 3-1. Using open

```
# Listing-1.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
print('Connection successful.')
conn.close()
print('Connection closed.')
exit(0)
```

This example opens up a read-write connection to the system's QEMU hypervisor driver, checks to make sure it was successful, and, if so, closes the connection. For more information on libvirt URIs, refer to Chapter 2.

openReadOnly

The `openReadOnly` function will attempt to open a connection for read-only access (Listing 3-2). Such a connection has a restricted set of method calls that are allowed and is typically useful for monitoring applications that should not be allowed to make changes. As with `open`, this method has no scope for authentication callbacks, so it relies on credentials.

Almost all functions that return informational data can be used in read-only mode. That includes most of the functions described in this chapter. Obviously, modifying the domains within a connection is not possible with a read-only connection, including starting and stopping a domain.

Listing 3-2. Using openReadOnly

```
# Example-2.py
from __future__ import print_function
import sys
import libvirt
```

```
conn = libvirt.openReadOnly('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
conn.close()
exit(0)
```

This example opens up a read-only connection to the system QEMU hypervisor driver, checks to make sure it was successful, and, if so, closes the connection. For more information on libvirt URIs, refer to Chapter 2.

openAuth

The `openAuth` function is the most flexible and effectively makes the previous two functions redundant. It takes an extra parameter that provides a Python `list` that contains the authentication credentials from the client app. The `flags` parameter allows the application to request a read-only connection with the `VIR_CONNECT_RO` flag if desired. Listing 3-3 shows a simple example Python program that uses `openAuth` with username and password credentials. As with `open`, this method has no scope for authentication callbacks, so it relies on credentials.

Listing 3-3. Using openAuth

```
# Example-3.py
from __future__ import print_function
import sys
import libvirt

SASL_USER = "my-super-user"
SASL_PASS = "my-super-pass"
```

```
def request_cred(credentials, user_data):
    for credential in credentials:
        if credential[0] == libvirt.VIR_CRED_AUTHNAME:
            credential[4] = SASL_USER
        elif credential[0] == libvirt.VIR_CRED_PASSPHRASE:
            credential[4] = SASL_PASS
    return 0

auth = [[libvirt.VIR_CRED_AUTHNAME, libvirt.VIR_CRED_PASSPHRASE], \
        request_cred, None]

conn = libvirt.openAuth('qemu+tcp://localhost/system', auth, 0)
if conn == None:
    print('Failed to open connection to qemu+tcp://localhost/
    system', \
          file=sys.stderr)
    exit(1)
conn.close()
exit(0)
```

To test this program, the all the following configurations must be present:

1. `/etc/libvirt/libvirtd.conf`

   ```
   listen_tls = 0
   listen_tcp = 1
   auth_tcp = "sasl"
   ```

2. `/etc/sasl2/libvirt.conf`

   ```
   mech_list: digest-md5
   ```

3. A user of the virt program has been added to the SASL database. This is usually a relational database configured to hold SASL credentials.

4. `libvirtd` has been started with `--listen`.

Once the configurations are done, Listing 3-3 can utilize the configured username and password and allow read-write access to libvirtd.

Unlike the libvirt C interface, Python does not provide for custom callbacks to gather credentials.

It should also be noted that SASL credentials are not the only authentication method available for this API. There are any number of credential interfaces available, including PolicyKit (PKI), GSSAPI, SSH, ESX, and XEN. Any or all of these can be configured.

close

A connection must be released by calling the `close` method of the `virConnection` class when no longer required. Connections are reference-counted objects, so there should be a corresponding call to the `close` method for each `open` function call (Listing 3-4).

As mentioned, connections are reference-counted; the count is explicitly increased by the initial `open`, `openAuth`, and the like. It is also temporarily increased by other methods that depend on the connection remaining alive. The `open` function call should have a matching `close`, and all other references will be released after the corresponding operation completes.

In Python, reference counts can be automatically decreased when a class instance goes out of scope or when the program ends. Reference counts are also decremented to zero when the process ends.

Listing 3-4. Using close with Additional References

```
# Example-5.py
from __future__ import print_function
import sys
import libvirt
```

```
conn1 = libvirt.open('qemu:///system')
if conn1 == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
conn2 = libvirt.open('qemu:///system')
if conn2 == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
conn1.close()
conn2.close()
exit(0)
```

Also note that every other class instance associated with a connection (`virDomain`, `virNetwork`, etc.) will hold a reference on the connection.

URI Formats

libvirt uses uniform resource identifiers (URIs) to identify hypervisor connections. Both local and remote hypervisors are addressed by libvirt using URIs. The URI scheme and path define the hypervisor to connect to, while the host part of the URI determines where it is located.

Local URIs

libvirt local URIs have one of the following forms:

```
driver:///system
driver:///session
driver+unix:///system
driver+unix:///session
```

All other uses of the libvirt URIs are considered remote and behave as such, even if connecting to localhost. See the section called "Remote URIs" for details on remote URIs.

The drivers listed in Table 3-1 are currently supported.

Table 3-1. *Supported Drivers*

Driver	Description
QEMU	For managing qemu and KVM guests
XEN	For managing old-style (Xen 3.1 and older) Xen guests
XENAPI	For managing new-style Xen guests
UML	For managing UML guests
LXC	For managing Linux containers
VBOX	For managing VirtualBox guests
OPENVZ	For managing OpenVZ containers
ESX	For managing VMware ESX guests
ONE	For managing OpenNebula guests
PHYP	For managing Power Hypervisor guests

Listing 3-5 shows how to connect to a local QEMU hypervisor using a local URI.

Listing 3-5. Connecting to a Local QEMU Hypervisor

```
# Example-6.py
from __future__ import print_function
import sys
import libvirt
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
conn.close()
exit(0)
```

Remote URIs

Remote URIs have the general form shown here ([...] means an optional part):

```
driver[+transport]://[username@][hostname][:port]/[path]
[?extraparameters]
```

Each component of the URI is described in Table 3-2.

Table 3-2. *URI Components*

Component	Description
`driver`	The name of the libvirt hypervisor driver to connect to. This is the same as that used in a local URI. Some examples are `xen`, `qemu`, `lxc`, `openvz`, and `test`. As a special case, the pseudo driver name `remote` can be used, which will cause the remote daemon to probe for an active hypervisor and pick one to use. As a general rule, if the application knows what hypervisor it wants, it should always specify the explicit driver name and not rely on automatic probing. Relying on automatic probing can cause the system to choose the wrong hypervisor for the needed domain.

(*continued*)

Table 3-2. (*continued*)

Component	**Description**
`transport`	The name of one of the data transports described earlier in this section. Possible values include `tls`, `tcp`, `unix`, `ssh`, and `ext`. If omitted, it will default to `tls` if a hostname is provided, or `unix` if no hostname is provided.
`username`	When using the SSH data transport, this allows the choice of a username that differs from the client's current login name.
`hostname`	The fully qualified hostname of the remote machine. If using TLS with x509 certificates, or SASL with the GSSAPI/Keberos plug-in, it is critical that this hostname match the hostname used in the server's x509 certificates/Kerberos principle. Mismatched hostnames will guarantee authentication failures.
`port`	Rarely needed, unless SSH or libvirtd has been configured to run on a nonstandard TCP port. This defaults to 22 for the SSH data transport, 16509 for the TCP data transport, and 16514 for the TLS data transport.
`path`	The path should be the same path used for the hypervisor driver's local URIs. For Xen, this is always just `/`, while for QEMU this would be `/system`.
`extraparameters`	Parameters that fine-tune some aspects of the remote connection, as discussed in depth in the next section.

Based on the information described here and with reference to the hypervisor-specific URIs earlier in this chapter, it is now possible to give some remote access URI examples.

- Connect to a remote Xen hypervisor on host `node.example.com` using SSH-tunneled data transport and the SSH username `root`: `xen+ssh://root@node.example.com/`
- Connect to a remote QEMU hypervisor on host `node.example.com` using TLS with x509 certificates: `qemu://node.example.com/system`
- Connect to a remote XEN hypervisor on host `node.example.com` using TLS, skipping verification of the server's x509 certificate (Note: this is compromising your security): `xen://node.example.com/?no_verify=1`
- Connect to the local QEMU instances over a nonstandard Unix socket (the full path to the Unix socket is supplied explicitly in this case): `qemu+unix:///system?socket=/opt/libvirt/run/libvirt/libvirt-sock`
- Connect to a libvirtd daemon offering unencrypted TCP/IP connections on an alternative TCP port 5000 and use the test driver with a default configuration: `test+tcp://node.example.com:5000/default`

Extra parameters can be added to remote URIs as part of the query string (the part following `?`). Remote URIs understand the extra parameters shown in Table 3-3. Any others are passed unmodified to the back end. Note that parameter values must be URI-escaped. Refer to `http://xmlsoft.org/html/libxml-uri.html#xmlURIEscapeStr` for more information.

Table 3-3. *Extra Parameters for Remote URIs*

Name	Transports	Description
`name`	Any transport	The local hypervisor URI passed to the remote open method. This URI is normally formed by removing the transport, hostname, port number, username, and extra parameters from the remote URI, but in certain complex cases it may be necessary to supply the name explicitly. Example: `name=qemu:///system`
`command`	`ssh`, `ext`	The external command. For the `ext` transport, this is required. For SSH, the default is `ssh`. The `PATH` environment is searched for the command. Example: `command=/opt/openssh/bin/ssh`
socket	`unix`, `ssh`	The external command. For the `ext` transport, this is required. For SSH, the default is `ssh`. The `PATH` environment is searched for the command. Example: `socket=/opt/libvirt/run/libvirt/libvirt-sock`
netcat	`ssh`	The name of the Netcat command on the remote machine. The default is `nc`. For the `ssh` transport, libvirt constructs an `ssh` command, which looks like this: `command -p port [-l username] hostname netcat -U socket` where `port`, `username`, and `hostname` can be specified as part of the remote URI, and `command`, `netcat`, and `socket` come from extra parameters (or sensible defaults). Example: `netcat=/opt/netcat/bin/nc`

(*continued*)

Table 3-3. (*continued*)

Name	Transports	Description
`no_verify`	`tls`	Client checks if the server's certificate is disabled if a nonzero value is set. Note that to disable server checks of the client's certificate or IP address, you must change the libvirtd configuration. Example: `no_verify=1`
`no_tty`	`ssh`	If set to a nonzero value, this stops SSH from asking for a password if it cannot log in to the remote machine automatically (for example, when using an SSH agent). Use this when you don't have access to a terminal such as in graphical programs that use libvirt. Example: `no_tty=1`

Listing 3-6 shows how to connect to a QEMU hypervisor using a remote URI.

Listing 3-6. Connecting to a Remote QEMU Hypervisor

```
# Example-7.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu+tls://host2/system')
if conn == None:
    print('Failed to open connection to qemu+tls://host2/system', \
          file=sys.stderr)
    exit(1)
conn.close()
exit(0)
```

Capability Information Methods

The `getCapabilities` method call can be used to obtain information about the capabilities of the virtualization host. If successful, it returns a Python string containing the capabilities XML (described in a moment). If an error occurs, `None` will be returned instead. Listing 3-7 demonstrates the use of the `getCapabilities` method.

Listing 3-7. Using getCapabilities

```
# Example-8.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

caps = conn.getCapabilities() # caps will be a string of XML
print('Capabilities:\n'+caps)

conn.close()
exit(0)
```

The capabilities XML format provides information about the host virtualization technology. In particular, it describes the capabilities of the virtualization host, the virtualization driver, and the kinds of guests that the virtualization technology can launch. Note that the capabilities XML can (and does) vary based on the libvirt driver in use. Listing 3-8 provides an example of the XML capabilities.

Listing 3-8. Example QEMU Driver Capabilities

```
<capabilities>
 <host>
   <cpu>
     <arch>x86_64</arch>
   </cpu>
   <migration_features>
     <live/>
     <uri_transports>
       <uri_transport>tcp</uri_transport>
     </uri_transports>
   </migration_features>
   <topology>
     <cells num='1'>
       <cell id='0'>
         <cpus num='2'>
           <cpu id='0'/>
           <cpu id='1'/>
         </cpus>
       </cell>
     </cells>
   </topology>
 </host>

 <guest>
   <os_type>hvm</os_type>
   <arch name='i686'>
     <wordsize>32</wordsize>
     <emulator>/usr/bin/qemu</emulator>
     <machine>pc</machine>
     <machine>isapc</machine>
     <domain type='qemu'>
```

```
      </domain>
      <domain type='kvm'>
        <emulator>/usr/bin/qemu-kvm</emulator>
      </domain>
    </arch>
    <features>
      <pae/>
      <nonpae/>
      <acpi default='on' toggle='yes'/>
      <apic default='on' toggle='no'/>
    </features>
  </guest>

  <guest>
    <os_type>hvm</os_type>
    <arch name='x86_64'>
      <wordsize>64</wordsize>
      <emulator>/usr/bin/qemu-system-x86_64</emulator>
      <machine>pc</machine>
      <machine>isapc</machine>
      <domain type='qemu'>
      </domain>
      <domain type='kvm'>
        <emulator>/usr/bin/qemu-kvm</emulator>
      </domain>
    </arch>
    <features>
      <acpi default='on' toggle='yes'/>
      <apic default='on' toggle='no'/>
    </features>
  </guest>

</capabilities>
```

Note The rest of the discussion will refer to this XML using XPath notation. In the capabilities XML, there is always the `/host` subdocument and zero or more `/guest` subdocuments (while zero guest subdocuments are allowed, this means that no guests of this particular driver can be started on this particular host).

The `<host>` subdocument describes the capabilities of the host.

`<host><uuid>` shows the UUID of the host. This is derived from the SMBIOS UUID if it is available and valid or can be overridden in `libvirtd.conf` with a custom value. If neither of these is properly set, a temporary UUID will be generated each time that libvirtd is restarted.

The `<host><cpu>` subdocument describes the capabilities of the host's CPUs. It is used by libvirt when deciding whether a guest can be properly started on this particular machine and is also consulted during live migration to determine whether the destination machine supplies the necessary flags to continue to run the guest.

`<host><cpu><<arch>` is a required XML node that describes the underlying host CPU architecture. As of this writing, all libvirt drivers initialize this from the output of `uname(2)`.

`<host><cpu><features>` is an optional subdocument that describes additional CPU features present on the host. As of this writing, it is used only by the xen driver to report on the presence or lack of the `svm` or `vmx` flag and to report on the presence or lack of the `pae` flag.

`<host><cpu><arch>` is a required XML node that describes the underlying host CPU architecture. As of this writing, all libvirt drivers initialize this from the output of `uname(2)`.

`<host><cpu><model>` is an optional element that describes the CPU model that the host CPUs most closely resemble. The CPU models that libvirt currently knows about are in the `cpu_map.xml` file.

`<host><cpu><feature>` consists of zero or more elements that describe additional CPU features that the host CPUs have that are not covered in `/host/cpu/model`.

`<host><cpu><features>` is an optional subdocument that describes additional CPU features present on the host. As of this writing, it is used by the xen driver only to report on the presence or lack of the `svm` or `vmx` flag and to report on the presence or lack of the `pae` flag.

`<host><migration_features>` is an optional subdocument that describes the migration features that this driver supports on this host (if any). If this subdocument does not exist, then migration is not supported. As of this writing, the XEN, QEMU, and ESX drivers support migration.

`The <host><migration_features><live>` XML node exists if the driver supports live migration.

`<host><migration_features><uri_transports>` is an optional subdocument that describes alternate migration connection mechanisms. These alternate connection mechanisms can be useful on multihomed virtualization systems. For instance, the `virsh migrate` command might connect to the source of the migration via 10.0.0.1 and to the destination of the migration via 10.0.0.2. However, because of the security policy, the source of the migration might only be allowed to talk directly to the destination of the migration via 192.168.0.0/24. In this case, using the alternate migration connection mechanism would allow this migration to succeed. As of this writing, the xen driver supports the alternate migration mechanism `xenmigr`, while the qemu driver supports the alternate migration mechanism `tcp`. Please see the documentation on migration for more information.

The `<host><topology>` subdocument describes the NUMA topology of the host machine; each NUMA node is represented by `<host><topology><cells><cell>` and describes which CPUs are in that NUMA node. If the host machine is a UMA (non-NUMA) machine, then there will be only one cell, and all CPUs will be in this cell. This is very hardware-specific so will vary between different machines.

<host><secmodel> is an optional subdocument that describes the security model in use on the host. <host><secmodel><model> shows the name of the security model, while <host><secmodel>https://doi.org/10.1007/978-1-4842-4862-1_3 shows the domain of interpretation. For more information about security, please see Chapter 14.

Each <guest> subdocument describes a kind of guest that this host driver can start. This description includes the architecture of the guest (i.e., i686) along with the ABI provided to the guest (i.e., HVM, XEN, or UML), as listed in Table 3-4.

<guest><os_type> is a required element that describes the type of guest.

Table 3-4. *Guest Types*

Driver	Guest Type
qemu	Always hvm
xen	Either xen for a paravirtualized guest or hvm for a fully virtualized guest
uml	Always uml
lxc	Always exe
vbox	Always hvm
openvz	Always exe
one	Always hvm
ex	Not supported at this time

<guest><arch> is the root of an XML subdocument describing various virtual hardware aspects of this guest type. It has a single attribute called name, which can be used to refer to this subdocument.

<guest><arch><wordsize> is a required element that describes how many bits per word this guest type uses. This is typically 32 or 64.

`<guest><arch><emulator>` is an optional element that describes the default path to the emulator for this guest type. Note that the emulator can be overridden by the `<guest><arch><domain><emulator>` element for guest types that need alternate binaries.

`<guest><arch><loader>` is an optional element that describes the default path to the firmware loader for this guest type. Note that the default loader path can be overridden by the `<guest><arch><domain><loader>` element for guest types that use alternate loaders. Currently, this is used only by the XEN driver for HVM guests.

There can be zero or more `<guest><arch><machine>` elements that describe the default types of machines that this guest emulator can emulate. These "machines" typically represent the ABI or hardware interface that a guest can get started with. Note that these machine types can be overridden by the `<guest><arch><domain><machine>` elements for virtualization technologies that provide alternate machine types. Typical values for this are `pc` and `isapc`, meaning a regular PCI-based PC and an older, ISA-based PC, respectively.

There can be zero or more `<guest><arch><domain>` XML subtrees (although with zero `<guest><arch><domain>` XML subtrees, no guests of this driver can be started). Each `/guest/arch/domain` XML subtree has optional `<emulator>`, `<loader>`, and `<machine>` elements that override the respective defaults specified earlier. For any of the elements that are missing, the default values are used.

The `<guest><features>` optional subdocument describes various additional guest features that can be enabled or disabled, along with their default state and whether they can be toggled on or off.

Host Information

Various Python `virConnection` methods can be used to get information about the virtualization host, including the hostname, maximum support guest CPUs, and so on.

getHostname

The getHostname method can be used to obtain the hostname of the virtualization host as returned by gethostname(). It is invoked via a virConnection instance and, if successful, returns a string containing the hostname possibly expanded to a fully qualified domain name. If an error occurred, None will be returned instead. It is the responsibility of the caller to free the memory returned from this method call. Listing 3-9 demonstrates the use of getHostname.

Listing 3-9. Using getHostname

```
# Example-9.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

host = conn.getHostname()
print('Hostname:'+host)

conn.close()
exit(0)
```

getMaxVcpus

The getMaxVcpus method can be used to obtain the maximum number of virtual CPUs per guest that the underlying virtualization technology supports. It takes a virtualization "type" as input (which can be None) and, if successful, returns the number of virtual CPUs supported. If an error occurs, -1 is returned instead. Listing 3-10 demonstrates the use of `getMaxVcpus`.

Listing 3-10. Using getMaxVcpus

```
# Example-10.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

vcpus = conn.getMaxVcpus(None)
print('Maximum support virtual CPUs: '+str(vcpus))

conn.close()
exit(0)
```

getInfo

The `getInfo` method can be used to obtain various information about the virtualization host. The method returns a Python `list` if successful and None if an error occurs. Table 3-5 lists the Python `list` members.

***Table 3-5.** virNodeInfo Structure Members*

Member	Description
`list[0]`	String indicating the CPU model.
`list[1]`	Memory size in MiB.
`list[2]`	Number of active CPUs.
`list[3]`	Expected CPU frequency (MHz).
`list[4]`	Number of NUMA nodes, 1 for uniform memory access. Nonuniform memory access (NUMA) is a method of configuring a cluster of microprocessors in a multiprocessing system so that they can share memory locally.
`list[5]`	Number of CPU sockets per node.
`list[6]`	Number of cores per socket.
`list[7]`	Number of threads per core.

Listing 3-11 demonstrates the use of `GetInfo`.

Listing 3-11. Using getInfo

```
# Example-12.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

nodeinfo = conn.getInfo()
```

```
print('Model: '+str(nodeinfo[0]))
print('Memory size: '+str(nodeinfo[1])+'MB')
print('Number of CPUs: '+str(nodeinfo[2]))
print('MHz of CPUs: '+str(nodeinfo[3]))
print('Number of NUMA nodes: '+str(nodeinfo[4]))
print('Number of CPU sockets: '+str(nodeinfo[5]))
print('Number of CPU cores per socket: '+str(nodeinfo[6]))
print('Number of CPU threads per core: '+str(nodeinfo[7]))

conn.close()
exit(0)
```

getCellsFreeMemory

The getCellsFreeMemory method can be used to obtain the amount of free memory (bytes) in some or all of the NUMA nodes in the system. It takes as input the starting cell and the maximum number of cells to retrieve data from. If successful, a Python list is returned with the amount of free memory in each node. On failure, None is returned. Listing 3-12 demonstrates the use of getCellsFreeMemory.

Listing 3-12. Using getCellsFreeMemory

```
# Example-13.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
nodeinfo = conn.getInfo()
numnodes = nodeinfo[4]

memlist = conn.getCellsFreeMemory(0, numnodes)
cell = 0
for cellfreemem in memlist:
    print('Node '+str(cell)+': '+str(cellfreemem)+' bytes free
    memory')
    cell += 1

conn.close()
exit(0)
```

The function will return the amount of available memory for each node in the system. If the system has no NUMA nodes, then it will return a single node and all the available memory. An example of a single node return is shown here:

```
Node 0: 13226958848 bytes free memory
```

getType

The getType method can be used to obtain the type of virtualization in use on this connection. If successful, it returns a string representing the type of virtualization in use. If an error occurs, None will be returned instead. Listing 3-13 demonstrates the use of getType.

Listing 3-13. Using virConnectGetType

```
# Example-14.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
```

```
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

print('Virtualization type: '+conn.getType())

conn.close()
exit(0)
```

getVersion

If no `name` parameter is passed (or `name` is `None`), then the version of the libvirt library is returned as an integer. If a name is passed and it refers to a driver linked to the libvirt library, then this returns a tuple of `(library version, driver version)`. The returned name is merely the driver name; for example, both KVM and QEMU guests are serviced by the driver for the `qemu://` URI, so a return of `QEMU` does not indicate whether KVM acceleration is present (Listing 3-14).

If the name passed refers to a nonexistent driver, then you will get the exception "no support for hypervisor." Versions numbers are integers: 1000000*major + 1000*minor + release.

Listing 3-14. Using virConnectGetVersion

```
# Example-15.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
ver = conn.getVersion()
print('Version: '+str(ver))

conn.close()
exit(0)
```

getLibVersion

The `getLibVersion` method can be used to obtain the version of the libvirt software in use on the host. If successful, it returns a Python string with the version; otherwise, it returns `None`. If the version is returned, it is in the format `1000000*version`. Listing 3-15 demonstrates the use of `getLibVersion`.

Listing 3-15. Using virConnectGetLibVersion

```
# Example-16.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

ver = conn.getLibVersion();
print('Libvirt Version: '+str(ver));

conn.close()
exit(0)
```

getURI

The getURI method can be used to obtain the URI for the current connection. While this is typically the same string that was passed into the open call, the underlying driver can sometimes canonicalize the string. This method will return the canonical version. If successful, it returns a URI string. If an error occurred, None will be returned instead. Listing 3-16 demonstrates the use of getURI.

Listing 3-16. Using virConnectGetURI

```
# Example-17.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

uri = conn.getURI()
print('Canonical URI: '+uri)

conn.close()
exit(0)
```

isEncrypted

The isEncrypted method can be used to find out whether a given connection is encrypted. If successful, it returns 1 for an encrypted connection and 0 for an unencrypted connection. If an error occurs, -1 will be returned. Listing 3-17 demonstrates the use of isEncrypted.

Listing 3-17. Using virConnectIsEncrypted

```
# Example-18.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

print('Connection is encrypted: '+str(conn.isEncrypted()))

conn.close()
exit(0)
```

isSecure

The `isSecure` method can be used to find out whether a given connection is encrypted. A connection will be classified secure if it is either encrypted or running on a channel that is not vulnerable to eavesdropping (like a UNIX domain socket). If successful, it returns 1 for a secure connection and 0 for an insecure connection. If an error occurs, -1 will be returned. Listing 3-18 demonstrates the use of `isSecure`.

Listing 3-18. Using virConnectIsSecure

```
# Example-19.py
from __future__ import print_function
import sys
import libvirt
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

print('Connection is secure: '+str(conn.isSecure()))

conn.close()
exit(0)
```

isAlive

This method determines whether the connection to the hypervisor is still alive. A connection will be classed as alive if it is either local or running over a channel (TCP or UNIX socket) that is not closed (Listing 3-19).

Listing 3-19. Using isAlive

```
# Example-21.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

alive = conn.isAlive()

print("Connection is alive = " + str(alive))

conn.close()
exit(0)
```

compareCPU

This method compares the given CPU description with the host CPU. This XML description argument is the same as used in the XML description for domain descriptions (Listing 3-20).

Listing 3-20. Using compareCPU

```
# Example-22.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

xml = '<cpu mode="custom" match="exact">' + \
        '<model fallback="forbid">kvm64</model>' + \
      '</cpu>'

retc = conn.compareCPU(xml)

if retc == libvirt.VIR_CPU_COMPARE_ERROR:
    print("CPUs are not the same or ther was error.")
elif retc == libvirt.VIR_CPU_COMPARE_INCOMPATIBLE:
    print("CPUs are incompatible.")
elif retc == libvirt.VIR_CPU_COMPARE_IDENTICAL:
    print("CPUs are identical.")
elif retc == libvirt.VIR_CPU_COMPARE_SUPERSET:
    print("The host CPU is better than the one specified.")
else:
    print("An Unknown return code was emitted.")

conn.close()
exit(0)
```

getFreeMemory

This method compares the given CPU description with the host CPU (Listing 3-21). Note that most libvirt APIs provide memory sizes in bytes, but in this function the returned value is in bytes.

Listing 3-21. Using getFreeMemory

```
# Example-23.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

mem = conn.getFreeMemory()

print("Free memory on the node (host) is " + str(mem) + " bytes.")

conn.close()
exit(0)
```

getFreePages

This method queries the host system for free pages of a specified size. For the input, the pages argument is a Python `list` of page sizes that the caller is interested in (the size unit is kilobytes, so, for example, pass 2048 for 2 MiB), the `start` argument refers to the first NUMA node that information should be collected from, and the `cellcount` argument tells how many consecutive nodes should be queried. The function returns a Python `list` containing an indicator of whether pages of the specified input sizes are available. An error will be thrown if the host system does not support memory pages of the size requested (Listing 3-22).

Listing 3-22. Using getFreePages

```
# Example-24.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

pages = [2048]
start = 0
cellcount = 4
buf = conn.getFreePages(pages, start, cellcount)

i = 0
for page in buf:
    print("Page Size: " + str(pages[i]) + \
          " Available pages: " + str(page))
    ++i
```

```
conn.close()
exit(0)
```

The following is an example of the output from Listing 3-22.

```
Page Size: 2048 Available pages: 0
```

getMemoryParameters

This method returns all the available memory parameters as strings (Listing 3-23).

Listing 3-23. Using getFreePages

```
# Example-25.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

buf = conn.getMemoryParameters()

print('shm_pages_shared        : ' + str(buf['shm_pages_shared']))
print('shm_full_scans          : ' + str(buf['shm_full_scans']))
print('shm_merge_across_nodes  : ' + str(buf['shm_merge_across_
                                    nodes']))
print('shm_pages_to_scan       : ' + str(buf['shm_pages_to_scan']))
print('shm_pages_unshared      : ' + str(buf['shm_pages_unshared']))
print('shm_sleep_millisecs     : ' + str(buf['shm_sleep_millisecs']))
       print('shm_pages_sharing       : ' + str(buf['shm_pages_sharing']))
       print('shm_pages_volatile      : ' + str(buf['shm_pages_volatile']))

       conn.close()
       exit(0)
```

getMemoryStats

This method returns the memory statistics for a single node (host). It returns a Python `list` of strings (Listing 3-24).

Listing 3-24. Using getMemoryStats

```
# Example-26.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

buf = conn.getMemoryStats(libvirt.VIR_NODE_MEMORY_STATS_ALL_CELLS)

print('cached : ' + str(buf['cached']))
print('total  : ' + str(buf['total']))
print('buffers: ' + str(buf['buffers']))
print('total  : ' + str(buf['total']))

conn.close()
exit(0)
```

getSecurityModel

In Listing 3-25, this method returns the security model (as a `list`) currently in use (if any).

Listing 3-25. Using getSecurityModel

```
# Example-27.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

model = conn.getSecurityModel()

print(model[0] + " " + model[1])

conn.close()
exit(0)
```

For most x86_64 Linux systems, this will be `SELinux`. For other architecture, the value may be different.

getSysinfo

This method returns the system information in the form of an XML definition. The format is the same as for a domain XML definition (Listing 3-26).

Listing 3-26. Using getSysinfo

```
# Example-28.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

xmlInfo = conn.getSysinfo()

print(xmlInfo)

conn.close()
exit(0)
```

getCPUMap

This method is used to get a CPU map of host node CPUs (Listing 3-27).

Listing 3-27. Using getCPUMap

```
# Example-29.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
map = conn.getCPUMap()

print("CPUs: " + str(map[0]))
print("Available: " + str(map[1]))

conn.close()
exit(0)
```

getCPUStats

This method is used to get the stats for one or all CPUs. This method requires a single argument that represents the CPU number to fetch the statistics for a single CPU or the value VIR_NODE_CPU_STATS_ALL_CPUS to fetch a Python list of statistics for all CPUs (Listing 3-28).

Listing 3-28. Using getCPUStats

```
# Example-30.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

stats = conn.getCPUStats(0)

print("kernel: " + str(stats['kernel']))
print("idle:   " + str(stats['idle']))
print("user:   " + str(stats['user']))
print("iowait: " + str(stats['iowait']))

conn.close()
exit(0)
```

getCPUModelNames

This method is used to get the list of CPU names that match an architecture type (Listing 3-29).

Listing 3-29. Using getCPUModelNames

```
# Example-31.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

models = conn.getCPUModelNames('x86_64')

for model in models:
    print(model)

conn.close()
exit(0)
```

Summary

This chapter presented the connection functions and covered how to connect and how to return information from the main host system. The connectivity functions include opening and closing connections. The informational functions include information about the host including CPU, memory, security, and other relevant host hardware information.

CHAPTER 4

Guest Domains

A *domain* is a reference to a running or nonrunning virtual operating system under libvirt. As many domains as necessary can be created/installed under libvirt up to the limit imposed by either the system administrator or the number that can be supported without performance penalties on the host system. This number will vary depending on the performance abilities of the host system.

A nonrunning domain does not add any performance penalties on the host machine; only running domains impose penalties on the host system. Therefore, the system administrator can impose a limit on the number of running domains instead of the number of total domains.

Domain Overview

As mentioned in Chapter 2, a domain is an instance of an operating system running on a virtualized machine. The connection object provides methods to enumerate the guest domains, create new guest domains, and manage existing domains. A guest domain is represented with an instance of the `virDomain` class and has a number of unique identifiers.

- **ID**: This is a positive integer, unique among the running guest domains on a single host. An inactive domain does not have an ID.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_4

- **Name**: This is a short string, unique among all the guest domains on a single host, both running and inactive. To ensure maximum portability between hypervisors, it is recommended that names include only alphanumeric (`a–Z`, `0–9`), hyphen (`-`), and underscore (`_`) characters.
- **UUID**: This consists of 16 unsigned bytes, guaranteed to be unique among all guest domains on any host. RFC 4122 defines the format for UUIDs and provides a recommended algorithm for generating UUIDs with guaranteed uniqueness.

A guest domain can be transient or persistent. A transient guest domain can be managed only while it is running on the host. Once it is powered off, all traces of it will disappear. A persistent guest domain has its configuration maintained in a data store on the host by the hypervisor, in an implementation-defined format. Thus, when a persistent guest is powered off, it is still possible to manage its inactive configuration. A transient guest can be turned into a persistent guest while it is running by defining a configuration for it.

Once an application has a unique identifier for a domain, it will often want to obtain the corresponding `virDomain` object. There are three methods to look up existing domains: `lookupByID`, `lookupByName`, and `lookupByUUID`. Each of these takes the domain identifier as a parameter. They will return `None` if no matching domain exists. The error object can be queried to find specific details of the error if required. Listing 4-1 shows an example of how to query that error.

Listing 4-1. Fetching a Domain Object from an ID

```
# Example-1.py
from __future__ import print_function
import sys
import libvirt
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainID = 6
dom = conn.lookupByID(domainID)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)

conn.close()
exit(0)
```

Note Listing 4-1 may abend if `domainid` is not found because it is possible for libvirt to abort the process rather than return an error to Python.

It should be noted that the `lookupByID` method will not work if the domain is not active. Inactive domains all have an ID of `-1`. Listing 4-2 shows how to fetch a domain using its libvirt name.

Listing 4-2. Fetching a Domain Object from a Name

```
# Example-2.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
domainName = 'someguest'
dom = conn.lookupByName(domainName)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)

conn.close()
```

Note Listing 4-2 may abend if `domainName` is not found because it is possible for libvirt to abort the process rather than return an error to Python.

Listing 4-3 shows how to fetch a domain from the domain's UUID.

Listing 4-3. Fetching a Domain Object from a UUID

```
# Example-3.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainUUID = '00311636-7767-71d2-e94a-26e7b8bad250'
dom = conn.lookupByUUID(domainUUID)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)

conn.close()
exit(0)
```

Note Listing 4-3 may abend if `domainUUID` is not found because it is possible for libvirt to abort the process rather than return an error to Python.

The previous UUID example uses the printable format of UUID. Using the equivalent raw bytes is not supported by Python.

Listing Domains

The libvirt classes expose two lists of domains: the first contains running domains, while the second contains inactive, persistent domains. The lists are intended to be nonoverlapping, exclusive sets, though there is always a small possibility that a domain can stop or start in between the querying of each set. The events class described later in this section provides a way to track all lifecycle changes, avoiding this potential race condition.

The method for listing active domains returns a list of domain IDs. Every running domain has a positive integer ID, uniquely identifying it among all running domains on the host. The method for listing active domains, `listDomainsID`, requires no parameters. The return value will be `None` upon error or a Python `list` of the IDs expressed as `ints`. Listing 4-4 shows how to obtain a list of all running (active) domains.

Listing 4-4. Listing Active Domains

```
# Example-4.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainIDs = conn.listDomainsID()
if domainIDs == None:
    print('Failed to get a list of domain IDs', \
          file=sys.stderr)

print("Active domain IDs:")
if len(domainIDs) == 0:
    print('  None')
else:
    for domainID in domainIDs:
        print('  '+str(domainID))

conn.close()
exit(0)
```

In addition to the running domains, there may be some persistent inactive domain configurations stored on the host. Since an inactive domain does not have any ID identifier, the listing of inactive domains is exposed as a list of name strings. The return value will be `None` upon error, or a Python `list` of elements filled with names (strings). Listing 4-5 shows how to obtain a list of all inactive domains.

Listing 4-5. Listing Inactive Domains

```
# Example-5.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainNames = conn.listDefinedDomains()
if conn == None:
    print('Failed to get a list of domain names', file=sys.stderr)

domainIDs = conn.listDomainsID()
if domainIDs == None:
    print('Failed to get a list of domain IDs', file=sys.stderr)
if len(domainIDs) != 0:
    for domainID in domainIDs:
        domain = conn.lookupByID(domainID)
        domainNames.append(domain.name)

print("All (active and inactive domain names:")
if len(domainNames) == 0:
    print('  None')
else:
    for domainName in domainNames:
        print('  '+domainName)

conn.close()
exit(0)
```

The methods for listing domains do not directly return the virDomain objects since this may incur an undue performance penalty for applications that want to query the list of domains on a frequent basis. However, the Python libvirt module does provide the method listAllDomains, which returns all the domains, active or inactive. It returns a Python list of the virDomain instances or None upon an error. The list can be empty when no persistent domains exist.

The `listAllDomains` method takes a single parameter that is a flag specifying a filter for the domains to be listed. If a value of `0` is specified, then all domains will be listed. Otherwise, any or all of the following constants can be added together to create a filter for the domains to be listed:

`VIR_CONNECT_LIST_DOMAINS_ACTIVE`
`VIR_CONNECT_LIST_DOMAINS_INACTIVE`
`VIR_CONNECT_LIST_DOMAINS_PERSISTENT`
`VIR_CONNECT_LIST_DOMAINS_TRANSIENT`
`VIR_CONNECT_LIST_DOMAINS_RUNNING`
`VIR_CONNECT_LIST_DOMAINS_PAUSED`
`VIR_CONNECT_LIST_DOMAINS_SHUTOFF`
`VIR_CONNECT_LIST_DOMAINS_OTHER`
`VIR_CONNECT_LIST_DOMAINS_MANAGEDSAVE`
`VIR_CONNECT_LIST_DOMAINS_NO_MANAGEDSAVE`
`VIR_CONNECT_LIST_DOMAINS_AUTOSTART`
`VIR_CONNECT_LIST_DOMAINS_NO_AUTOSTART`
`VIR_CONNECT_LIST_DOMAINS_HAS_SNAPSHOT`
`VIR_CONNECT_LIST_DOMAINS_NO_SNAPSHOT`

Listing 4-6 shows how to obtain a list of active and inactive domainslistAllDomains method.

Listing 4-6. Fetching All Domain Objects

```
# Example-6.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

print("All (active and inactive) domain names:")
domains = conn.listAllDomains(0)
if len(domains) != 0:
    for domain in domains:
        print('  '+domain.name())
else:
    print('  None')

conn.close()
exit(0)
```

Obtaining State Information About a Domain

Once a domain instance has been obtained, it is possible to fetch information about the state of the domain, such as the type of OS being hosted, running state, ID, UUID, and so on. The following methods will demonstrate how to fetch this information.

Fetching the ID of a Domain

The ID of a domain can be obtained by using the ID method. Only running domains have an ID; fetching the ID of a nonrunning domain always returns -1. Listing 4-7 shows how to obtain the ID of a domain from the libvirt domain name.

Listing 4-7. Fetching the ID of a Domain

```
# Example-43.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

id = dom.ID()
if id == -1:
    print('The domain is not running so has no ID.')
else:
    print('The ID of the domain is ' + str(id))

conn.close()
exit(0)
```

Fetching the UUID of a Domain

The UUID of a domain can be obtained by using the `UUID` or `UUIDString` method. The `UUID` method is not all that useful for Python programs because it is a binary value. The `UUIDString` method is much more useful because it returns a formatted string value that can be easily parsed.

The UUID is not dependent on the running state of the domain and always returns a valid UUID. Listing 4-8 shows how to fetch the UUID of a domain using the domain name.

Listing 4-8. Fetching the UUID of a Domain

```
# Example-44.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

uuid = dom.UUIDString()
print('The UUID of the domain is ' + uuid)

conn.close()
exit(0)
```

Fetching the OS Type of a Domain

The type of OS hosted by a domain is also available. Only running domains have an ID; fetching the ID of a nonrunning domain always returns `-1`. This same information can be retrieved via the `info` method. Listing 4-9 shows how to fetch the operating system type from a running domain.

Listing 4-9. Fetching the ID of a Domain

```
# Example-45.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

type = dom.OSType()
print('The OS type of the domain is "' + type + '"')

conn.close()
exit(0)
```

Determining Whether the Domain Has a Current Snapshot

The `hasCurrentSnapshot` method returns a Boolean value indicating if a current snapshot is available. This method always returns a valid value and is not dependent on the running state of the domain. Listing 4-10 determines whether a domain has a current snapshot.

Listing 4-10. Determining Whether the Domain Has a Current Snapshot

```
# Example-47.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

flag = dom.hasCurrentSnapshot()
print('The value of the current snapshot flag is ' + str(flag))

conn.close()
exit(0)
```

Determining Whether the Domain Has Managed Save Images

The hasManagedSaveImages method returns a Boolean value indicating if a domain has a managed save image. A saved image is created on an as-needed basis by the user.

Note that a running domain should never have a saved image because that image should have been removed when the domain was restarted. Listing 4-11 determines whether the domain has a managed save image.

Listing 4-11. Determining Whether the Domain Has a Managed Save Image

```
# Example-48.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'


conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

flag = dom.hasManagedSaveImage()
```

```
print('The value of the manaed save images flag is ' +
str(flag))

conn.close()
exit(0)
```

Fetching the Hostname of the Domain

The `hostname` method returns the hostname of the domain. The `hostname` method is highly dependent on the hypervisor and/or the `qemu-guest-agent`. It may throw an error if the method cannot complete successfully. Listing 4-12 will return the hostname of the domain (if available).

Listing 4-12. Fetch the Hostname of the Domain

```
# Example-49.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)
```

```
name = dom.hostname()
print('The hostname of the domain  is ' + str(name))

conn.close()
exit(0)
```

Getting the Domain Hardware Information

The `info` method returns some general information about the domain hardware. There should be five entries returned in a Python `list`: the state, max memory, memory, CPUs, and CPU time for the domain. Listing 4-13 obtains general information from a domain.

Listing 4-13. Get the Domain Info

```
# Example-50.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)
```

```
state, maxmem, mem, cpus, cput = dom.info()
print('The state is ' + str(state))
print('The max memory is ' + str(maxmem))
print('The memory is ' + str(mem))
print('The number of cpus is ' + str(cpus))
print('The cpu time is ' + str(cput))

conn.close()
exit(0)
```

Determining Whether the Domain Is Running

The `isActive` method returns a Boolean flag indicating whether the domain is active (running). Listing 4-14 returns a Boolean indicating the running state of the domain.

Listing 4-14. Determine Whether the Domain Is Running

```
# Example-51.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

flag = dom.isActive()
if flag == True:
    print('The domain is active.')
else:
    print('The domain is not active.')

conn.close()
exit(0)
```

Determining Whether the Domain Is Persistent

The `isPersistent` method returns a Boolean flag indicating whether or not the domain is persistent (the domain will be persistent after a reboot). Listing 4-15 determines whether a running domain is persistent.

Listing 4-15. Determine Whether the Domain Is Persistent

```
# Example-52.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

flag = dom.isPersistent()
if flag == 1:
    print('The domain is persistent.')
elif flag == 0:
    print('The domain is not persistent.')
else:
    print('There was an error.')

conn.close()
exit(0)
```

Determining Whether the Domain Is Updated

The `isUpdated` method returns a Boolean flag indicating whether the domain has been updated since it was created. An update to the domain can be just about any change to the domain's configuration. Listing 4-16 returns a flag indicating the domain's change state.

Listing 4-16. Determine Whether the Domain Is Updated

```
# Example-59.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
```

```
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

flag = dom.isUpdated()
if flag == 1:
    print('The domain is updated.')
elif flag == 0:
    print('The domain is not updated.')
else:
    print('There was an error.')

conn.close()
exit(0)
```

Determining the Max Memory of the Domain

The `maxMemory` method returns the maximum memory allocated to the domain. This same information may be retrieved via the `info` method. Listing 4-17 returns the max memory allocated to the domain.

Listing 4-17. Determine the Max Memory of the Domain

```
# Example-53.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom
```

```
domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

mem = dom.maxMemory()
if mem > 0:
    print('The max memory for domain is ' + str(mem) + 'MB')
else:
    print('There was an error.')

conn.close()
exit(0)
```

Determining the Max VCPUs of the Domain

The `maxVcpus` method returns the maximum number of virtual CPUs allocated to the domain. This same information can be retrieved via the `info` method. This works only on active domains. Listing 4-18 returns the max VCPUs allocated to the domain.

Listing 4-18. Determine the Max VCPUs of the Domain

```
# Example-54.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, \
          file=sys.stderr)
    exit(1)

cpus = dom.maxVcpus()
if cpus != -1:
    print('The max Vcpus for domain is ' + str(cpus))
else:
    print('There was an error.')

conn.close()
exit(0)
```

Fetching the Name of the Domain

The name method returns the name of the domain. Listing 4-19 returns the name of a domain.

Listing 4-19. Fetch the Name of the Domain

```
# Example-55.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

name = dom.name()
print('Thename of the domain is "' + name +'".')

conn.close()
exit(0)
```

Fetching the State of the Domain

The state method returns the state of the domain. Listing 4-20 returns the state of the domain.

Listing 4-20. Fetch the State of the Domain

```
# Example-56.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

state, reason = dom.state()

if state == libvirt.VIR_DOMAIN_NOSTATE:
    print('The state is VIR_DOMAIN_NOSTATE')
elif state == libvirt.VIR_DOMAIN_RUNNING:
    print('The state is VIR_DOMAIN_RUNNING')
elif state == libvirt.VIR_DOMAIN_BLOCKED:
    print('The state is VIR_DOMAIN_BLOCKED')
```

```
elif state == libvirt.VIR_DOMAIN_PAUSED:
    print('The state is VIR_DOMAIN_PAUSED')
elif state == libvirt.VIR_DOMAIN_SHUTDOWN:
    print('The state is VIR_DOMAIN_SHUTDOWN')
elif state == libvirt.VIR_DOMAIN_SHUTOFF:
    print('The state is VIR_DOMAIN_SHUTOFF')
elif state == libvirt.VIR_DOMAIN_CRASHED:
    print('The state is VIR_DOMAIN_CRASHED')
elif state == libvirt.VIR_DOMAIN_PMSUSPENDED:
    print('The state is VIR_DOMAIN_PMSUSPENDED')
else:
    print(' The state is unknown.')
print('The reason code is ' + str(reason))

conn.close()
exit(0)
```

Extracting the Time Information from the Domain

The `getTime` method extracts the current timestamp from the domain. The method returns the same value as the Python `time.struct_time` function. Listing 4-21 will fetch the current time as known by the domain.

Listing 4-21. Fetch the Time Information of the Domain

```
# Example-57.py
from __future__ import print_function
import sys, time
import libvirt
from xml.dom import minidom

domName = 'CentOS7'
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

struct = dom.getTime()
timestamp = time.ctime(float(struct['seconds']))
print('The domain current time is ' + timestamp)

conn.close()
exit(0)
```

Extracting the Network Interface Addresses from a Domain

The `interfaceAddresses` method extracts the current network interface address. This information is dynamic in nature as it can be modified by a guest domain system administrator. Thus, you should not depend on this information remaining static. The QEMU guest agent is queried by the libvirt library to obtain this information, and thus the agent must be running on the guest domain for any information to be returned.

Listing 4-22 will return the list of IP addresses assigned to an interface.

Listing 4-22. Fetch the Network Interface IP Addresses of the Domain

```
# Example-58.py
from __future__ import print_function
import libvirt
```

```
# setup
"""List network interface IP Addresses for a guest domain.
"""

domName = 'CentOS7'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domain = conn.lookupByName(domName)
if domain == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

# make sure the domain is running
if domain.isActive() == False:
    print('Error: Domain is not active or never started.')
    exit(1)

# get and list the ip addresses, use QEMU as the source
information
ifaces = domain.interfaceAddresses(libvirt. \
                VIR_DOMAIN_INTERFACE_ADDRESSES_SRC_AGENT, 0)
for (name, val) in ifaces.iteritems():
    if val['addrs']:
        for ipaddr in val['addrs']:
            print(name+' '+str(ipaddr))

conn.close()
exit(0)
```

Note This program will fail if the QEMU guest agent does not support this function.

Lifecycle Control

libvirt can control the entire lifecycle of guest domains (see Figure 4-1). Guest domains transition through several states throughout their lifecycle.

1. `Undefined`: This is the baseline state. An undefined guest domain has not been defined or created in any way.
2. `Defined`: A defined guest domain has been defined but is not running. This state could also be described as `Stopped`.
3. `Running`: A running guest domain is defined and being executed on a hypervisor.
4. `Paused`: A paused guest domain is in a suspended state from the `Running` state. Its memory image has been temporarily stored, and it can be resumed to the `Running` state without the guest domain operating system being aware it was ever suspended.
5. `Saved`: A saved domain has had its memory image, as captured in the `Paused` state, saved to persistent storage. It can be restored to the `Running` state without the guest domain operating system being aware it was ever suspended.

The transitions between these states fall into several categories; see the following sections for more details.

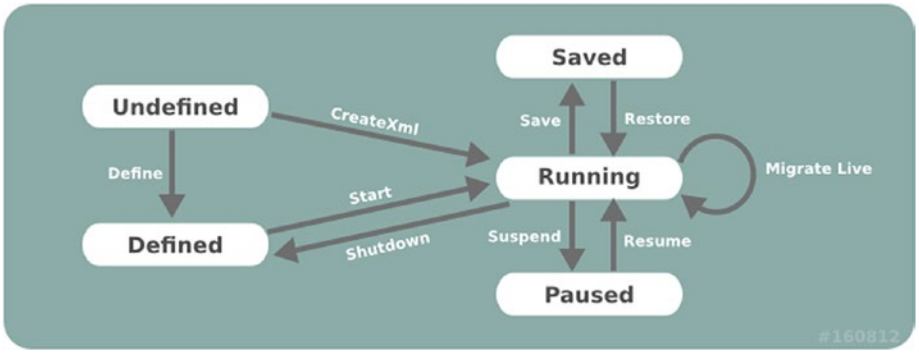

Figure 4-1. *Guest domain lifecycle*

Provisioning and Starting

Provisioning refers to the task of creating new guest domains, typically using some form of operating system installation media. There are a wide variety of ways in which a guest can be provisioned, but the choices available will vary according to the hypervisor and type of guest domain being provisioned. It is not uncommon for an application to support several different provisioning methods. *Starting* refers to executing a provisioned guest domain on a hypervisor.

Methods for Provisioning

There are up to three methods involved in provisioning guests. The `createXML` method will create and immediately boot a new transient guest domain. When this guest domain shuts down, all traces of it will disappear. The `defineXML` method will store the configuration for a persistent guest domain. The `create` method will boot a previously defined guest domain from its persistent configuration. One important thing to note is that the `defineXML` command can be used to turn a previously booted transient guest domain into a persistent domain. This can be useful for some provisioning scenarios that will be illustrated later.

Booting a Transient Guest Domain

Booting a transient guest domain simply requires a connection to libvirt and a string containing the XML document describing the required guest configuration and a flag that controls the startup of the domain.

If the `VIR_DOMAIN_START_PAUSED` flag is set, the guest domain will be started, but its CPUs will remain paused. The CPUs can later be manually started using the `resume` method.

If the `VIR_DOMAIN_START_AUTODESTROY` flag is set, the guest domain will be automatically destroyed when the `virConnect` object is finally released. This will also happen if the client application crashes or loses its connection to the libvirtd daemon. Any domains marked for autodestroy will block attempts at migrating domains, saving to a file, or taking snapshots.

Listing 4-23 shows how to create a domain from an existing XML file. XML files will be covered later in the chapter.

Listing 4-23. Provisioning a Transient Guest Domain

```
# Example-7.py
from __future__ import print_function
import sys
import libvirt

xmlconfig = '<domain>........</domain>'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.createXML(xmlconfig, 0)
if dom == None:
    print('Failed to create a domain from an XML definition.', \
          file=sys.stderr)
    exit(1)

print('Guest '+dom.name()+' has booted', file=sys.stderr)

conn.close()
exit(0)
```

Note Listing 4-23 is just an example, and the XML does not exist in the provided code examples.

If the domain creation attempt succeeded, then the returned `virDomain` instance will be returned; otherwise, `None` will be returned. Although the domain was booted successfully, this does not guarantee that the domain is still running. It is entirely possible for the guest domain to crash, in which case attempts to use the returned `virDomain` object will generate an error since transient guests cease to exist when they are shut down (whether via a planned shutdown or a crash). Managing this scenario requires use of a persistent guest.

Defining and Booting a Persistent Guest Domain

Before a persistent domain can be booted, it must have its configuration defined. This again requires a connection to libvirt and a string containing the XML document describing the required guest configuration. The `virDomain` object obtained from defining the guest can then be used to boot it.

Currently the `defineDomain` method defines a `flags` parameter that is unused. A value of `0` should always be supplied for that parameter. This may be changed in later versions of the method. Listing 4-24 is a sample of how to create a domain. It is not a complete program as the XML for the program does not exist.

Listing 4-24. Defining and Booting a Persistent Guest Domain

```
# Example-8.py
from __future__ import print_function
import sys
import libvirt

xmlconfig = '<domain>........</domain>'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.defineXML(xmlconfig, 0)
if dom == None:
    print('Failed to define a domain from an XML definition.', \
          file=sys.stderr)
    exit(1)

if dom.create(dom) < 0:
    print('Can not boot guest domain.', file=sys.stderr)
    exit(1)

print('Guest '+dom.name()+' has booted', file=sys.stderr)

conn.close()
exit(0)
```

New Guest Domain Provisioning Techniques

This section will first illustrate two configurations that allow for a provisioning approach that is comparable to those used for physical machines. It then outlines a third option that is specific to virtualized hardware but has some interesting benefits. For the purposes of illustration, the examples that follow will use an XML configuration that sets up a KVM fully virtualized guest, with a single disk and network interface and a video card using VNC for display.

The following is a sample XML configuration. Note that the `demo.img` file does not exist.

```
<domain type='kvm'>
  <name>demo</name>
  <os>
    <type arch='i686' machine='pc'>hvm</type>
    <boot> dec='hd'/>
  </os>
  <uuid>c7a5fdbd-cdaf-9455-926a-d65c16db1809</uuid>
  <memory>500000</memory>
  <vcpu>1</vcpu>
  .... the <os> block will vary per approach ...
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <source file='/var/lib/libvirt/images/demo.img'/>
      <driver name='qemu' type='raw'/>
      <target dev='hda'/>
    </disk>
    <interface type='bridge'>
      <mac address='52:54:00:d8:65:c9'/>
      <source bridge='br0'/>
    </interface>
    <input type='mouse' bus='ps2'/>
    <graphics type='vnc' port='-1' listen='127.0.0.1'/>
  </devices>
</domain>
```

Important Be careful when choosing initial memory allocation. Too low of a value may cause mysterious crashes and installation failures. Some operating systems need as much as 600 MB of memory for initial installation, though this can often be reduced post-install.

CD-ROM/ISO Image Provisioning

All full virtualization technologies have support for emulating a CD-ROM device in a guest domain, making this an obvious choice for provisioning new guest domains. It is, however, fairly rare to find a hypervisor that provides CD-ROM devices for paravirtualized guests.

The first obvious change required to the XML configuration to support CD-ROM installation is to add a CD-ROM device. A guest domain's CD-ROM device can be pointed either to a host CD-ROM device or to an ISO image file. The next change is to determine what the BIOS boot order should be, with there being two possible options. If the hard disk is listed ahead of the CD-ROM device, then the CD-ROM media won't be booted unless the first boot sector on the hard disk is blank. If the CD-ROM device is listed ahead of the hard disk, then it will be necessary to alter the guest config after install to make it boot off the installed disk. While both can be made to work, the first option is easiest to implement.

The guest configuration shown earlier will have the following XML chunk inserted:

```
<os>
  <type arch='x86_64' machine='pc'>hvm</type>
  <boot dev='hd'/>
  <boot dev='cdrom'/>
</os>
```

This assumes the hard disk boot sector is blank initially so that the first boot attempt falls through to the CD-ROM drive. It will also need a CD-ROM drive device added.

```
<disk type='file' device='cdrom'>
  <source file='/var/lib/libvirt/images/rhel5-x86_64-dvd.iso'/>
  <target dev='hdc' bus='ide'/>
</disk>
```

With the configuration determined, it is now possible to provision the guest. This is an easy process, simply requiring a persistent guest to be defined and then booted. Listing 4-25 shows how to define and boot a domain. Note that the XML is missing, so this is just a sample program and will not run properly.

Listing 4-25. Defining and Booting a Persistent Guest Domain

```
# Example-12.py
from __future__ import print_function
import sys
import libvirt

xmlconfig = '<domain>........</domain>'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
dom = conn.defineXML(xmlconfig, 0)
if dom == None:
    print('Failed to define a domain from an XML definition.', \
          file=sys.stderr)
    exit(1)

if dom.create(dom) < 0:
    print('Can not boot guest domain.', file=sys.stderr)
    exit(1)

print('Guest '+dom.name()+' has booted', file=sys.stderr)

conn.close()
exit(0)
```

If it had not been possible to guarantee that the boot sector of the hard disk was blank, then provisioning would have been a two-step process. First a transient guest would have been booted using a CD-ROM drive as the primary boot device. Once that was completed, then a persistent configuration for the guest would be defined to boot off the hard disk.

In addition to the `defineXML` method, the alternative method `defineXMLFlags` is available.

PXE Boot Provisioning

Some newer full virtualization technologies provide a BIOS that is able to use the PXE boot protocol to boot off the network. If an environment already has a PXE boot provisioning server deployed, this is a desirable method to use for guest domains.

PXE booting a guest obviously requires that the guest has a network device configured. The LAN that this network card is attached to also needs a PXE/TFTP server available. The next change is to determine what the BIOS boot order should be, with there being two possible options. If the hard disk is listed ahead of the network device, then the network card won't PXE boot unless the first boot sector on the hard disk is blank. If the network device is listed ahead of the hard disk, then it will be necessary to alter the guest config after install to make it boot off the installed disk. While both can be made to work, the first option is easiest to implement.

The guest configuration shown earlier will have the following XML chunk inserted:

```
<os>
  <type arch='x86_64' machine='pc'>hvm</type>
  <boot dev='hd'/>
  <boot dev='network'/>
</os>
```

This assumes the hard disk boot sector is blank initially, so the first boot attempt falls through to the NIC. With the configuration determined, it is now possible to provision the guest. This is an easy process, simply requiring a persistent guest to be defined and then booted.

Listing 4-26 shows how to define a PXE domain and boot it. Note that the XML is missing, so this is just a sample program and will not run properly.

Listing 4-26. PXE Boot Provisioning

```
# Example-14.py
from __future__ import print_function
import sys
import libvirt

xmlconfig = '<domain>........</domain>'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
dom = conn.defineXML(xmlconfig, 0)
if dom == None:
    print('Failed to define a domain from an XML definition.', \
          file=sys.stderr)
    exit(1)

if dom.create(dom) < 0:
    print('Can not boot guest domain.', file=sys.stderr)
    exit(1)

print('Guest '+dom.name()+' has booted', file=sys.stderr)

conn.close()
exit(0)
```

If it had not been possible to guarantee that the boot sector of the hard disk was blank, then provisioning would have been a two-step process. First a transient guest would have been booted using a network as the primary boot device. Once that was completed, then a persistent configuration for the guest would be defined to boot off the hard disk.

Direct Kernel Boot Provisioning

Paravirtualization technologies emulate a fairly restrictive set of hardware, often making it impossible to use the provisioning options just outlined. For such scenarios, it is often possible to boot a new guest domain directly from a kernel and initrd image stored on the host file system. This has one interesting advantage, which is that it is possible to directly set kernel command-line boot arguments, making it easy to carry out a fully automated installation. This advantage can be compelling enough that this technique is used even for fully virtualized guest domains with CD-ROM drive/PXE support.

The one complication with direct kernel booting is that provisioning becomes a two-step process. For the first step, it is necessary to configure the guest XML configuration to point to a kernel/initrd. Listing 4-27 shows how to specify a kernel boot configuration.

Listing 4-27. Kernel Boot Provisioning XML

```
<os>
  <type arch='x86_64' machine='pc'>hvm</type>
  <kernel>/var/lib/libvirt/boot/f11-x86_64-vmlinuz</kernel>
  <initrd>/var/lib/libvirt/boot/f11-x86_64-initrd.img</initrd>
  <cmdline>method=http://download.fedoraproject.org/pub/fedora/
  linux/releases/11/x86_64/os console=ttyS0 console=tty</cmdline>
</os>
```

Notice how the kernel command line provides the URL of the download site containing the distro install tree matching the kernel/initrd. This allows the installer to automatically download all its resources without prompting the user for an install URL. It could also be used to provide a kickstart file for a completely unattended installation. Finally, this command line also tells the kernel to activate both the first serial port and the VGA card as consoles, with the latter being the default. Having kernel messages duplicated on the serial port in this manner can be a useful debugging avenue. Of course, valid command-line arguments vary according to the particular kernel being booted. Consult the kernel vendor/distributor's documentation for valid options.

The last XML configuration detail before starting the guest is to change the `on_reboot` element action to `destroy`. This ensures that when the guest installer finishes and requests a reboot, the guest is instead powered off. This allows the management application to change the configuration to make it boot from the just installed hard disk again. The provisioning process can be started now by creating a transient guest with the first XML configuration.

Listing 4-28 shows how to create a kernel boot domain. Note that the XML is missing, so the program will not work until the XML is supplied.

Listing 4-28. Kernel Boot Provisioning

```
# Listing-14.py
from __future__ import print_function
import sys
import libvirt

xmlconfig = '<domain>........</domain>'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.createXML(xmlconfig, 0)
if dom == None:
    print('Unable to boot transient guest configuration.', \
          file=sys.stderr)
    exit(1)

print('Guest '+dom.name()+' has booted', file=sys.stderr)

conn.close()
exit(0)
```

Once this guest shuts down, the second phase of the provisioning process can be started. For this phase, the `OS` element will have the `kernel/initrd/cmdline` elements removed and replaced by either a reference to a host-side bootloader or a BIOS boot setup. The former is used for Xen paravirtualized guests, while the latter is used for fully virtualized guests.

The phase 2 configuration for a Xen paravirtualized guest would thus look like this:

```
<bootloader>/usr/bin/pygrub</bootloader>
<os>
  <type arch='x86_64' machine='pc'>xen</type>
</os>
```

A fully virtualized guest would use this:

```
<bootloader>/usr/bin/pygrub</bootloader>
<os>
  <type arch='x86_64' machine='pc'>hvm</type>
  <boot dev='hd'/>
</os>
```

With the second phase configuration determined, the guest can be re-created, this time using a persistent configuration. Listing 4-29 shows how to create a persistent kernel boot domain.

Listing 4-29. Kernel Boot Provisioning for a Persistent Guest Domain

```
# Example-18.py
from __future__ import print_function
import sys
import libvirt

xmlconfig = '<domain>........</domain>'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
dom = conn.createXML(xmlconfig, 0)
if dom == None:
    print('Unable to define persistent guest configuration.', \
          file=sys.stderr)
    exit(1)

if dom.create(dom) < 0:
    print('Can not boot guest domain.', file=sys.stderr)
    exit(1)

print('Guest '+dom.name()+' has booted', file=sys.stderr)

conn.close()
exit(0)
```

In addition to the `createXML` method, the alternative method `createXMLFlags` is available.

Stopping

Stopping refers to the process of halting a running guest. A guest can be stopped by two methods: `shutdown` and `destroy`.

The `shutdown` method is a clean stop process, which sends a signal to the guest domain operating system asking it to shut down immediately. The guest will be stopped only once the operating system has successfully shut down. The `shutdown` process is analogous to running a `shutdown` command on a physical machine. There is also a `shutdownFlags` method that can, depending on what the guest OS supports, shut down the domain and leave the object in a usable state.

The `destroy` and `destroyFlags` methods immediately terminate the guest domain. The `destroy` process is analogous to pulling the plug on a physical machine.

Suspending/Resuming and Saving/Restoring

The `suspend` and `resume` methods refer to the process of taking a running guest and temporarily saving its memory state. At a later time, it is possible to resume the guest to its original running state, continuingly executing where it left off. Suspend does not save a persistent image of the guest's memory. For this, `save` is used.

The `save` and `restore` methods refer to the process of taking a running guest and saving its memory state to a file. At some time later, it is possible to restore the guest to its original running state, continuing execution where it left off.

It is important to note that the save/restore methods save only the memory state; no storage state is preserved. Thus, when the guest is restored, the underlying guest storage must be in the same state as it was when the guest was initially saved. For basic usage, this implies that a guest can be restored only once from any given saved state image. To allow a guest to be restored from the same saved state multiple times, the application must also have taken a snapshot of the guest storage at the time of saving and explicitly revert to this storage snapshot when restoring. A future enhancement in libvirt will allow for an automated snapshot capability that saves memory and storage state in one operation.

The save operation requires a fully qualified path to a file in which the guest domain's memory state will be saved. This path/file name must reside in the hypervisor's file system, not the libvirt client application's. There's no difference between the two if managing a local hypervisor, but it is critically important if connecting remotely to a hypervisor across the network. The example in Listing 4-30 demonstrates saving a guest called `demo-guest` to a file. It checks to verify that the guest is running before saving, though this is technically redundant since the hypervisor driver will do such a check itself.

Listing 4-30 shows how to save a guest domain and then stop it in one step.

Listing 4-30. Saving a Guest Domain

```
# Example-20.py
from __future__ import print_function
import sys
import libvirt

filename = '/var/lib/libvirt/save/demo-guest.img'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName('demo-guest')
if dom == None:
    print('Cannot find guest to be saved.', file=sys.stderr)
    exit(1)

info = dom.info()
if info == None:
    print('Cannot check guest state', file=sys.stderr)
    exit(1)

if info.state == VIR_DOMAIN_SHUTOFF:
    print('Not saving guest that is not running', file=sys.stderr)
    exit(1)

if dom.save(filename) < 0:
    print('Unable to save guest to '+filename, file=sys.stderr)

print('Guest state saved to '+filename, file=sys.stderr)

conn.close()
exit(0)
```

Some period of time later, the saved state file can then be used to restart the guest where it left off, using the `restore` method. The hypervisor driver will return an error if the guest is already running; however, it won't prevent attempts to restore from the same state file multiple times. As noted earlier, it is the application's responsibility to ensure the guest storage is in the same state as it was when the save image was created.

In addition, the `saveFlags` method allows the domain to be saved and at the same alter the configuration of the saved image. When the domain is restored, the new configuration will be applied to the running domain.

There is also another way to save a domain. The `managedSave` method can save a running domain state; however, in this case, the system selects the location for the saved image. In addition, the domain will be restored to the saved state when the domain is restarted.

Listing 4-31 shows how to restore a domain to a running state from a save image. Note that in this case the image file may not exist, and thus the program may fail if you try to run it.

Listing 4-31. Restoring a Guest Domain

```
# Example-21.py
from __future__ import print_function
import sys
import libvirt

filename = '/var/lib/libvirt/save/demo-guest.img'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
if id = conn.restore(filename) < 0:
    print('Unable to restore guest from '+filename, \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(id);
if dom == None:
    print('Cannot find guest that was restored', file=sys.stderr)
    exit(1)

print('Guest state restored from '+filename, file=sys.stderr)

conn.close()
exit(0)
```

Restoring a guest domain does not update the domain's current date/time.

When a guest domain is restored, it is returned to the same state when it was saved. This will include the date and time when the domain was saved. The guest domain usually will not be able to determine that a time period has passed since it was saved. This means the current time will not be automatically updated either during or after the restore operation.

This warning is still valid even if NTP is configured.

In addition to the `restore` method, the alternative method `restoreFlags` is available.

Migrating

Migration is the process of taking the image of a guest domain and moving it somewhere, typically from a hypervisor on one node to a hypervisor on another node. There are two methods for migration. The `migrate` method takes an established hypervisor connection and instructs the domain to migrate to this connection. The `migrateToUri` method takes a URI specifying a hypervisor connection, opens the connection, and then instructs the domain to migrate to this connection. Both these methods can be passed a parameter to specify a live migration. For migration to complete successfully, storage needs to be shared between the source and target hypervisors.

The first parameter of the `migrate` method specifies the connection to be used to the target of the migration. This parameter is required.

The second parameter of the `migrate` method specifies a set of flags that control how the migration takes place over the connection. If no flags are needed, then the parameter should be set to zero.

Flags may be one of more of the following:

`VIR_MIGRATE_LIVE`
`VIR_MIGRATE_PEER2PEER`
`VIR_MIGRATE_TUNNELLED`
`VIR_MIGRATE_PERSIST_DEST`
`VIR_MIGRATE_UNDEFINE_SOURCE`
`VIR_MIGRATE_PAUSED`
`VIR_MIGRATE_NON_SHARED_DISK`

`VIR_MIGRATE_NON_SHARED_INC`

`VIR_MIGRATE_CHANGE_PROTECTION`

`VIR_MIGRATE_UNSAFE`

`VIR_MIGRATE_OFFLINE`

The third parameter of the `migrate` method specifies a new name for the domain on the target of the migration. Not all hypervisors support this operation. If no rename of the domain is required, then the parameter should be set to `None`.

The fourth parameter of the `migrate` method specifies the URI to be used as the target of the migration. A URI is required only when the target system supports multiple hypervisors. If there is only a single hypervisor on the target system, then the parameter can be set to `None`.

The fifth and last parameter of the `migrate` method specifies the bandwidth in MiB/s to be used. If this maximum is not needed, then set the parameter to zero.

Migration is tricky if the domain is running and in use. The user may experience delays, or even a dropped connection in the case of a remote migration. You should ensure that the domain is at least not being used if it is running to make sure the users are not impacted.

To migrate a guest domain to a connection that is already open, use the `migrate` method. Listing 4-32 shows how to migrate a domain.

Listing 4-32. Migrate a Domain to an Open Connection

```
# Example-22.py
from __future__ import print_function
import sys
import libvirt
```

```
domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dest_conn = libvirt.open('qemu+ssh://desthost/system')
if dest_conn == None:
    print('Failed to open connection to qemu+ssh://desthost/system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

new_dom = dom.migrate(dest_conn, 0, None, None, 0)
if new_dom == None:
    print('Could not migrate to the new domain', file=sys.stderr)
    exit(1)

print('Domain was migrated successfully.', file=sys.stderr)

destconn.close()
conn.close()
exit(0)
```

The `migrateToURI` method is similar except that the destination URI is the first parameter instead of an existing connection. To migrate a guest domain to a URI, use the `migrateToURI` method. Listing 4-33 shows how to migrate a domain to a URI.

Listing 4-33. Migrate a Domain to a URI

```
# Example-23.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

new_dom = dom.migrateToURI('qemu+ssh://desthost/system', 0, None, 0)
if new_dom == None:
    print('Could not migrate to the new domain', file=sys.stderr)
    exit(1)

print('Domain was migrated successfully.', file=sys.stderr)

conn.close()
exit(0)
```

To migrate a live guest domain to a URI, use `migrate` or `migrateToURI` with the `VIR_MIGRATE_LIVE` flag set. Listing 4-34 shows how to migrate a live (running) domain to a URI.

Listing 4-34. Migrate a Live Domain to a URI

```
# Example-24.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dest_conn = libvirt.open('qemu+ssh://desthost/system')
if conn == None:
    print('Failed to open connection to qemu+ssh://desthost/system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(6)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

new_dom = dom.migrate(dest_conn, libvirt.VIR_MIGRATE_LIVE, None, \
                      None, 0)
if new_dom == None:
    print('Could not migrate to the new domain', file=sys.stderr)
    exit(1)

print('Domain was migrated successfully.', file=sys.stderr)

destconn.close()
conn.close()
exit(0)
```

In addition to the `migrate` method, there are the alternative methods called `migrate2`, `migrate3`, `migrateToURI`, `migrateToURI2`, and `migrateToURI3` to migrate over other types of connections.

Autostarting

A guest domain can be configured to autostart on a particular hypervisor, either by the hypervisor itself or by libvirt. In combination with a managed save, this allows the operating system on a guest domain to withstand host reboots without ever considering itself to have rebooted. When libvirt restarts, the guest domain will be automatically restored. This is handled by an API separate from regular save and restore operations because the paths must be known to libvirt without user input. Listing 4-35 shows how to set a domain to autostart when the main host is booted.

Listing 4-35. Set Autostart for a Domain

```
# Example-25.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(6)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)
```

```
dom.setAutostart(1)  # turn on autostart

conn.close()
exit(0)
```

Domain Configuration

Domains are defined in libvirt using XML. Everything related only to the domain, such as memory and CPU, is defined in the domain XML. The domain XML format is specified at `http://libvirt.org/formatdomain.html`. This can be accessed locally in `/usr/share/doc/libvirt-devel-version/` if your system has the libvirt-devel package installed.

Boot Modes

Booting via the BIOS is available for hypervisors supporting full virtualization. In this case, the BIOS has a boot order priority (floppy, hard disk, CD-ROM, network) that determines where to obtain/find the boot image (Listing 4-36).

Listing 4-36. Setting the Boot Mode

```
 ...
  <os>
    <type>hvm</type>
    <loader readonly='yes' type='rom'>
        /usr/lib/xen/boot/hvmloader
    </loader>
    <nvram template='/usr/share/OVMF/OVMF_VARS.fd'>
        /var/lib/libvirt/nvram/guest_VARS.fd
    </nvram>
    <boot dev='hd'/>
    <boot dev='cdrom'/>
    <bootmenu enable='yes' timeout='3000'/>
    <smbios mode='sysinfo'/>
    <bios useserial='yes' rebootTimeout='0'/>
  </os>
  ...
```

Memory/CPU Resources

CPU and memory resources can be set at the time the domain is created or dynamically while the domain is either active or inactive.

CPU resources are set at domain creation time using tags in the XML definition of the domain. The hypervisor defines a limit on the number of virtual CPUs that may not be exceeded either at domain creation time or at a later time. This maximum can be dependent on a number of resource and hypervisor limits. An example of the CPU XML specification follows:

```
<domain>
  ...
  <vcpu placement='static' cpuset="1-4,^3,6" current="1">2</vcpu>
  ...
</domain>
```

Memory resources are also set at domain creation using tags in the XML definition of the domain. Both the maximum and current allocations of memory to the domain should be set. An example of the memory XML specification follows:

```
<domain>
  ...
  <maxMemory slots='16' unit='KiB'>1524288</maxMemory>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  ...
</domain>
```

After the domain has been created, the number of virtual CPUs can be increased via the `setVcpus` or `setVcpusFlags` method. The number of CPUs may not exceed the hypervisor maximum discussed earlier. Listing 4-37 sets the maximum number of virtual CPUs for the domain. This can be done for both inactive and active domains. If the domain is active, then the new number will not be effective until the domain reboots.

Listing 4-37. Set the Number of Maximum Virtual CPUs for a Domain

```
# Example-29.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(6)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

dom.setVcpus(2)

conn.close()
exit(0)
```

Also, after the domain has been created, the amount of memory can be changed via the `setMemory` or `setMemoryFlags` method. The amount of memory should be expressed in kilobytes. Listing 4-38 sets the maximum amount of memory for a domain.

Listing 4-38. Set the Amount of Memory for a Domain

```
# Example-30.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByName(domName)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

dom.setMemory(4096) # 4 GigaBytes

conn.close()
exit(0)
```

In addition to the `setMemory` method, the alternative method `setMemoryFlags` is available.

Monitoring Performance

Statistical metrics are available for monitoring the utilization rates of domains, vCPUs, memory, block devices, and network interfaces.

Domain Block Device Performance

Disk usage statistics are provided by the blockStats method. Listing 4-39 shows how to fetch the statistics for a running domain. Note that the statistics can be returned for an inactive domain.

Listing 4-39. Get the Disk Block I/O Statistics

```
# Example-31.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(6)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

rd_req, rd_bytes, wr_req, wr_bytes, err = \
dom.blockStats('/path/to/linux-0.2.img')
print('Read requests issued:  '+str(rd_req))
print('Bytes read:            '+str(rd_bytes))
print('Write requests issued: '+str(wr_req))
print('Bytes written:         '+str(wr_bytes))
print('Number of errors:      '+str(err))

conn.close()
exit(0)
```

The returned tuple contains the number of read (write) requests issued and the actual number of bytes transferred. A block device is specified by the image file path or the device bus name set by the `devices/disk/target[@dev]` element in the domain XML.

In addition to the `blockStats` method, the alternative method `blockStatsFlags` is available.

vCPU Performance

To obtain the individual VCPU statistics, use the `getCPUStats` method. Listing 4-40 shows how to display the CPU statistics.

Listing 4-40. Get the Individual CPU Statistics

```
# Example-33.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(5)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

cpu_stats = dom.getCPUStats(False)
for (i, cpu) in enumerate(cpu_stats):
    print('CPU '+str(i)+' Time: '+ \
          str(cpu['cpu_time'] / 1000000000.))

conn.close()
exit(0)
```

getCPUStats takes one parameter, a Boolean. When False is used, the statistics are reported as an aggregate of all the CPUs. When True is used, then each CPU reports its individual statistics. Either way, a list is returned. The statistics are reported in nanoseconds. If a host has four CPUs, there will be four entries in the cpu_stats list.

getCPUStats(True) aggregates the statistics for all CPUs on the host. Listing 4-41 displays the aggregate statistics for a domain.

Listing 4-41. Get the Aggregate CPU Statistics

```
# Example-34.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(5)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)
```

```
stats = dom.getCPUStats(True)
print('cpu_time:    '+str(stats[0]['cpu_time']))
print('system_time: '+str(stats[0]['system_time']))
print('user_time:   '+str(stats[0]['user_time']))

conn.close()
exit(0)
```

Memory Statistics

To obtain the amount of memory currently used by the domain, you can use the `memoryStats` method. Listing 4-42 shows how to display the memory usage of a domain. Note that different kinds of memory are displayed including swap usage.

Listing 4-42. Get the Memory Statistics

```
# Example-35.py
from __future__ import print_function
import sys
import libvirt

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(5)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)
```

```
stats  = dom.memoryStats()
print('memory used:')
for name in stats:
    print('  '+str(stats[name])+' ('+name+')')

conn.close()
exit(0)
```

Note that `memoryStats` returns a dictionary object. This object will contain a variable number of entries depending on the hypervisor and guest domain capabilities.

I/O Statistics

To get the network statistics, you'll need the name of the host interface that the domain is connected to (usually vnetX). To find it, retrieve the domain XML description (libvirt modifies it at runtime). Then, look for the `devices/interface/target[@dev]` elements. Listing 4-43 shows how to display the I/O statistics for the domain.

Listing 4-43. Get the Network I/O Statistics

```
# Example-32.py
from __future__ import print_function
import sys
import libvirt
from xml.etree import ElementTree

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
dom = conn.lookupByID(5)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

tree = ElementTree.fromstring(dom.XMLDesc())
iface = tree.find('devices/interface/target').get('dev')
stats = dom.interfaceStats(iface)
print('read bytes:    '+str(stats[0]))
print('read packets:  '+str(stats[1]))
print('read errors:   '+str(stats[2]))
print('read drops:    '+str(stats[3]))
print('write bytes:   '+str(stats[4]))
print('write packets: '+str(stats[5]))
print('write errors:  '+str(stats[6]))
print('write drops:   '+str(stats[7]))

conn.close()
exit(0)
```

The `interfaceStats` method returns the number of bytes (packets) received (transmitted) and the number of reception/transmission errors.

Device Configuration

Configuration information for a guest domain can be obtained by using the `XMLDesc` method. This method returns the current description of a domain as an XML data stream. This stream can then be parsed to obtain detailed information about the domain and all the parts that make up the domain.

The `flags` parameter may contain any number of the following constants:

`VIR_DOMAIN_XML_SECURE`
`VIR_DOMAIN_XML_INACTIVE`
`VIR_DOMAIN_XML_UPDATE_CPU`
`VIR_DOMAIN_XML_MIGRATABLE`

Listing 4-44 shows how to display the domain's XML information.

Listing 4-44. Get Basic Domain Information from the Domain's XML Description

```
# Example-36.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(5)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)
```

```
raw_xml = dom.XMLDesc(0)
xml = minidom.parseString(raw_xml)
domainTypes = xml.getElementsByTagName('type')
for domainType in domainTypes:
    print(domainType.getAttribute('machine'))
    print(domainType.getAttribute('arch'))

conn.close()
exit(0)
```

Emulator

To discover the guest domain's emulator, find and display the content of the emulator XML tag. Listing 4-45 shows how to display the emulator the domain is using. There are a number of possibilities for this, but they are too numerous to list here.

Listing 4-45. Get the Domain's Emulator Information

```
# Example-37.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(5)
```

```
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

raw_xml = dom.XMLDesc(0)
xml = minidom.parseString(raw_xml)
domainEmulator = xml.getElementsByTagName('emulator')
print('emulator: '+domainEmulator[0].firstChild.data)

conn.close()
exit(0)
```

Listing 4-46 shows the XML configuration for the emulator.

Listing 4-46. Domain Emulator XML Information

```
<domain type='kvm'>
    ...
    <emulator>/usr/libexec/qemu-kvm</emulator>
    ...
</domain>
```

Disks

To discover the guest domain's disk (or disks), find and display the content of the disk XML tag (or tags). Listing 4-47 displays all the configured disks available to the domain.

Listing 4-47. Get the Domain's Disk Information

```
# Example-39.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom
```

```
domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(1)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

raw_xml = dom.XMLDesc(0)
xml = minidom.parseString(raw_xml)
diskTypes = xml.getElementsByTagName('disk')
for diskType in diskTypes:
    print('disk: type='+diskType.getAttribute('type')+' device='+ \
          diskType.getAttribute('device'))
    diskNodes = diskType.childNodes
    for diskNode in diskNodes:
        if diskNode.nodeName[0:1] != '#':
            print('  '+diskNode.nodeName)
            for attr in diskNode.attributes.keys():
                print('    '+diskNode.attributes[attr].name+' = '+
                 diskNode.attributes[attr].value)

conn.close()
exit(0)
```

Listing 4-48 shows the XML configuration for disks.

Listing 4-48. Domain Disk XML Information

```
<domain type='kvm'>
    ...
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2' cache='none'/>
      <source file='/var/lib/libvirt/images/RHEL7.1-x86_64-1.img'/>
      <target dev='vda' bus='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x06'
               function='0x0'/>
    </disk>
    <disk type='file' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <target dev='hdc' bus='ide'/>
      <readonly/>
      <address type='drive' controller='0' bus='1' target='0'
               unit='0'/>
    </disk>
    ...
</domain>
```

Networking

To discover the guest domain's network interfaces, find and display the interface XML tag. Listing 4-49 lists all the network interfaces for a domain.

Listing 4-49. Get the Domain's Network Interface Information

```
# Listing-38.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'Fedora22-x86_64-1'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(1)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

raw_xml = dom.XMLDesc(0)
xml = minidom.parseString(raw_xml)
interfaceTypes = xml.getElementsByTagName('interface')
for interfaceType in interfaceTypes:
    print('interface: type='+interfaceType.getAttribute('type'))
    interfaceNodes = interfaceType.childNodes
    for interfaceNode in interfaceNodes:
        if interfaceNode.nodeName[0:1] != '#':
            print('  '+interfaceNode.nodeName)
            for attr in interfaceNode.attributes.keys():
                print('    '+interfaceNode.attributes[attr].
                      name+' = '+
                 interfaceNode.attributes[attr].value)

conn.close()
exit(0)
```

Listing 4-50 shows the XML configuration for network interfaces.

Listing 4-50. Domain Network Interface XML Information

```
<domain type='kvm'>
    ...
    <interface type='network'>
      <mac address='52:54:00:94:f0:a4'/>
      <source network='default'/>
      <model type='virtio'/>
      <address type='pci' domain='0x0000' bus='0x00'
               slot='0x03' function='0x0'/>
    </interface>
    ...
</domain>
```

Mice, Keyboard, and Tablets

To discover the guest domain's input devices, find and display the input XML tags. Listing 4-51 shows how to list the XML information concerning mice, keyboards, and tablets.

Listing 4-51. Get the Domain's Input Device Information

```
# Example-40.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

domName = 'Fedora22-x86_64-1'
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = conn.lookupByID(1)
if dom == None:
    print('Failed to find the domain '+domName, file=sys.stderr)
    exit(1)

raw_xml = dom.XMLDesc(0)
xml = minidom.parseString(raw_xml)
devicesTypes = xml.getElementsByTagName('input')
for inputType in devicesTypes:
    print('input: type='+inputType.getAttribute('type')+' bus='+ \
          inputType.getAttribute('bus'))
    inputNodes = inputType.childNodes
    for inputNode in inputNodes:
        if inputNode.nodeName[0:1] != '#':
            print('  '+inputNode.nodeName)
            for attr in inputNode.attributes.keys():
                print('    '+inputNode.attributes[attr].name+' = '+
                 inputNode.attributes[attr].value)

conn.close()
exit(0)
```

Listing 4-52 shows the XML configuration for mouse, keyboard, and tablet.

Listing 4-52. Domain Mouse, Keyboard, and Tablet XML Information

```
<domain type='kvm'>
    ...
    <input type='tablet' bus='usb'/>
    <input type='mouse' bus='ps2'/>
    ...
</domain>
```

USB Device Passthrough

The USB device passthrough capability allows a physical USB device from the host machine to be assigned directly to a guest machine. The guest OS drivers can use the device hardware directly without relying on any driver capabilities from the host OS.

Important USB devices are inherited by the guest domain only at boot time. Newly activated USB devices cannot be inherited from the host after the guest domain has booted.

Some caveats apply when using USB device passthrough. When a USB device is directly assigned to a guest, migration will not be possible without first hot-unplugging the device from the guest. In addition, libvirt does not guarantee that direct device assignment is secure, leaving security policy decisions to the underlying virtualization technology.

PCI Device Passthrough

The PCI device passthrough capability allows a physical PCI device from the host machine to be assigned directly to a guest machine. The guest OS drivers can use the device hardware directly without relying on any driver capabilities from the host OS.

Some caveats apply when using PCI device passthrough. When a PCI device is directly assigned to a guest, migration will not be possible without first hot-unplugging the device from the guest. In addition, libvirt does not guarantee that direct device assignment is secure, leaving security policy decisions to the underlying virtualization technology. Secure PCI device passthrough typically requires special hardware capabilities, such as the VT-d feature for Intel chipset, or the IOMMU for AMD chipsets.

There are two modes in which a PCI device can be attached, managed or unmanaged mode, although at time of writing only KVM supports managed mode attachment. In managed mode, the configured device will be automatically detached from the host OS drivers when the guest is started and then re-attached when the guest shuts down. In unmanaged mode, the device must be explicitly detached ahead of booting the guest. The guest will refuse to start if the device is still attached to the host OS. The libvirt Node Device APIs provide a means to detach/reattach PCI devices from/to host drivers. Alternatively, the host OS may be configured to blacklist the PCI devices used for guests so that they never get attached to host OS drivers.

In both modes, the virtualization technology will always perform a reset on the device before starting a guest and after the guest shuts down. This is critical to ensure isolation between the host and guest OSs. There are a variety of ways in which a PCI device can be reset. Some reset techniques are limited in scope to a single device/function, while others may affect multiple devices at once. In the latter case, it will be necessary to co-assign all affected devices to the same guest; otherwise, a reset will be impossible to do safely. The Node Device APIs can be used to determine whether a device needs to be co-assigned by manually detaching the device and then attempting to perform the reset operation. If this succeeds, then it will be possible to assign the device to a guest on its own. If it fails, then it will be necessary to co-assign the device with others on the same PCI bus.

A PCI device is attached to a guest using the hostdevice element. The mode attribute should always be set to subsystem, and the type attribute should be set to pci. The managed attribute can be either yes or no as required by the application. Within the hostdevice element there is a source element, and within that a further address element is used to specify the PCI device to be attached. The address element expects attributes for domain, bus, slot, and function. This is easiest to see with the short example in Listing 4-53.

Listing 4-53. Get the Domain's Input Device Information

```
<hostdev mode='subsystem' type='pci' managed='yes'>
  <source>
    <address domain='0x0000'
             bus='0x06'
             slot='0x12'
             function='0x5'/>
  </source>
</hostdev>
```

Block Device Jobs

libvirt provides generic block job methods that can be used to initiate and manage operations on disks that belong to a domain. Jobs are started by calling the function associated with the desired operation (e.g., blockPull). Once started, all block jobs are managed in the same manner. They can be aborted, throttled, and queried. Upon completion, an asynchronous event is issued to indicate the final status.

The following block jobs can be started:

- blockPull starts a block pull operation for the specified disk. This operation is valid only for specially configured disks. blockPull will populate a disk image with data from its backing image. Once all data from its backing image has been pulled, the disk no longer depends on a backing image.

- A disk can be queried for active block jobs by using `blockJobInfo`. If found, job information is reported in a structure that contains the job type, bandwidth throttling setting, and progress information.
- Use `virDomainBlockJobAbort()` to cancel the active block job on the specified disk.
- Use `blockJobSetSpeed()` to limit the amount of bandwidth that a block job may consume. Bandwidth is specified in units of MB/sec. Listing 4-54 displays the domain's input device information.

Listing 4-54. Get the Domain's Input Device Information

```
# Example-40.py
from __future__ import print_function
import sys
import libvirt

domxml =
 """<domain type='kvm'>
      <name>example</name>
      <memory>131072</memory>
      <vcpu>1</vcpu>
      <os>
        <type arch='x86_64' machine='pc-0.13'>hvm</type>
      </os>
      <devices>
        <disk type='file' device='disk'>
          <driver name='qemu' type='qed'/>
          <source file='/var/lib/libvirt/images/example.qed' />
          <target dev='vda' bus='virtio'/>
        </disk>
      </devices>
    </domain>"""

def do_cmd (cmdline):
    status = os.system(cmdline)
    if status < 0:
        return -1
    return WEXITSTATUS(status)

def make_domain (conn):
    do_cmd("qemu-img create -f raw " + \
           "/var/lib/libvirt/images/backing.qed 100M")
    do_cmd("qemu-img create -f qed -b " + \
           "/var/lib/libvirt/images/backing.qed"+ \
           "/var/lib/libvirt/images/example.qed")
    dom = conn.createXML(domxml, 0)
    return dom

dom = None
disk = "/var/lib/libvirt/images/example.qed"

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

dom = make_domain(conn)
if dom == None:
    print("Failed to create domain", file=sys.stderr)
    exit(1)
```

```
if dom.blockPull(disk, 0, 0) < 0:
    print("Failed to start block pull", file=sys.stderr)
    exit(1)

while (1):
    info = dom.blockJobInfo(disk, 0);
    if (info != None:
        print("BlockPull progress: %0.0f %%",
            float(100 * info.cur / info.end))
    elif info.cur == info.end):
        printf("BlockPull complete")
        break
    else:
        print("Failed to query block jobs", file=os.stderr)
        break
    time.sleep(1)

os.unlink("/var/lib/libvirt/images/backing.qed")
os.unlink("/var/lib/libvirt/images/example.qed")
if dom != NULL:
   conn.destroy(dom)

conn.close()
exit(0)
```

Summary

In this chapter, you learned more about the concept of domains. This includes creating domains, configuring domains, setting information, returning information, and migrating domains. This chapter may require multiple reads to glean the information you require to work with domains. Therefore, this chapter has presented the information in such a way that it will be easy to refer to when you need more information or a reminder of how to perform a specific activity.

CHAPTER 5

Storage Pools and Volumes

libvirt provides storage management on the physical host through storage pools and volumes. Storage pools and volumes can be located on the main host or remotely via NFS mounts on the host. Storage pools are areas of storage set aside to contain volumes. Volumes are directly used by client domains for file system storage and are formatted by the client domain. A volume can contain any type of domain client partition type(s) and is controlled strictly by the domain client. A volume is always a member of a single storage pool. However, a domain can have access to multiple client volumes as long as none of those volumes is shared with any other active client domains; this is because there are no facilities in libvirt to share volumes between domains.

Pools Overview

This section will introduce the different types of storage pools and the advantages and disadvantages of using them. A storage pool is a quantity of storage set aside by an administrator, often a dedicated storage administrator, for use by virtual machines. Storage pools are divided into storage volumes either by the storage administrator or by the system administrator, and the volumes are assigned to VMs as block devices.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_5

NFS Storage Pools

The storage administrator responsible for an NFS server creates a share to store virtual machines' data. The system administrator defines a pool on the virtualization host with the details of the share (e.g., `nfs.example.com:/path/to/share` should be mounted on `/vm_data`). When the pool is started, libvirt mounts the share on the specified directory, just as if the system administrator logged in and executed `mount nfs.example.com:/path/to/share/vmdata`. If the pool is configured to autostart, libvirt ensures that the NFS share is mounted on the directory specified when libvirt is started.

Once the pool is started, the files in the NFS share are reported as volumes, and the storage volumes' paths may be queried using the libvirt APIs. The volumes' paths can then be copied into the section of a VM's XML definition describing the source storage for the VM's block devices. In the case of NFS, an application using the libvirt methods can create and delete volumes in the pool (files in the NFS share) up to the limit of the size of the pool (the storage capacity of the share). Not all pool types support creating and deleting volumes. Stopping the pool (somewhat misleadingly referred to by virsh and the API as `pool-destroy`) undoes the start operation, in this case, unmounting the NFS share. The data on the share is not modified by the destroy operation, despite the name. See the system `man virsh` page for more details.

The advantage of NFS storage pools is that read/write access is removed from the client domain and host and placed on the remote server hosting the NFS share. This is also the weakness as the remote server can become overloaded if it hosts too many client domains.

iSCSI Storage Pools

A second example is an iSCSI pool. A storage administrator provisions an iSCSI target to present a set of LUNs to the host running the VMs. LUNs are a unique identifier that enables separate devices in a storage subsystem to be addressed by Fibre Channel, iSCSI, or SCSI network protocols. When libvirt is configured to manage that iSCSI target as a pool, libvirt will ensure that the host logs into the iSCSI target, and libvirt can then report the available LUNs as storage volumes. The volumes' paths can be queried and used in the VM's XML definitions as in the NFS example. In this case, the LUNs are defined on the iSCSI server, and libvirt cannot create and delete volumes.

The advantage of using iSCSI storage pools is that management and read/write activities are taken care of remotely and not on the client domain or main host. However, management of this type of storage can become a hassle because an extra layer of ownership (for the storage volume) has been introduced.

Other Storage Pools

The most common storage pools, especially when testing, are local storage pools. These can be placed anywhere on the local storage of the main host. The advantage of this type of storage pool is that it is easily managed. The disadvantage is that if too many domains are active, it can slow the server down if each domain is heavily using its volume(s).

Storage pools and volumes are not required for the proper operation of VMs. Pools and volumes provide a way for libvirt to ensure that a particular piece of storage will be available for a VM, but some administrators prefer to manage their own storage, and VMs operate properly without any pools or volumes defined. On systems that do not use pools, system administrators must ensure the availability of the VMs' storage using whatever tools they prefer by, for example, adding the NFS share to the host's fstab so that the share is mounted at boot time.

If at this point the value of pools and volumes over traditional system administration tools is unclear, note that one of the features of libvirt is its remote protocol, so it's possible to manage all aspects of a virtual machine's lifecycle as well as the configuration of the resources required by the VM. These operations can be performed on a remote host entirely within the Python libvirt module. In other words, a management application using libvirt can enable a user to perform all the required tasks for configuring the host for a VM: allocating resources, running the VM, shutting it down, and deallocating the resources, without requiring shell access or any other control channel.

libvirt supports the following storage pool types:

- **Directory back end**: This is a local storage directory hosted on the main host.
- **Local file system back end**: This is a complete file system hosted by the local host.
- **Network file system back end**: This is usually an NFS share mounted on the local host.
- **Logical back end**: This is can be any logical file system.
- **Disk back end**: This is one or more disks dedicated to hosting the domain client and mounted on the local host.
- **iSCSI back end**: This is an iSCSI mount dedicated to hosting domains and mounted on the local host.
- **SCSI back end**: This is a SCSI mount dedicated to hosting domains and mounted on the local host.
- **Multipath back end**: This is usually used with iSCSI to provide a multipath file system mounted on the local host.
- **RADOS Block Device (RBD) back end**: This is a CEPH file system mounted on the local domain.
- **Sheepdog back end**: This is a sheepdog system mounted on the local host.
- **Gluster back end**: This is a Gluster system mounted on the local host.
- **ZFS back end**: This is a ZFS partition mounted on the local host.

Listing Pools

A list of storage pool objects can be obtained using the `listAllStoragePools` method of the `virConnect` class, as shown in Listing 5-1.

The `flags` parameter can be one or more of the following constants:

```
VIR_CONNECT_LIST_STORAGE_POOLS_INACTIVE
VIR_CONNECT_LIST_STORAGE_POOLS_ACTIVE
VIR_CONNECT_LIST_STORAGE_POOLS_PERSISTENT
VIR_CONNECT_LIST_STORAGE_POOLS_TRANSIENT
VIR_CONNECT_LIST_STORAGE_POOLS_AUTOSTART
VIR_CONNECT_LIST_STORAGE_POOLS_NO_AUTOSTART
VIR_CONNECT_LIST_STORAGE_POOLS_DIR
VIR_CONNECT_LIST_STORAGE_POOLS_FS
VIR_CONNECT_LIST_STORAGE_POOLS_NETFS
VIR_CONNECT_LIST_STORAGE_POOLS_LOGICAL
VIR_CONNECT_LIST_STORAGE_POOLS_DISK
VIR_CONNECT_LIST_STORAGE_POOLS_ISCSI
VIR_CONNECT_LIST_STORAGE_POOLS_SCSI
VIR_CONNECT_LIST_STORAGE_POOLS_MPATH
VIR_CONNECT_LIST_STORAGE_POOLS_RBD
VIR_CONNECT_LIST_STORAGE_POOLS_SHEEPDOG
VIR_CONNECT_LIST_STORAGE_POOLS_GLUSTER
VIR_CONNECT_LIST_STORAGE_POOLS_ZFS
```

Listing 5-1. Get the List of Storage Pools

```
# Example-1.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
```

```
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

pools = conn.listAllStoragePools(0)
if pools == None:
    print('Failed to locate any StoragePool objects.', \
          file=sys.stderr)
    exit(1)

for pool in pools:
    print('Pool: '+pool.name())

conn.close()
exit(0)
```

Pool Usage

There are a number of methods available in the `virStoragePool` class. The example in Listing 5-2 features a number of the methods that describe some attributes of a pool.

Listing 5-2. Usage of Some Storage Pool Methods

```
# Example-2.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
pool = conn.storagePoolLookupByName('default')
if pool == None:
    print('Failed to locate any StoragePool objects.', \
          file=sys.stderr)
    exit(1)

info = pool.info()

print('Pool: '+pool.name())
print('  UUID: '+pool.UUIDString())
print('  Autostart: '+str(pool.autostart()))
print('  Is active: '+str(pool.isActive()))
print('  Is persistent: '+str(pool.isPersistent()))
print('  Num volumes: '+str(pool.numOfVolumes()))
print('  Pool state: '+str(info[0]))
print('  Capacity: '+str(info[1]))
print('  Allocation: '+str(info[2]))
print('  Available: '+str(info[3]))

conn.close()
exit(0)
```

Many of the methods shown in the previous example provide information concerning storage pools that are on remote file systems, disk systems, or types other than local file systems. For instance, if the `autostart` flag is set, then when the user connects to the storage pool, libvirt will automatically make the storage pool available if it is not on a local file system, e.g., an NFS mount. Storage pools on local file systems also need to be started if `autostart` is not set.

The `isActive` method indicates whether the user must activate the storage pool in some way. The `create` method can activate a storage pool.

The isPersistent method indicates whether a storage pool needs to be activated using the create method. A value of 1 indicates that the storage pool is persistent and will remain on the file system after it is released.

The flags parameter can be the following constant or any of the other available constants:

```
VIR_STORAGE_XML_INACTIVE
```

Listing 5-3 shows how to get the XML description of a storage pool.

Listing 5-3. Get the XML Description of a Storage Pool

```
# Example-3.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

pool = conn.storagePoolLookupByName('default')
if pool == None:
    print('Failed to locate any StoragePool objects.', \
          file=sys.stderr)
    exit(1)
xml = pool.XMLDesc(0)
print(xml)

conn.close()
exit(0)
```

Lifecycle Control

Listing 5-4 shows how to create and destroy both a persistent storage pool and a nonpersistent storage pool. Note that a storage pool cannot be destroyed if it is in an active state. By default, storage pools are created in an inactive state. Note: In Listing 5-4, be sure to change `<path>` in the `xmlDesc` to a valid location on your file system with enough space.

Listing 5-4. Create and Destroy Storage Pools

```
# Example-4.py
from __future__ import print_function
import sys
import libvirt
xmlDesc = """
<pool type='dir'>
  <name>mypool</name>
  <uuid>8c79f996-cb2a-d24d-9822-ac7547ab2d01</uuid>
  <capacity unit='bytes'>4306780815</capacity>
  <allocation unit='bytes'>237457858</allocation>
  <available unit='bytes'>4069322956</available>
  <source>
  </source>
  <target>
    <path>/home/dashley/images</path>
    <permissions>
      <mode>0755</mode>
      <owner>-1</owner>
      <group>-1</group>
    </permissions>
  </target>
</pool>"""
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

# create a new persistent storage pool
pool = conn.storagePoolDefineXML(xmlDesc, 0)
if pool == None:
    print('Failed to create StoragePool object.', \
          file=sys.stderr)
    exit(1)

# destroy the storage pool
pool.undefine()

# create a new non-persistent storage pool
pool = conn.storagePoolCreateXML(xmlDesc, 0)
if pool == None:
    print('Failed to create StoragePool object.', \
          file=sys.stderr)
    exit(1)

# destroy the storage pool
pool.undefine()

conn.close()
exit(0)
```

Note that the storage volumes defined in a storage pool will remain on the file system unless the `delete` method is called. But be careful about leaving storage volumes in place. If they exist on a remote file system or disk, then that file system may become unavailable to the guest domain since there will be no mechanism to reactivate the remote file system or disk by the libvirt storage system at a future time.

Discovering Pool Sources

The sources for a storage can be discovered by examining the pool's XML description. An example program follows that prints out a pool's source description attributes. Currently the `flags` parameter for the `createXML` method should always be 0. Note that Listing 5-5 will not work if a `poolName` value of `default` already exists.

Listing 5-5. Discover a Storage Pool's Sources

```
# Example-5.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

poolName = 'default'


conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

sp = conn.storagePoolLookupByName(poolName)
if sp == None:
    print('Failed to find storage pool '+poolName, \
          file=sys.stderr)
    exit(1)

raw_xml = sp.XMLDesc(0)
xml = minidom.parseString(raw_xml)
name = xml.getElementsByTagName('name')
print('pool name: '+poolName)
```

```
spTypes = xml.getElementsByTagName('source')
for spType in spTypes:
    attr = spType.getAttribute('name')
    if attr != None:
        print('  name = '+attr)
    attr = spType.getAttribute('path')
    if attr != None:
        print('  path = '+attr)
    attr = spType.getAttribute('dir')
    if attr != None:
        print('  dir = '+attr)
    attr = spType.getAttribute('type')
    if attr != None:
        print('  type = '+attr)
    attr = spType.getAttribute('username')
    if attr != None:
        print('  username = '+attr)

conn.close()
exit(0)
```

Pool Configuration

There are a number of methods that can configure aspects of a storage pool, but the main method is `setAutostart`, as shown in Listing 5-6. Note that Listing 5-6 will not work if the `poolName` value of `default` already exists.

Listing 5-6. setAutostart Method

```
# Example-6.py
from __future__ import print_function
import sys
import libvirt
```

```
poolName = 'default'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

sp = conn.storagePoolLookupByName(poolName)
if sp == None:
    print('Failed to find storage pool '+poolName, \
          file=sys.stderr)
    exit(1)

print('Current autostart seting: '+str(sp.autostart()))
if sp.autostart() == True:
    sp.setAutostart(0)
else:
    sp.setAutostart(1)
print('Current autostart seting: '+str(sp.autostart()))

conn.close()
exit(0)
```

Volume Overview

Storage volumes are the basic unit of storage that house a guest domain's storage requirements. All the necessary partitions used to house a guest domain are encapsulated by the storage volume. Storage volumes are in turn contained in storage pools. A storage pool can contain as many storage pools as the underlying disk partition will hold.

Listing Volumes

Listing 5-7 demonstrates how to list all the storage volumes contained by the default storage pool. Listing 5-7 will not work if a poolName value of default already exists.

Listing 5-7. List the Storage Volumes

```
# Example-7.py
from __future__ import print_function
import sys
import libvirt
from xml.dom import minidom

poolName = 'default'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

sp = conn.storagePoolLookupByName(poolName)
if sp == None:
    print('Failed to find storage pool '+poolName, \
          file=sys.stderr)
    exit(1)

stgvols = sp.listVolumes()
print('Storage pool: '+poolName)
for stgvol in stgvols :
    print('  Storage vol: '+stgvol)

conn.close()
exit(0)
```

Volume Information

Information about a storage volume is obtained by using the `info` method. Listing 5-8 shows how to list the information about each storage volume in the `default` storage pool. Listing 5-8 will not work if a `poolName` value of `default` already exists.

Listing 5-8. List Storage Volume Information

```
# Example-8.py
from __future__ import print_function
import sys
import libvirt

poolName = 'default'


conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

pool = conn.storagePoolLookupByName(poolName)
if pool == None:
    print('Failed to locate any StoragePool objects.', \
          file=sys.stderr)
    exit(1)

stgvols = pool.listVolumes()

print('Pool: '+pool.name())
for stgvolname in stgvols:
    print('  Volume: '+stgvolname)
    stgvol = pool.storageVolLookupByName(stgvolname)
    info = stgvol.info()
    print('    Type: '+str(info[0]))
    print('    Capacity: '+str(info[1]))
    print('    Allocation: '+str(info[2]))

conn.close()
exit(0)
```

Creating and Deleting Volumes

Storage volumes are created using the storage pool's `createXML` method. The type and attributes of the storage volume are specified in the XML passed to the `createXML` method. The `flags` parameter can be one or more of the following constants:

```
VIR_STORAGE_VOL_CREATE_PREALLOC_METADATA
VIR_STORAGE_VOL_CREATE_REFLINKVIR_CONNECT_LIST_STORAGE_POOLS_
INACTIVE
```

Listing 5-9 will not work if the XML entry `<path>` does not point to a valid location.

Listing 5-9. Create a Storage Volume

```
# Example-9.py
from __future__ import print_function
import sys
import libvirt

stgvol_xml = """
<volume>
  <name>sparse.img</name>
  <allocation>0</allocation>
  <capacity unit="G">2</capacity>
  <target>
    <path>/var/lib/virt/images/sparse.img</path>
    <permissions>
      <owner>107</owner>
      <group>107</group>
      <mode>0744</mode>
      <label>virt_image_t</label>
    </permissions>
  </target>
</volume>"""
pool = 'default'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

pool = conn.storagePoolLookupByName(pool)
if pool == None:
    print('Failed to locate any StoragePool objects.', \
          file=sys.stderr)
    exit(1)

stgvol = pool.createXML(stgvol_xml, 0)
if stgvol == None:
    print('Failed to create a  StorageVol objects.', file=sys.stderr)
    exit(1)

# remove the storage volume
# physically remove the storage volume from the underlying disk media
stgvol.wipe(0)
# logically remove the storage volume from the storage pool
stgvol.delete(0)
```

```
conn.close()
exit(0)
```

Cloning Volumes

Cloning a storage volume is similar to creating a new storage volume, except that an existing storage volume is used for most of the attributes. Only the name and permissions in the XML parameter are used for the new volume; everything else is inherited from the existing volume.

It should be noted that cloning can take a long time to accomplish, depending on the size of the storage volume being cloned. This is because the clone process copies the data from the source volume to the new target volume. Listing 5-10 will not work if the XML entry `<path>` does not point to a valid location.

Listing 5-10. Clone an Existing Storage Volume

```
# Example-10.py
from __future__ import print_function
import sys
import libvirt

stgvol_xml = """
<volume>
  <name>sparse.img</name>
  <allocation>0</allocation>
  <capacity unit="G">2</capacity>
  <target>
    <path>/var/lib/virt/images/sparse.img</path>
    <permissions>
      <owner>107</owner>
      <group>107</group>
      <mode>0744</mode>
      <label>virt_image_t</label>
    </permissions>
  </target>
</volume>"""
stgvol_xml2 = """
<volume>
  <name>sparse2.img</name>
  <allocation>0</allocation>
  <capacity unit="G">2</capacity>
  <target>
    <path>/var/lib/virt/images/sparse.img</path>
    <permissions>
      <owner>107</owner>
      <group>107</group>
      <mode>0744</mode>
      <label>virt_image_t</label>
    </permissions>
  </target>
</volume>"""
pool = 'default'

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

pool = conn.storagePoolLookupByName(pool)
if pool == None:
    print('Failed to locate any StoragePool objects.', \
          file=sys.stderr)
    exit(1)
```

```
# create a new storage volume
stgvol = pool.createXML(stgvol_xml, 0)
if stgvol == None:
    print('Failed to create a  StorageVol object.', \
          file=sys.stderr)
    exit(1)

# now clone the existing storage volume
print('This could take some time...')
stgvol2 = pool.createXMLFrom(stgvol_xml2, stgvol, 0)
if stgvol2 == None:
    print('Failed to clone a  StorageVol object.', \
          file=sys.stderr)
    exit(1)

# remove the cloned storage volume
# physically remove the storage volume from the
#    underlying disk media
stgvol2.wipe(0)
# logically remove the storage volume from the
#    storage pool
stgvol2.delete(0)

# remove the storage volume
# physically remove the storage volume from the
#    underlying disk media
stgvol.wipe(0)
# logically remove the storage volume from the
#    storage pool
stgvol.delete(0)

conn.close()
exit(0)
```

Configuring Volumes

Listing 5-11 is an XML description for a storage volume.

Listing 5-11. XML Description for a Storage Volume

```
<volume>
  <name>sparse.img</name>
  <allocation>0</allocation>
  <capacity unit="G">2</capacity>
  <target>
    <path>/var/lib/virt/images/sparse.img</path>
    <permissions>
      <owner>107</owner>
      <group>107</group>
      <mode>0744</mode>
      <label>virt_image_t</label>
    </permissions>
  </target>
</volume>
```

Summary

This chapter introduced the topics of storage pools and volumes. The chapter covered different types of storage pools and the advantages and disadvantages of them. It also described volumes and some of their performance characteristics.

CHAPTER 6

Virtual Networks

A virtual network provides a method for connecting the network devices of one or more guest domains within a single host. The virtual network can do either of the following:

- Remain isolated to the host.
- Allow routing of traffic off-node via the active network interfaces of the host OS. This includes the option to apply NAT to IPv4 traffic.

A virtual network is represented by an instance of the `virNetwork` class and has two unique identifiers.

- **Name**: This is a short string, unique among all the virtual networks on a single host, both running and inactive. For maximum portability between hypervisors, applications should use only alphanumeric (`a–Z`, `0–9`), hyphen (`-`), and underscore (`_`) characters in names.
- **UUID**: This consists of 16 unsigned bytes, guaranteed to be unique among all the virtual networks on any host. RFC 4122 defines the format for UUIDs and provides a recommended algorithm for generating UUIDs with guaranteed uniqueness.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_6

A virtual network can be transient or persistent. A transient virtual network can be managed only while it is running on the host. When taken offline, all traces of it will disappear. A persistent virtual network has its configuration maintained in a data store on the host, in an implementation-defined format. Thus, when a persistent network is brought offline, it is still possible to manage its inactive configuration. A transient network can be turned into a persistent network on the fly by defining a configuration for it.

After the installation of libvirt, every host will get a single virtual network instance called `default`, which provides DHCP services to guests and allows NAT'd IP connectivity to the host's interfaces. This service is of most use to hosts with intermittent network connectivity such as laptops using wireless networking.

Bridged networking is also supported. This allows a virtualized client to share the host's network adapter directly and thus exist on the host's real network. There are two ways to create this type of network. The old way is to set up a bridged network on the host. The second way is a routed network, which is beyond the scope of this book.

Recently another network model has been added to libvirt, known as *passthrough networking*. This method allows the virtualized client to make itself visible to the outside world. This method is also beyond the scope of this book.

Listing Networks

Virtual networks are discovered using the class `virConnect` methods `networkLookupByName`, `networkLookupByUUID`, `networkLookupByUUIDString`, and `listNetworks`. Listing 6-1 shows how to use these methods.

Listing 6-1. Discovering and Finding Virtual Networks

```
# Example-1.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

# discover all the virtual networks
networks = conn.listNetworks()
print('Virtual networks:')
for network in networks:
    print('  '+network)
print()

# lookup the default network by name
network = conn.networkLookupByName('default')
print('Virtual network default:')
print('  name: '+network.name())
uuid = network.UUIDString()
print('  UUID: '+uuid)
print('  bridge: '+network.bridgeName())
print()

# lookup the default network by name
network = conn.networkLookupByUUIDString(uuid)
print('Virtual network default:')
print('  name: '+network.name())
print('  UUID: '+network.UUIDString())
print('  bridge: '+network.bridgeName())

conn.close()
exit(0)
```

Lifecycle Control

Listing 6-2 shows how to use the `networkCreateXML`, `networkDefineXML`, and `destroy` methods.

Listing 6-2. Creating and Destroying Virtual Networks

```
# Example-2.py
from __future__ import print_function
import sys
import libvirt

xml = """
<network>
  <name>mynetwork</name>
  <bridge name="virbr1" />
  <forward mode="nat"/>
  <ip address="192.168.142.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.142.2" end="192.168.142.254" />
    </dhcp>
  </ip>
</network>"""

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

# create a persistent virtual network
network = conn.networkCreateXML(xml)
if network == None:
    print('Failed to create a virtual network', file=sys.
stderr)
    exit(1)
active = network.isActive()
if active == 1:
    print('The new persistent virtual network is active')
else:
    print('The new persistent virtual network is not active')

# now destroy the persistent virtual network
network.destroy()
print()

# create a transient virtual network
network = conn.networkDefineXML(xml)
if network == None:
    print('Failed to define a virtual network', file=sys.
    stderr)
    exit(1)
active = network.isActive()
if active == 1:
    print('The new transient virtual network is active')
else:
    print('The new transient virtual network is not active')
network.create() # set the network active
active = network.isActive()
```

```
if active == 1:
    print('The new transient virtual network is active')
else:
    print('The new transient virtual network is not active')

# now destroy the transient virtual network
network.destroy()

conn.close()
exit(0)
```

Network Configuration

Listing 6-3 shows how to use the `XMLDesc`, `autostart`, `isActive`, `isPersistent`, and `setAutostart` methods.

Listing 6-3. Configuring Virtual Networks

```
# Example-1.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

# lookup the default network by name
network = conn.networkLookupByName('default')
print('Virtual network default:')
print('  name: '+network.name())
print('  UUID: '+network.UUIDString())
print('  bridge: '+network.bridgeName())
print('  autostart: '+str(network.autostart()))
print('  is active: '+str(network.isActive()))
print('  is persistent: '+str(network.isPersistent()))
print()

print('Unsetting autostart')
network.setAutostart(0)
print('  autostart: '+str(network.autostart()))
print('Setting autostart')
network.setAutostart(1)
print('  autostart: '+str(network.autostart()))
print()

xml = network.XMLDesc(0)
print('XML description:')
print(xml)

conn.close()
exit(0)
```

Summary

In this chapter, you learned how to use the basic libvirt networking functions. You saw how to create and destroy networks as well as configure them. These are the basic libvirt functions for networking, but there are some others I have not covered. This chapter gave the basic set of functions for dealing with networks.

CHAPTER 7

Network Interfaces

You can configure network interfaces on physical hosts with the methods in the `virInterface` class. This is useful for setting up the host to share one physical interface between the multiple guest domains that you want connected directly to the network (briefly, you can enslave a physical interface to the bridge and then create a tap device for each VM that is sharing the interface), as well as for general host network interface management. In addition to configuring the physical hardware, you can use the methods to configure bridges, bonded interfaces, and VLAN interfaces.

The `virInterface` class is *not* used to configure virtual networks (which are used to conceal the guest domain's interface behind a NAT); virtual networks are instead configured using the `virNetwork` class described in Chapter 6.

Each host interface is represented by an instance of the `virInterface` class, and each of these instances has a single unique identifier: `name`.

The `name` method returns a string unique among all interfaces (active or inactive) on a host. This is the same string used by the operating system to identify the interface (e.g., `eth0` or `br1`).

Each interface object also has a second, nonunique index that can be duplicated in other interfaces on the same host.

The `MACString` method returns an ASCII string representation of the MAC address of this interface. Since multiple interfaces can share the

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_7

same MAC address (for example, in the case of VLANs), this is *not* a unique identifier. However, it can still be used to search for an interface.

All interfaces configured with libvirt should be considered as persistent since libvirt is actually changing the host's own persistent configuration data (usually contained in files somewhere under `/etc`) and not the interface itself.

When a new interface is defined (using the `interfaceDefineXML` method) or the configuration of an existing interface is changed (again, with the `interfaceDefineXML` method), this configuration will be stored on the host. The live configuration of the interface itself will not be changed until the interface is restarted manually or the host is rebooted.

This chapter covers the management of physical network interfaces using the libvirt `virInterface` class.

XML Interface Description Format

The current Relax-NG definition of the XML that is produced and accepted by `interfaceDefineXML` and `XMLDesc` can be found in the file `data/xml/interface.rng` of the netcf package, available at `http://git.fedorahosted.org/git/netcf.git?p=netcf.git;a=tree`. Listings 7-1 through 7-4 provide some examples of common interface configurations.

Listing 7-1. XML Definition of an Ethernet Interface Using DHCP

```
<interface type='ethernet' name='eth0'>
  <start mode='onboot'/>
  <mac address='aa:bb:cc:dd:ee:ff'/>
  <protocol family='ipv4'>
    <dhcp/>
  </protocol>
</interface>
```

Listing 7-2. XML Definition of an Ethernet Interface with Static IP

```
<interface type='ethernet' name='eth0'>
  <start mode='onboot'/>
  <mac address='aa:bb:cc:dd:ee:ff'/>
  <protocol family='ipv4'>
    <ip address="192.168.0.5" prefix="24"/>
    <route gateway="192.168.0.1"/>
  </protocol>
</interface>
```

Listing 7-3. XML Definition of a Bridge Device with eth0 and eth1 Attached

```
<interface type="bridge" name="br0">
  <start mode="onboot"/>
  <mtu size="1500"/>
  <protocol family="ipv4">
    <dhcp/>
  </protocol>
  <bridge stp="off" delay="0.01">
    <interface type="ethernet" name="eth0">
      <mac address="ab:bb:cc:dd:ee:ff"/>
    </interface>
    <interface type="ethernet" name="eth1"/>
  </bridge>
</interface>
```

Listing 7-4. XML Definition of a VLAN Interface Associated with eth0

```
<interface type="vlan" name="eth0.42">
  <start mode="onboot"/>
  <protocol family="ipv4">
```

```
    <dhcp peerdns="no"/>
  </protocol>
  <vlan tag="42">
    <interface name="eth0"/>
  </vlan>
</interface>
```

Retrieving Information About Interfaces

You can retrieve information about network interfaces using several methods. The following examples show how to obtain that information.

Enumerating Interfaces

Once you have a connection to a host, you can determine the number of interfaces on the host with the `numOfInterfaces` and `numOfDefinedInterfaces` methods. You can obtain a list of those interfaces' names with the `listInterfaces` method and the `listDefinedInterfaces` method ("defined" interfaces are those that have been defined but are currently inactive). The list methods return a Python `list`. All four functions return `None` if an error is encountered.

Listing 7-5 shows how to get a list of the active network interfaces from a domain. Note that error handling has been omitted for clarity.

Listing 7-5. Getting a List of Active ("Up") Interfaces on a Host

```
# Example-5.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
```

```
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

ifaceNames = conn.listInterfaces()

print("Active host interfaces:")
for ifaceName in ifaceNames:
    print('  '+ifaceName)

conn.close()
exit(0)
```

Listing 7-6 shows how to get a list of the inactive domains from a domain.

Listing 7-6. Getting a List of Inactive ("Down") Interfaces on a Host

```
# Example-6.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

ifaceNames = conn.listDefinedInterfaces()

print("Inactive host interfaces:")
for ifaceName in ifaceNames:
    print('  '+ifaceName)

conn.close()
exit(0)
```

Obtaining a virInterface Instance for an Interface

Many operations require that you have an instance of `virInterface`, but you may have only the name or MAC address of the interface. You can use `interfaceLookupByName` and `interfaceLookupByMACString` to get the `virInterface` instance in these cases.

Listing 7-7 shows how to obtain a `virInterface` for a given domain interface name. Note that error handling has been omitted for clarity.

Listing 7-7. Fetching the virInterface Instance for a Given Interface Name

```
# Example-7.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.interfaceLookupByName('eth0')

print("The interface name is: "+iface.name())

conn.close()
exit(0)
```

Listing 7-8 shows how to obtain a `virInterface` for a given domain MAC address. Note that error handling has been omitted for clarity.

Listing 7-8. Fetching the virInterface Instance for a Given Interface MAC Address

```
# Example-8.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.interfaceLookupByMACString('00:01:02:03:04:05')

print("The interface name is: "+iface.name())

conn.close()
exit(0)
```

Retrieving Detailed Interface Information

You may also find yourself with a `virInterface` instance and need the name or MAC address of the interface or want to examine the full interface configuration. The `name`, `MACString`, and `XMLDesc` methods provide this capability.

Listing 7-9 shows how to obtain detailed information about a domain interface.

Listing 7-9. Fetching the name and mac Addresses from an Interface Object

```
# Example-9.py
from __future__ import print_function
```

```
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.interfaceLookupByName('eth0')

print("The interface name is: "+iface.name())
print("The interface mac string is: "+iface.MACString())

conn.close()
exit(0)
```

Listing 7-10 shows how to obtain configuration information about a domain interface.

Listing 7-10. Fetching the XML Configuration String from an Interface Object

```
# Example-10.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.interfaceLookupByName('eth0')

print("The interface XML description is:\n"+iface.XMLDesc(0))
```

```
conn.close()
exit(0)
```

Retrieving Interface Network Addresses

You may find yourself with a `virDomain` instance and need the IP addresses of one or more guest domain interfaces. The `interfaceAddresses` method provides this capability.

Listing 7-11 shows how to obtain the IP addresses of a given domain interface.

Listing 7-11. Fetching the IP Addresses of All the Guest Domain Network Interfaces

```
# Example-14.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainName = 'CentOS7'
dom = conn.lookupByName(domainName)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)

ifaces = \
  dom.interfaceAddresses(libvirt.VIR_DOMAIN_INTERFACE_
  ADDRESSES_SRC_AGENT, \
                        0)
```

```
print("The interface IP addresses:")
for (name, val) in ifaces.iteritems():
    if val['addrs']:
        for ipaddr in val['addrs']:
            if ipaddr['type'] == libvirt.VIR_IP_ADDR_TYPE_IPV4:
                print(ipaddr['addr'] + " VIR_IP_ADDR_TYPE_IPV4")
            elif ipaddr['type'] == libvirt.VIR_IP_ADDR_TYPE_IPV6:
                print(ipaddr['addr'] + " VIR_IP_ADDR_TYPE_IPV6")

conn.close()
exit(0)
```

Managing Interface Configuration Files

In libvirt, *defining* an interface means creating or changing the configuration, and *undefining* means deleting that configuration from the system. Newcomers may sometimes confuse these two operations with Create/Delete (which actually are used to activate and deactivate an existing interface; see the section called "Interface Lifecycle Management" later in this chapter).

Defining an Interface Configuration

The `interfaceDefineXML` method is used for both adding new interface configurations and modifying existing configurations. It either adds a new interface (with all information, including the interface name, given in the XML data) or modifies the configuration of an existing interface. The newly defined interface will be inactive until separate action is taken to make the new configuration take effect (for example, rebooting the host or calling Create, as described in the section "Interface Lifecycle Management").

If the interface is successfully added/modified in the host's configuration, `interfaceDefineXML` returns a `virInterface` instance. This can be used as a handle to perform further actions on the new interface, for example, making it active with `create`.

Currently, the `flags` parameter should always be `0`.

Listing 7-12 shows how to define a new interface to a domain.

Listing 7-12. Defining a New Interface

```
# Example-11.py
from __future__ import print_function
import sys
import libvirt

xml = """"
<interface type='ethernet' name='eth0'>
  <start mode='onboot'/>
  <mac address='aa:bb:cc:dd:ee:ff'/>
  <protocol family='ipv4'>
    <ip address="192.168.0.5" prefix="24"/>
    <route gateway="192.168.0.1"/>
  </protocol>
</interface>"""

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to create the interface eth0', \
          file=sys.stderr)
    exit(1)

# create/modify a network interface
iface = conn.interfaceDefineXML(xml, 0)
# activate the interface
iface.create(0)
```

```
print("The interface name is: "+iface.name())

conn.close()
exit(0)
```

The undefine method completely and permanently removes the configuration for the given interface from the host's configuration files. If you want to re-create this configuration in the future, you should invoke the XMLDesc method and save the string prior to the undefine method. Listing 7-13 shows how to undefine an interface from a domain.

Listing 7-13. Undefining the br0 Interface After Saving Its XML Data

```
# Example-12.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.interfaceLookupByName('br0')

# get the xml prior to undefining the interface
xml = iface.XMLDesc(0)
# now undefine the interface
iface.undefine()
# the interface is now undefined and the iface variable
#     is no longer be usable

conn.close()
exit(0)
```

Using changeRollback

This method rolls back all defined interface definitions since the last call to the changeCommit method.

Listing 7-14 shows how to change the rollback for a domain. Note that an error will be thrown if no interface definitions are pending.

Listing 7-14. Using changeRollback

```
# Example-30.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.changeRollback()

conn.close()
exit(0)
```

Using changeBegin

The changeBegin method creates a restore point to which one can return later by calling the changeRollback method. This function should be called before any transaction with the interface configuration. Once it is known that a new configuration works, it can be committed via the changeCommit method, which frees the restore point.

Listing 7-15 shows how to create a restore point for a domain. If the changeBegin method is called when a transaction is already opened, an error will be thrown.

Listing 7-15. Using changeBegin

```
# Example-31.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.changeBegin()


conn.close()
exit(0)
```

Using changeCommit

This method commits the changes made to interfaces and frees the restore point created by the changeBegin method.

Listing 7-16 shows how to commit a change to a domain.

Listing 7-16. Using changeCommit

```
# Example-32.py
from __future__ import print_function
import sys
import libvirt
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

conn.changeCommit()

conn.close()
exit(0)
```

The `changeRollback` method can be used to roll back an uncommited change.

Interface Lifecycle Management

In libvirt parlance, *creating* an interface means making it active, or "bringing it up," and *deleting* an interface means making it inactive, or "bringing it down." On hosts that are using the netcf back end for interface configuration, such as Fedora and Red Hat Enterprise Linux, this is the same as calling the system shell scripts `ifup` and `ifdown` for the interface.

Activating an Interface

The `create` method makes the given inactive interface active ("up"). On success, it returns 0. If there is any problem making the interface active, -1 is returned. Listing 7-12 showed a typical usage of this method.

Deactivating an Interface

The `destroy` method makes the given interface inactive ("down"). On success, it returns 0. If there is any problem making the interface active, -1 is returned.

Listing 7-17 shows how to temporarily bring down an interface in a domain.

Listing 7-17. Temporarily Bring Down eth2, Then Bring It Back Up

```
# Example-12.py
from __future__ import print_function
import sys
import libvirt

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

iface = conn.interfaceLookupByName('br0')

# get the xml prior to undefining the interface
xml = iface.XMLDesc(0)
# now undefine the interface
iface.undefine()
# the interface is now undefined and the iface variable
#     is no longer be usable

conn.close()
exit(0)
```

Summary

This chapter showed how to manipulate interfaces belonging to a domain. There are methods for creating interfaces, deleting them, obtaining information, and manipulating commits to domain interfaces.

CHAPTER 8

Error Handling

The libvirt error functions are designed to give more detailed information about what caused a failure in the case that a normal libvirt function or method returns an error. An important thing to note about Python libvirt error reporting is that errors are stored on a per-thread basis and not per connection.

The libvirtError Class

The libvirt Python module defines a standard exception class called `libvirtError` that can be subclassed to add additional functionality when raising a libvirt exception. Listing 8-1 shows a partial definition of the `libvirtError` class definition.

Listing 8-1. libvirt Module's libvirtError Class Definition

```
class libvirtError(exceptions.Exception)
    def __init__(self, defmsg, conn=None, dom=None, \
                 net=None, pool=None, vol=None):

        # Never call virGetLastError().
        # virGetLastError() is now thread local
        err = virGetLastError()
        if err is None:
            msg = defmsg
```

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_8

```
        else:
            msg = err[2]

        Exception.__init__(self, msg)

        self.err = err

    def get_error_code(self):
        if self.err is None:
            return None
        return self.err[0]

    def get_error_domain(self):
        if self.err is None:
            return None
        return self.err[1]

    def get_error_message(self):
        if self.err is None:
            return None
        return self.err[2]

    def get_error_level(self):
        if self.err is None:
            return None
        return self.err[3]

    def get_str1(self):
        if self.err is None:
            return None
        return self.err[4]

    def get_str2(self):
        if self.err is None:
            return None
        return self.err[5]

    def get_str3(self):
        if self.err is None:
            return None
        return self.err[6]

    def get_int1(self):
        if self.err is None:
            return None
        return self.err[7]

    def get_int2(self):
        if self.err is None:
            return None
        return self.err[8]
```

There are a few things to note about this definition. The first is that you can instantiate this class by using the Python `raise` statement. However, this is not much use unless you subclass the `libvirtError` class with your own class, which would contain the needed behavior. You should also make note of the methods that can obtain the error information. A description of these methods is provided here.

The method `get_error_code` returns the error code of the error. This is one of the data definitions from the Python libvirt module. Some of the higher-numbered entries from this list may not be available in your Python libvirt module.

```
VIR_ERR_OK = 0
VIR_ERR_INTERNAL_ERROR = 1    # internal error
VIR_ERR_NO_MEMORY = 2         # memory allocation failure
VIR_ERR_NO_SUPPORT = 3        # no support for this function
VIR_ERR_UNKNOWN_HOST = 4      # could not resolve hostname
```

```
VIR_ERR_NO_CONNECT = 5         # can not connect to hypervisor
VIR_ERR_INVALID_CONN = 6       # invalid connection object
VIR_ERR_INVALID_DOMAIN = 7     # invalid domain object
VIR_ERR_INVALID_ARG = 8        # invalid function argument
VIR_ERR_OPERATION_FAILED = 9   # a command to hypervisor failed
VIR_ERR_GET_FAILED = 10        # a HTTP GET command to failed
VIR_ERR_POST_FAILED = 11       # a HTTP POST command to failed
VIR_ERR_HTTP_ERROR = 12        # unexpected HTTP error code
VIR_ERR_SEXPR_SERIAL = 13      # failure to serialize an S-Expr
VIR_ERR_NO_XEN = 14            # could not open Xen hypervisor
                                 control
VIR_ERR_XEN_CALL = 15          # failure doing an hypervisor call
VIR_ERR_OS_TYPE = 16           # unknown OS type
VIR_ERR_NO_KERNEL = 17         # missing kernel information
VIR_ERR_NO_ROOT = 18           # missing root device information
VIR_ERR_NO_SOURCE = 19         # missing source device information
VIR_ERR_NO_TARGET = 20         # missing target device information
VIR_ERR_NO_NAME = 21           # missing domain name information
VIR_ERR_NO_OS = 22             # missing domain OS information
VIR_ERR_NO_DEVICE = 23         # missing domain devices information
VIR_ERR_NO_XENSTORE = 24       # could not open Xen Store control
VIR_ERR_DRIVER_FULL = 25       # too many drivers registered
VIR_ERR_CALL_FAILED = 26       # not supported by the drivers
                                 (DEPRECATED)
VIR_ERR_XML_ERROR = 27         # an XML description is not well
                                 formed or broken
VIR_ERR_DOM_EXIST = 28         # the domain already exist
VIR_ERR_OPERATION_DENIED = 29 # operation forbidden on read-
                                 only connections
VIR_ERR_OPEN_FAILED = 30       # failed to open a conf file
VIR_ERR_READ_FAILED = 31       # failed to read a conf file
```

```
VIR_ERR_PARSE_FAILED = 32        # failed to parse a conf file
VIR_ERR_CONF_SYNTAX = 33         # failed to parse the syntax
                                   of a conf file
VIR_ERR_WRITE_FAILED = 34        # failed to write a conf file
VIR_ERR_XML_DETAIL = 35          # detail of an XML error
VIR_ERR_INVALID_NETWORK = 36     # invalid network object
VIR_ERR_NETWORK_EXIST = 37       # the network already exist
VIR_ERR_SYSTEM_ERROR = 38        # general system call failure
VIR_ERR_RPC = 39                 # some sort of RPC error
VIR_ERR_GNUTLS_ERROR = 40        # error from a GNUTLS call
VIR_WAR_NO_NETWORK = 41          # failed to start network
VIR_ERR_NO_DOMAIN = 42           # domain not found or
                                   unexpectedly disappeared
VIR_ERR_NO_NETWORK = 43          # network not found
VIR_ERR_INVALID_MAC = 44         # invalid MAC address
VIR_ERR_AUTH_FAILED = 45         # authentication failed
VIR_ERR_INVALID_STORAGE_POOL = 46 # invalid storage pool object
VIR_ERR_INVALID_STORAGE_VOL = 47 # invalid storage vol object
VIR_WAR_NO_STORAGE = 48          # failed to start storage
VIR_ERR_NO_STORAGE_POOL = 49     # storage pool not found
VIR_ERR_NO_STORAGE_VOL = 50      # storage pool not found
VIR_WAR_NO_NODE = 51             # failed to start node driver
VIR_ERR_INVALID_NODE_DEVICE = 52 # invalid node device object
VIR_ERR_NO_NODE_DEVICE = 53      # node device not found
VIR_ERR_NO_SECURITY_MODEL = 54   # security model not found
VIR_ERR_OPERATION_INVALID = 55   # operation is not applicable
                                   at this time
VIR_WAR_NO_INTERFACE = 56        # failed to start interface
                                   driver
VIR_ERR_NO_INTERFACE = 57        # interface driver not running
VIR_ERR_INVALID_INTERFACE = 58   # invalid interface object
```

```
VIR_ERR_MULTIPLE_INTERFACES = 59       # more than one matching
                                         interface found
VIR_WAR_NO_NWFILTER = 60               # failed to start nwfilter
                                         driver
VIR_ERR_INVALID_NWFILTER = 61          # invalid nwfilter object
VIR_ERR_NO_NWFILTER = 62               # nw filter pool not found
VIR_ERR_BUILD_FIREWALL = 63            # nw filter pool not found
VIR_WAR_NO_SECRET = 64                 # failed to start secret
                                         storage
VIR_ERR_INVALID_SECRET = 65            # invalid secret
VIR_ERR_NO_SECRET = 66                 # secret not found
VIR_ERR_CONFIG_UNSUPPORTED = 67        # unsupported configuration
                                         construct
VIR_ERR_OPERATION_TIMEOUT = 68         # timeout occurred during
                                         operation
VIR_ERR_MIGRATE_PERSIST_FAILED = 69    # a migration worked, but
                                         making the VM persist on
                                         the dest host failed
VIR_ERR_HOOK_SCRIPT_FAILED = 70        # a synchronous hook script
                                         failed
VIR_ERR_INVALID_DOMAIN_SNAPSHOT = 71   # invalid domain snapshot
VIR_ERR_NO_DOMAIN_SNAPSHOT = 72        # domain snapshot was not
                                         found
VIR_ERR_INVALID_STREAM = 73            # invalid i/o stream
VIR_ERR_ARGUMENT_UNSUPPORTED = 74      # an argument was unsupported
VIR_ERR_STORAGE_PROBE_FAILED = 75      # a storage probe failed
VIR_ERR_STORAGE_POOL_BUILT = 76
VIR_ERR_SNAPSHOT_REVERT_RISKY = 77
VIR_ERR_OPERATION_ABORTED = 78         # the operation was aborted
VIR_ERR_AUTH_CANCELLED = 79
VIR_ERR_NO_DOMAIN_METADATA = 80        # no domain metadata was
                                         found
```

```
VIR_ERR_MIGRATE_UNSAFE = 81
VIR_ERR_OVERFLOW = 82              # an overflow situation was
                                     detected
VIR_ERR_BLOCK_COPY_ACTIVE = 83
VIR_ERR_OPERATION_UNSUPPORTED = 84 # the operation was unsupported
VIR_ERR_SSH = 85                   # an ssh error was detected
VIR_ERR_AGENT_UNRESPONSIVE = 86    # an agent timeout was detected
VIR_ERR_RESOURCE_BUSY = 87
VIR_ERR_ACCESS_DENIED = 88
VIR_ERR_DBUS_SERVICE = 89
VIR_ERR_STORAGE_VOL_EXIST = 90
VIR_ERR_CPU_INCOMPATIBLE = 91
VIR_ERR_XML_INVALID_SCHEMA = 92
```

The method get_error_domain is named as such for legacy reasons, but it really represents which part of libvirt generated the error. This is one of the data definitions from the Python libvirt module. Some of the higher-numbered entries from this list may not be available in your Python libvirt module. The full list is as follows:

```
VIR_FROM_NONE = 0
VIR_FROM_XEN = 1             # Error at Xen hypervisor layer
VIR_FROM_XEND = 2            # Error at connection with xend
                               daemon
VIR_FROM_XENSTORE = 3        # Error at connection with xen store
VIR_FROM_SEXPR = 4           # Error in the S-Expression code
VIR_FROM_XML = 5             # Error in the XML code
VIR_FROM_DOM = 6             # Error when operating on a domain
VIR_FROM_RPC = 7             # Error in the XML-RPC code
VIR_FROM_PROXY = 8           # Error in the proxy code
VIR_FROM_CONF = 9            # Error in the configuration file
                               handling
VIR_FROM_QEMU = 10           # Error at the QEMU daemon
```

```
VIR_FROM_NET = 11             # Error when operating on a network
VIR_FROM_TEST = 12            # Error from test driver
VIR_FROM_REMOTE = 13          # Error from remote driver
VIR_FROM_OPENVZ = 14          # Error from OpenVZ driver
VIR_FROM_XENXM = 15           # Error at Xen XM layer
VIR_FROM_STATS_LINUX = 16     # Error in the Linux Stats code
VIR_FROM_LXC = 17             # Error from Linux Container driver
VIR_FROM_STORAGE = 18         # Error from storage driver
VIR_FROM_NETWORK = 19         # Error from network config
VIR_FROM_DOMAIN = 20          # Error from domain config
VIR_FROM_UML = 21             # Error at the UML driver
VIR_FROM_NODEDEV = 22         # Error from node device monitor
VIR_FROM_XEN_INOTIFY = 23     # Error from xen inotify layer
VIR_FROM_SECURITY = 24        # Error from security framework
VIR_FROM_VBOX = 25            # Error from VirtualBox driver
VIR_FROM_INTERFACE = 26       # Error when operating on an
                                interface
VIR_FROM_ONE = 27             # Error from OpenNebula driver
VIR_FROM_ESX = 28             # Error from ESX driver
VIR_FROM_PHYP = 29            # Error from IBM power hypervisor
VIR_FROM_SECRET = 30          # Error from secret storage
VIR_FROM_CPU = 31             # Error from CPU driver
VIR_FROM_XENAPI = 32          # Error from XenAPI
VIR_FROM_NWFILTER = 33        # Error from network filter driver
VIR_FROM_HOOK = 34            # Error from Synchronous hooks
VIR_FROM_DOMAIN_SNAPSHOT = 35 # Error from domain snapshot
VIR_FROM_AUDIT = 36
VIR_FROM_SYSINFO = 37
VIR_FROM_STREAMS = 38
VIR_FROM_VMWARE = 39
VIR_FROM_EVENT = 40
VIR_FROM_LIBXL = 41
```

```
VIR_FROM_LOCKING = 42
VIR_FROM_HYPERV = 43
VIR_FROM_CAPABILITIES = 44
VIR_FROM_URI = 45
VIR_FROM_AUTH = 46
VIR_FROM_DBUS = 47
VIR_FROM_PARALLELS = 48
VIR_FROM_DEVICE = 49
VIR_FROM_SSH = 50
VIR_FROM_LOCKSPACE = 51
VIR_FROM_INITCTL = 52
VIR_FROM_IDENTITY = 53
VIR_FROM_CGROUP = 54
VIR_FROM_ACCESS = 55
VIR_FROM_SYSTEMD = 56
VIR_FROM_BHYVE = 57
VIR_FROM_CRYPTO = 58
VIR_FROM_FIREWALL = 59
VIR_FROM_POLKIT = 60
```

The method `get_error_message` is a human-readable string describing the error.

The method `get_error_level` describes the severity of the error. This is one of the data definitions from the Python libvirt module. The full list of levels is as follows:

```
VIR_ERR_NONE = 0
VIR_ERR_WARNING = 1  # A simple warning
VIR_ERR_ERROR = 2    # An error
```

The method `get_error_str1` gives extra human-readable information.

The method `get_error_str2` gives extra human-readable information.

The method `get_error_str3` gives extra human-readable information.

The method `get_error_int1` gives extra numeric information that may be useful for further classifying the error.

The method `get_error_int2` gives extra numeric information that may be useful for further classifying the error.

Example code that uses various parts of this structure will be presented in subsequent subsections.

Using virGetLastError

The `virGetLastError` function can be used to obtain a Python `list` that contains all the information from the error reported from libvirt. This information is kept in thread-local storage, so separate threads can safely use this function concurrently. Note that it does not make a copy, so error information can be lost if the current thread calls this function subsequently. Listing 8-2 demonstrates the use of `virGetLastError`.

Listing 8-2. Using virGetLastError

```
# Example-24.py
from __future__ import print_function
import sys
import libvirt

def report_libvirt_error():
    """Call virGetLastError function to get the last error
    information."""
    err = libvirt.virGetLastError()
    print('Error code:    '+str(err[0]), file=sys.stderr)
    print('Error domain:  '+str(err[1]), file=sys.stderr)
    print('Error message: '+str(err[2]), file=sys.stderr)
    print('Error level:   '+str(err[3]), file=sys.stderr)
    if err[4] != None:
        print('Error string1: '+str(err[4]), file=sys.stderr)
    else:
        print('Error string1:', file=sys.stderr)
    if err[5] != None:
        print('Error string2: '+str(err[5]), file=sys.stderr)
    else:
        print('Error string2:', file=sys.stderr)
    if err[6] != None:
        print('Error string3: '+str(err[6]), file=sys.stderr)
    else:
        print('Error string3:', file=sys.stderr)
    print('Error int1:    '+str(err[7]), file=sys.stderr)
    print('Error int2:    '+str(err[8]), file=sys.stderr)
    exit(1)

try:
    conn = libvirt.open('qemu:///system') # invalidate the
           parameter to force an error
except:
    report_libvirt_error()
conn.close()
exit(0)
```

Subclassing libvirtError

It is possible to subclass the `libvirtError` class to add functionality. The default `libvirtError` does not provide any ability to either save the error information or present the information to the user. Subclassing `libvirtError` gives the programmer the flexibility to add any functionality needed. Listing 8-3 shows an example of this.

Listing 8-3. Subclassing libvirtError

```
# Example-25.py
from __future__ import print_function
import sys
import libvirt

class report_libvirt_error(libvirt.libvirtError):
    """Subclass virError to get the last error information."""
    def __init__(self, defmsg, conn=None, dom=None, net=None, \
                 pool=None, vol=None):
        libvirt.libvirtError.__init__(self, defmsg, conn=None, \
                                      dom=None, net=None,
                                      pool=None, \
                                      vol=None)
        print('Default msg:   '+str(defmsg), file=sys.stderr)
        print('Error code:    '+str(self.get_error_code()),
                                     file=sys.stderr)
        print('Error domain:  '+str(self.get_error_domain()),
                                     file=sys.stderr)
        print('Error message: '+self.get_error_message(),
                                 file=sys.stderr)
        print('Error level:   '+str(self.get_error_level()),
                                 file=sys.stderr)
        if self.err[4] != None:
            print('Error string1: '+self.get_str1(),
            file=sys.stderr)
        else:
            print('Error string1:', file=sys.stderr)
        if self.err[5] != None:
            print('Error string2: '+self.get_str2(),
            file=sys.stderr)
        else:
            print('Error string2:', file=sys.stderr)
        if self.err[6] != None:
            print('Error string3: '+self.get_str3(),
            file=sys.stderr)
        else:
            print('Error string3:', file=sys.stderr)
        print('Error int1:    '+str(self.get_int1()),
                               file=sys.stderr)
        print('Error int2:    '+str(self.get_int2()),
                               file=sys.stderr)
        exit(1)

try:
    conn = libvirt.open('qemu:///system') # invalidate the parameter
                                          # to force an error
except libvirt.libvirtError:
    raise report_libvirt_error('Connection error')
conn.close()
exit(0)
```

Registering an Error Handler Function

libvirt also supports setting up an error handler Python function. This is done using the libvirt function `registerErrorHandler`, which returns 1 in the case of success.

The registered function is called as `f(ctx, error)`, with `error` being a list of information about the error being raised.

Listing 8-4 shows an example of this function.

Listing 8-4. Registering an Error Handler

```
# Example-26.py
from __future__ import print_function
import sys
import libvirt

def libvirt_error_handler(ctx, err):
    print('Error code:    '+str(err[0]), file=sys.stderr)
    print('Error domain:  '+str(err[1]), file=sys.stderr)
    print('Error message: '+err[2], file=sys.stderr)
    print('Error level:   '+str(err[3]), file=sys.stderr)
    if err[4] != None:
        print('Error string1: '+err[4], file=sys.stderr)
    else:
        print('Error string1:', file=sys.stderr)
    if err[5] != None:
        print('Error string2: '+err[5], file=sys.stderr)
    else:
        print('Error string2:', file=sys.stderr)
    if err[6] != None:
        print('Error string3: '+err[6], file=sys.stderr)
    else:
        print('Error string3:', file=sys.stderr)
    print('Error int1:    '+str(err[7]), file=sys.stderr)
    print('Error int2:    '+str(err[8]), file=sys.stderr)
    exit(1)

ctx = 'just some information'
libvirt.registerErrorHandler(libvirt_error_handler, ctx)
```

```
conn = libvirt.open('qemu:///system') # invalidate the parameter
                                      # to force an error

conn.close()
exit(0)
```

Summary

This chapter concentrated on the errors that can be raised by libvirt. You examined the default error class and saw how this class can be extended to include other information. Understanding how to use the error class is key to building a program that can withstand errors that may occur during execution and possibly include automatic error correction.

CHAPTER 9

Event and Timer Handling

The Python libvirt module provides a complete interface for handling both events and timers. Both event handling and timer handling are invoked through a function interface as opposed to a class/method interface. This makes it easier to integrate the interface into either a graphical or console program.

Event Handling

The Python libvirt module supplies a framework for event handling. While this is most useful for graphical programs, it can also be used for console programs to provide a consistent user interface and control the processing of console events.

Event handling is done through the functions `virEventAddHandle`, `virEventRegisterDefaultImpl`, `virEventRegisterImpl`, `virEventRemoveHandle`, `virEventRunDefaultImpl`, and `virEventUpdateHandle`.

Creating an event requires that an event loop has previously been registered with `virEventRegisterImpl` or `virEventRegisterDefaultImpl`.

Listing 9-1 is an example program that uses most of these functions.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_9

Listing 9-1. Providing a Persistent Console That Survives Guest Reboots

```
# Example-1.py
# consolecallback - provide a persistent console that survives
#    guest reboots
from __future__ import print_function

import sys, os, logging, libvirt, tty, termios, atexit

def reset_term():
    termios.tcsetattr(0, termios.TCSADRAIN, attrs)

def error_handler(unused, error):
    # The console stream errors on VM shutdown; we don't care
    if (error[0] == libvirt.VIR_ERR_RPC and
        error[1] == libvirt.VIR_FROM_STREAMS):
        return
    logging.warn(error)

class Console(object):
    def __init__(self, uri, uuid):
        self.uri = uri
        self.uuid = uuid
        self.connection = libvirt.open(uri)
        self.domain = self.connection.lookupByUUIDString(uuid)
        self.state = self.domain.state(0)
        self.connection.domainEventRegister(lifecycle_callback, self)
        self.stream = None
        self.run_console = True
        logging.info("%s initial state %d, reason %d",
                     self.uuid, self.state[0], self.state[1])
```

```
def check_console(console):
    if (console.state[0] == libvirt.VIR_DOMAIN_RUNNING or
        console.state[0] == libvirt.VIR_DOMAIN_PAUSED):
        if console.stream is None:
            console.stream = \
               console.connection.newStream(libvirt.VIR_STREAM_
               NONBLOCK)
            console.domain.openConsole(None, console.stream, 0)
            console.stream.eventAddCallback(libvirt.VIR_STREAM_
            EVENT_READABLE, \
                                           stream_callback, console)
    else:
        if console.stream:
            console.stream.eventRemoveCallback()
            console.stream = None

    return console.run_console

def stdin_callback(watch, fd, events, console):
    readbuf = os.read(fd, 1024)
    if readbuf.startswith(""):
        console.run_console = False
        return
    if console.stream:
        console.stream.send(readbuf)

def stream_callback(stream, events, console):
    try:
        received_data = console.stream.recv(1024)
    except:
        return
    os.write(0, received_data)
```

```
def lifecycle_callback (connection, domain, event, detail, console):
    console.state = console.domain.state(0)
    logging.info("%s transitioned to state %d, reason %d",
                 console.uuid, console.state[0], console.state[1])

# main
if len(sys.argv) != 3:
    print("Usage:", sys.argv[0], "URI UUID")
    print("for example:", sys.argv[0], \
          "'qemu:///system' '32ad945f-7e78-c33a-e96d-39f25e025d81'")
    sys.exit(1)

uri = sys.argv[1]
uuid = sys.argv[2]

print("Escape character is ^]")
logging.basicConfig(filename='msg.log', level=logging.DEBUG)
logging.info("URI: %s", uri)
logging.info("UUID: %s", uuid)

libvirt.virEventRegisterDefaultImpl()
libvirt.registerErrorHandler(error_handler, None)

atexit.register(reset_term)
attrs = termios.tcgetattr(0)
tty.setraw(0)

console = Console(uri, uuid)
console.stdin_watch = libvirt.virEventAddHandle(0, \
              libvirt.VIR_EVENT_HANDLE_READABLE,
              stdin_callback, console)

while check_console(console):
    libvirt.virEventRunDefaultImpl()
```

This example shows how to handle events, but it has one problematic issue. There is no way to exit the program easily. You are reduced to using the command line to exit the program. I will leave it to you to figure out how to make this happen without interrupting the program.

Timer Handling

The Python libvirt module supplies a framework for timer handling. Creating a timer requires that an event loop has previously been registered with `virEventRegisterImpl` or `virEventRegisterDefaultImpl`.

Timer handling is done through the functions `virEventAddTimeout`, `virEventUdateTimeout`, and `virEventRemoveTimeout`. The implementation will support many timers.

To create a new timer, call `VirEventAddTimout` after `virEventRegisterImpl` or the `virEventRegisterDefaultImpl` function has been invoked.

The timer can be removed using `VirEventRemoveTimout` or updated with the `virEventUpdateTimeout` function after it has been added. See Listing 9-2.

Listing 9-2. Example Timer Implementation

```
# Example-2.py
import libvirt
import time
import threading

def callback(conn, dom, event, detail, opaque):
    print "EVENT: Domain %s(%s) %s %s" % (dom.name(), dom.ID(),
                                          event, detail)

eventLoopThread = None
```

```
def virEventLoopNativeRun():
    while True:
        libvirt.virEventRunDefaultImpl()

def virEventLoopNativeStart():
    global eventLoopThread
    libvirt.virEventRegisterDefaultImpl()
    eventLoopThread = threading.Thread(target=virEventLoopNativeRun,
                                       name="libvirtEventLoop")
    eventLoopThread.setDaemon(True)
    eventLoopThread.start()

virEventLoopNativeStart()

conn = libvirt.openReadOnly('qemu:///system')

conn.domainEventRegister(callback, None)
conn.setKeepAlive(5, 3)

while conn.isAlive() == 1:
    time.sleep(1)
```

Summary

This chapter introduced the topic of event handling. You learned how to capture and handle events using Python functions created just for that purpose. The chapter also introduced the topic of event timers and how to deal with them.

CHAPTER 10

Using the QEMU Guest Agent

When the libvirt project was first released, the scope was limited to the external control of guest domains. It became clear in a short time that to fully administer a guest domain it would be necessary to communicate with the domain operating system to query dynamic runtime information and issue commands to the OS.

The best way to accomplish this would be to create an OS daemon to handle communication with the external libvirt system and perform all the necessary requested operations and queries. The QEMU Guest Agent was created to perform those functions. The agent was developed and is maintained by the libvirt development team and was designed to be an extension of the libvirt library. While it has a documented interface with many different useful functions, most of them are really meant to be wrapped by the libvirt library. In other words, libvirt functions call the agent interfaces on behalf of the user.

However, there is a set of functions that are useful to the programmer/administrator. This chapter will concentrate on that set of functions and the two common methods of invoking them.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_10

Installing the QEMU Guest Agent

If the guest domain is a modern OS distribution, the QEMU Guest Agent will be automatically installed by the distribution install program during the OS installation. The OS installation program will detect the virtual environment and automatically install the QEMU Guest Agent.

Most of the time the OS install program will take care of everything related to the guest domain. But if the distribution is very old, then you will need to build the QEMU Guest Agent from the source code and manually install it yourself. The source code is available at `www.qemu.org/download/`.

Using the QEMU Guest Agent

Since most functions provided by the QEMU Guest Agent are meant to be wrapped by the libvirt library, this chapter will not attempt to document or describe all the available functions and commands that are used by the agent. The main thing to remember is to not disable the agent on the guest domain! If you do, many libvirt functions and methods will suddenly begin to fail or to supply incomplete information.

Unfortunately, the libvirt development team has not documented which libvirt functions and methods make use of the QEMU Guest Agent. So, you should always assume that the agent needs to be running at all times on all active guest domains. The QEMU commands are documented in the QEMU Guest Agent Protocol Reference at `https://qemu.weilnetz.de/doc/qemu-ga-ref.html`.

There are two preferred methods to invoking these commands: through virsh or through the libvirt API. Both methods are easy to use but require the command to be wrapped in a JSON schema/IDL.

Warning When a command is sent directly to the QEMU Guest Agent, you are bypassing the libvirt library. It is therefore possible to cause the guest domain to become “out of sync” with the information maintained by libvirt. For instance, you could create a new device or delete an existing device through the Guest Agent and libvirt would not receive any notification concerning the change of state in the guest domain. This is because under normal conditions only libvirt should be calling the QEMU Guest Agent.

Be careful! Use the QEMU Guest Agent functions only after eliminating all the other alternatives.

Before submitting commands to the QEMU Guest Agent, you must first discover what commands the Guest Agent supports. Not all guest agents support all the available commands, but the QEMU Guest Agent knows what it does support and can return that information to the caller through the `guest-info` command. This should be the first command issued to the Guest Agent. After you check the list of returned commands, you are free to send other commands to the Guest Agent as long as they exist in the list.

Most of the commands I will be using in the examples are simple, with either one or zero arguments. For more complex commands, you will need advanced knowledge of JSON containers. When a command succeeds, it usually returns an empty JSON container.

virsh and the QEMU Guest Agent

The `virsh command`/environment can be used to issue commands to a libvirt domain's QEMU Guest Agent (QEMU-GA). The `virsh` subcommand `qemu-agent-command` directs that the following QEMU command be sent to the specified domain of QEMU-GA. The syntax for this command is as follows:

```
virsh qemu-agent-command domain json-wrapper option-arg
```

The `virsh` and `qemu-agent-command` keywords are constants specifying the command name and the subcommand name. These keywords are required for all QEMU-GA commands.

`domain` specifies the name or other identifier for the libvirt domain that should receive the QEMU-GA command. The `domain` argument is required.

`json-wrapper` is a placeholder for the JSON container containing the command to be executed by the QEMU-GA. The `json-wrapper` argument is required.

`option-arg` specifies one or more keyword arguments. These arguments are preceded with double hyphens (`--`). These arguments are optional.

All `qemu-agent-command` subcommands send their return output to `stdout`. If you need to parse this output, you will need to capture it and use some method to parse it. The output is always a JSON container. If the command succeeds, the output is usually an empty JSON container.

As usual, the first QEMU-GA command you need to send to a libvirt domain is `guest-info`, which will return all the QEMU-GA commands supported by the installed Guest Agent.

```
virsh qemu-agent-command centos7.0 '{"execute": "guest-info"}'
--pretty
```

The output from this command is partially listed here:

```
{
  "return": {
    "version": "2.8.0",
    "supported_commands": [
      {
        "enabled": true,
        "name": "guest-sync-delimited",
        "success-response": true
      },
      {
        "enabled": true,
        "name": "guest-sync",
        "success-response": true
      },
      {
        "enabled": true,
        "name": "guest-suspend-ram",
        "success-response": false
      },
              .
              .                        <--- Many lines not shown
              .
      {
        "enabled": false,
        "name": "guest-exec-status",
        "success-response": true
      },
      {
        "enabled": false,
        "name": "guest-exec",
        "success-response": true
      }
    ]
  }
}
```

The returned JSON container specifies the name of the commands that are supported, whether the command is enabled or not, and whether the command sends a response to the caller. Obviously, you will need to parse this output to make sense of it.

One of the interesting things to notice about the output is whether a command returns any output. In every case where no output is sent, the command is causing a change in the run state of the domain. For instance, if the domain ram is suspended or the domain is shut down, it makes no sense to return any output as it will cause the state of the domain to be reverted or delayed.

Now that you can determine whether a command is supported by the Guest Agent, let's look at some commonly supported commands in detail and how they can be used to enhance the applications running on the guest domain.

QEMU Guest Agent Time Commands

The QEMU time commands `guest-get-time` and `guest-set-time` are useful if you ever save or restore a running guest domain. When you save a running domain, all states of the system are saved, including the current time and date. When you then restore the domain to a running state, the saved date/time is also restored. In other words, the current time is not fetched from the hardware clock set in the operating system. This means that the clock in the guest domain may be radically different from the host node after the guest domain is restored.

This is different from a domain that is shut down and then restarted. During startup of the domain, the hardware clock is read and used to set the operating system clock. Then NTP may be used to keep the OS clock in sync with the world clock.

Note You should not depend on NTP to resync the OS clock after a save and restore is performed. The difference between the restored time and the current time may be too great for NTP to work with, and the time difference may not ever change. In fact, you should not depend on NTP at all in this circumstance because NTP works only on small interval differences.

First, let's determine the date/time the guest domain believes is the current time.

```
virsh qemu-agent-command centos7.0 '{"execute": "guest-get-time"}'
```

This command always returns the guest domain time/date as the number of microseconds from the Unix/Linux/iOS/Windows epoch (January 1, 1970, midnight UTC/GMT).

```
{"return":1535650424741255000}
```

The returned value translates to the human-readable form "Thursday, August 30, 2018 5:33:44.741 PM GMT." You can compare the time returned from the guest domain to the host node to find out how big the difference is between the two entities. Then you can make a decision to either deal with it or ignore it.

If you want to modify the time/date of the guest domain, you can do it via the GEMU-GA command that follows:

```
virsh qemu-agent-command centos7.0 '{"execute": "guest-set-time"}'
```

This command has an optional argument to specify the time/date to be set in the guest domain. If the argument is not supplied, then the real-time clock will be queried for the time to be set. This applies to all operating systems except Windows, which has no visible user interface to the real-time clock. You will always need to supply the time to be set as an argument when working with the Windows OS.

The returned value from the command is always an empty JSON container.

```
{"return":{}}
```

To determine whether the command worked, you will need to retrieve the time from the guest domain again and compare that to the host node.

QEMU Guest Agent Network Interface Commands

The QEMU network interface command `guest-network-get-interfaces` can supply some basic information about all the available guest domain network interfaces. The main thing you usually want is the IP address of the interface (IPv4 and/or IPv6).

Let's take a look at `guest-network-get-interfaces`.

```
virsh qemu-agent-command centos7.0 '{"execute": "guest-network-
get-interfaces"}' --pretty
```

This command always returns information about each available network interface. The result is always a JSON container.

```
{
  "return": [
    {
      "name": "lo",
      "ip-addresses": [
        {
```

```
        "ip-address-type": "ipv4",
        "ip-address": "127.0.0.1",
        "prefix": 8
      },
      {
        "ip-address-type": "ipv6",
        "ip-address": "::1",
        "prefix": 128
      }
    ],
    "hardware-address": "00:00:00:00:00:00"
  },
  {
    "name": "eth0",
    "ip-addresses": [
      {
        "ip-address-type": "ipv4",
        "ip-address": "192.168.122.63",
        "prefix": 24
      },
      {
        "ip-address-type": "ipv6",
        "ip-address": "fe80::fb07:2e01:f94:bc7d",
        "prefix": 64
      }
    ],
    "hardware-address": "52:54:00:b5:32:27"
  },
  {
    "name": "virbr0",
    "ip-addresses": [
        {
          "ip-address-type": "ipv4",
          "ip-address": "192.168.124.1",
          "prefix": 24
        }
      ],
      "hardware-address": "52:54:00:fd:e9:b7"
    },
    {
      "name": "virbr0-nic",
      "hardware-address": "52:54:00:fd:e9:b7"
    }
  ]
}
```

You will need to parse the JSON container to make sense of the information. The information in the JSON container includes the interface name, the IPv4 and IPv6 addresses, and the hardware interface address.

The information returned should be considered dynamic. An interface could be removed or added at any time, so this command should be used to discover added or removed network interfaces when you receive an error on an interface.

Python and the QEMU Guest Agent

Python also has a function available that can access the `qemu-agent-command` environment. The syntax for this function is as follows:

```
ret = qemuAgentCommand(domain, cmd, timeout, flags)
```

This function is QEMU specific, so it will work only with hypervisor connections to the QEMU Guest Agent driver.

The `domain` argument specifies the name or other identifier for the libvirt domain that should receive the QEMU-GA command. The `domain` argument is required and must be available as an instance of the libvirt `Domain` class.

The `cmd` part is a placeholder for the JSON container containing the command to be executed by the QEMU-GA. The `cmd` part is required.

The `timeout` argument takes one of the following values:

- `VIR_DOMAIN_QEMU_AGENT_COMMAND_BLOCK` blocks until the command is complete.
- `VIR_DOMAIN_QEMU_AGENT_COMMAND_MIN` blocks a minimal amount of time.
- `VIR_DOMAIN_QEMU_AGENT_COMMAND_DEFAULT` blocks the default amount of time.
- `VIR_DOMAIN_QEMU_AGENT_COMMAND_NOWAIT` does not block at all.
- `VIR_DOMAIN_QEMU_AGENT_COMMAND_SHUTDOWN` blocks until OS shutdown.

The `flags` argument currently has no options and should always be zero. In the future, there may be options available for this argument.

As usual, the first QEMU-GA command you need to send to a libvirt domain is `guest-info`, which will return all the QEMU-GA commands supported by the installed Guest Agent. Listing 10-1 shows how to obtain a listing of the supported commands.

Listing 10-1. Get the QEMU-GA guest-info

```
# Example-01.py
from __future__ import print_function
import sys
import libvirt, libvirt_qemu
```

```
conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainName = 'centos7.0'
dom = conn.lookupByName(domainName)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)

ret_json = libvirt_qemu.qemuAgentCommand(dom, '{"execute":
"guest-info"}',
                libvirt_qemu.VIR_DOMAIN_QEMU_AGENT_COMMAND_
                DEFAULT, 0)
if ret_json == None:
    print('Failed to get the JSON object', file=sys.stderr)
else:
    print(ret_json)

conn.close()
```

The output from this function is partially listed here:

```
{"return":{"version":"2.8.0","supported_commands":[{"enabled":
true,"name":
"guest-sync-delimited","success-response":true},{"enabled":true
,"name":
"guest-sync","success-response":true},{"enabled":true,"name":
"guest-suspend-ram","success-response":false},{"enabled":true,"name":
"guest-suspend-hybrid","success-response":false},{"enabled":true,
"name":
```

```
"guest-suspend-disk","success-response":false},{"enabled":true,"name":
              .
              .                      <--- Many lines not shown
              .
"guest-file-read","success-response":true},{"enabled":false,"name":
"guest-file-open","success-response":true},{"enabled":false,"name":
"guest-file-flush","success-response":true},{"enabled":false,"name":
"guest-file-close","success-response":true},{"enabled":false,"name":
"guest-exec-status","success-response":true},{"enabled":false,"name":
"guest-exec","success-response":true}]}}
```

The returned JSON container specifies the name of the commands that are supported, whether the command is enabled or not, and whether the command sends a response to the caller. Obviously, you will need to parse this output to make sense of it.

One of the interesting things to notice about the output is whether a command returns any output. In every case where no output is sent, the command is causing a change in the run state of the domain. For instance, if the domain RAM is suspended or the domain is shut down, it makes no sense to return any output because it will cause the state of the domain to be reverted or delayed.

QEMU Guest Agent Time Commands

The QEMU time commands `guest-get-time` and `guest-set-time` are useful if you ever save or restore a running guest domain. When you save a running domain, all states of the system are saved, including the current time and date. When you then restore the domain to a running state, the saved date/time is also restored; in other words, the current time is not fetched from the hardware clock set in the operating system. This means that the clock in the guest domain may be radically different from the host node after the guest domain is restored.

This is different from a domain that is shut down and then restarted. During startup of the domain, the hardware clock is read and used to set the operating system clock. Then NTP cans be used to keep the OS clock in sync with the world clock.

Note You should not depend on NTP to resync the OS clock after a save and restore is performed. The difference between the restored time and the current time may be too great for NTP to work with, and the time difference may not ever change.

First, let's determine the date/time the guest domain believes is the current time.

Listing 10-2 shows how to return the time on a guest domain using the QEMU interface.

Listing 10-2. Get the QEMU-GA guest-get-time

```
# Example-02.py
from __future__ import print_function
import sys
import libvirt, libvirt_qemu

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)
```

```
domainName = 'centos7.0'
dom = conn.lookupByName(domainName)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)

ret_json = libvirt_qemu.qemuAgentCommand(dom, '{"execute":
"guest-get-time"}',
                libvirt_qemu.VIR_DOMAIN_QEMU_AGENT_COMMAND_
                DEFAULT, 0)
if ret_json == None:
    print('Failed to get the JSON object', file=sys.stderr)
else:
    print(ret_json)

conn.close()
```

This command always returns the guest domain time/date as the number of microseconds from the Unix/Linux/iOS/Windows epoch (January 1, 1970, midnight UTC/GMT).

```
{"return":1535650424741255000}
```

The returned value translates to the human-readable form "Thursday, August 30, 2018 5:33:44.741 PM GMT." You can compare the time returned from the guest domain to the host node to find out how big the difference is between the two entities. Then you can make a decision to either deal with it or ignore it.

If you want to modify the time/date of the guest domain, you can do it via the GEMU-GA command in Listing 10-3.

Listing 10-3. Set the QEMU-GA guest-set-time

```
# Example-03.py
from __future__ import print_function
import sys
import libvirt, libvirt_qemu

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainName = 'centos7.0'
dom = conn.lookupByName(domainName)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)

ret_json = libvirt_qemu.qemuAgentCommand(dom, '{"execute":
"guest-set-time"}',
                libvirt_qemu.VIR_DOMAIN_QEMU_AGENT_COMMAND_
                DEFAULT, 0)
if ret_json == None:
    print('Failed to get the JSON object', file=sys.stderr)
else:
    print(ret_json)

conn.close()
```

This command has an optional argument to specify the time/date to be set in the guest domain. If the argument is not supplied, then the real-time clock will be queried for the time to be set. This applies to all operating systems except Windows, which has no visible user interface to the real-time clock. You will always need to supply the time to be set as an argument when working with the Windows OS.

The returned value from the command is always an empty JSON container.

```
{"return":{}}
```

To determine whether the command worked, you will need to retrieve the time from the guest domain again and compare that to the host node.

QEMU Guest Agent Network Interface Commands

The QEMU network interface command `guest-network-get-interfaces` can supply some basic information about all the available guest domain network interfaces. The main thing you usually want is the IP address of the interface (IPv4 and/or IPv6).

Let's take a look at the `guest-network-get-interfaces` in Listing 10-4.

Listing 10-4. Get the QEMU-GA guest-network-get-interfaces

```
# Example-03.py
from __future__ import print_function
import sys
import libvirt, libvirt_qemu

conn = libvirt.open('qemu:///system')
if conn == None:
    print('Failed to open connection to qemu:///system', \
          file=sys.stderr)
    exit(1)

domainName = 'centos7.0'
dom = conn.lookupByName(domainName)
if dom == None:
    print('Failed to get the domain object', file=sys.stderr)
```

```
ret_json = libvirt_qemu.qemuAgentCommand(dom,
                '{"execute": " guest-network-get-interfaces”}',
                libvirt_qemu.VIR_DOMAIN_QEMU_AGENT_COMMAND_
                DEFAULT, 0)
if ret_json == None:
    print('Failed to get the JSON object', file=sys.stderr)
else:
    print(ret_json)

conn.close()
```

This command always returns information about each available network interface. The result is always a JSON container.

```
{"return":[{"name":"lo","ip-addresses":[{"ip-address-type":"ipv4",
"ip-address":"127.0.0.1","prefix":8},{"ip-address-type":"ipv6",
"ip-address":"::1","prefix":128}],"hardware-
address":"00:00:00:00:00:00"},
{"name":"eth0","ip-addresses":[{"ip-address-type":"ipv4",
"ip-address":
"192.168.122.63","prefix":24},{"ip-address-type":"ipv6",
"ip-address":
"fe80::fb07:2e01:f94:bc7d","prefix":64}],"hardware-address":
"52:54:00:b5:32:27"},{"name":"virbr0","ip-addresses":
[{"ip-address-type":
"ipv4","ip-address":"192.168.124.1","prefix":24}],
"hardware-address":
"52:54:00:fd:e9:b7"},{"name":"virbr0-nic","hardware-address":
"52:54:00:fd:e9:b7"}]}
```

You will need to parse the JSON container to make sense of the information. The information in the JSON container includes the interface name, the IPv4 and IPv6 addresses, and the hardware interface address. The information returned should be considered dynamic. An interface can be removed or added at any time, so this command should be used to discover added or removed network interfaces when you receive an error on an interface.

Summary

This chapter introduced the QEMU Guest Agent and how to use it to both obtain and set information on the libvirt client. The Guest Agent normally uses a JSON container to receive and return information. You will need to parse the returned information to make use of it. Finally, note that bypassing the libvirt interface and going to the QEMU Guest Agent directly can cause “out of sync” problems between libvirt and the QEMU Guest Agent as far as their configuration data is concerned. The Guest Agent should be used only as a last resort when libvirt cannot solve the problem.

CHAPTER 11

Debugging/Logging

This chapter covers the debugging and logging APIs in libvirt. This should supply you with enough information to solve runtime problems that sometimes occur with libvirt. Although the example programs in this book do not show how to incorporate debugging and logging in your program, there is enough information presented here to make that task easy enough.

Logging Facilities

libvirt includes logging facilities to facilitate the tracing of library execution. These logs will frequently be requested when trying to obtain support for libvirt, so familiarity with them is essential.

The logging facilities in libvirt are based on three key concepts.

- **Log messages**: Generated at runtime by the libvirt code, they include a timestamp, a priority level (`DEBUG` = 1, `INFO` = 2, `WARNING` = 3, `ERROR` = 4), a category, a function name, a line number indicating where the message originated, and a formatted message.
- **Log filters**: These are patterns and priorities that control whether a particular message is displayed. The format for a filter is as follows:

  ```
  x:name
  ```

https://doi.org/10.1007/978-1-4842-4862-1_11

where x is the minimal priority level where the match should apply, and name is a string to match against. The priority levels are as follows:

- 1 (or DEBUG): Log all messages
- 2 (or INFO): Log all nondebugging information
- 3 (or WARNING): Log only warnings and errors; this is the default
- 4 (or ERROR): Log only errors

For instance, to log all debug messages to the qemu driver, the following filter can be used:

```
1:qemu
```

Multiple filters can be specified together with a space separating them; the following example logs all debug messages from qemu and logs all error messages from the remote driver:

```
1:qemu 4:remote
```

- **Log outputs**: The output specifies where to send the message once it has passed through filters. The format for the log output is one of the following:

```
x:stderr - log to stderr
x:syslog:name - log to syslog with a prefix of "name"
x:file:file_path - log to a file specified by "file_path"
```

where x is the minimal priority level. For instance, to log all warnings and errors to syslog with a prefix of libvirtd, the following output can be used:

```
3:syslog:libvirtd
```

Multiple outputs can be specified with a space separating them; the following example logs all error and warning messages to syslog and logs all debug, information, warning, and error messages to `/tmp/libvirt.log`:

```
3:syslog:libvirtd 1:file:/tmp/libvirt.log
```

Environment Variables

The desired log priority level, filters, and outputs are specified to the libvirt library through the use of environment variables.

- `LIBVIRT_DEBUG` specifies the minimum priority level of messages that will be logged. This can be thought of as a “global” priority level; if a particular log message does not match a specific filter in `LIBVIRT_LOG_FILTERS`, it will be compared to this global priority and logged as appropriate.
- `LIBVIRT_LOG_FILTERS` specifies the filters to apply.
- `LIBVIRT_LOG_OUTPUTS` specifies the outputs to send the message to.

To see more detailed information about `virsh` errors, you can run `virsh` after setting the following environment variables:

Listing 11-1. Running virsh with Environment Variables

```
LIBVIRT_DEBUG=error
LIBVIRT_LOG_FILTERS="1:remote"
virsh list
```

This example will only print error messages from `virsh`, *except* that the remote driver will print all debug, information, warning, and error messages.

Summary

This chapter covered the debugging and logging facilities in libvirt. The examples should give you enough information to be able to use them in your own programs.

CHAPTER 12

A Sample Problem

This chapter introduces a sample problem. This problem can be solved through the use of multiple guest domains (virtual machines) and some automation programs. You will see some alternatives to solving a problem using automation. This will help prepare you to solve similar problems in your own environment.

Problem Statement

The sample problem is simple. The user has a software project that needs to be built, installed, tested, and uninstalled on multiple operating systems. There are two options for solving this problem.

- Install a different operating system on multiple machines and schedule the needed tasks on each of them. This option is both costly and hard to maintain as any change in build/test requirements will need to be reflected on each machine.
- Install multiple guest domains on one or more hosts running a virtual machine environment. This option is much cheaper in both hardware and administrative costs.

The second one is the better solution and the one that will be used to solve this problem in this chapter.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_12

Solution Requirements

To implement the solution, a more extensive set of requirements is needed, as follows (in no particular order):

- The software project must be built, installed, tested, and removed from a reference operating system environment. Thus, after the software has been installed and then uninstalled, the operating system must be returned to its reference state.
- The operating systems only needs to be in a running state during the build, installation, test, and uninstall tasks. It is reasonable that the operating system might not be in a running state if none of the needed tasks is running.
- The software will be built for both RPM- and DEB-based operating systems. But the output DEB or RPM install file will be specific for only a single operating system release. Install files will not be shared among operating systems.
- SSH will be used to communicate with each guest domain. All of the tasks (build, install, test, and uninstall) will be performed via scripts. Each script will be responsible for saving the outputs from that script (output files, logs, etc.) to an external machine.
- The host machine will be responsible for managing each guest domain and running the tasks listed earlier on a scheduled basis. The host will also need to maintain any information it needs to perform these tasks.

Even a first reading of these requirements raises a number of questions concerning just how each requirement should be solved. There are also many obvious questions about how to implement the solution for each requirement. All these will be explored subsequently in this chapter.

Using Guest Domains to Solve the Problem

Guest domains are able to solve a whole array of problems. The sample problem presented is a typical problem that can be solved by implementing one or more guest domains.

Using guest domains in an automated fashion is both easy and hard. While there are tools such as the libvirt library to help automate the required tasks, this also introduces additional requirements that the tools will need to track to successfully manage multiple guest domains. Thus, automation makes it easy to manage guest domains but at the cost of introducing additional overhead.

To successfully automate guest domains, there are a number of available software tools that interface to the libvirt library such as C, Python, Java, virsh, and others. This chapter will concentrate on examples using Python and sometimes virsh.

Introducing virsh

virsh is a command shell that interfaces to the libvirt library. It provides many commands that can create, destroy, and manipulate guest domains. It can be run as a shell, or the virsh commands can be invoked from a parent shell or process (such as the Python `os.system` routine).

The virsh commands are fully documented and available at `http://libvirt.org/virshcmdref.html`.

Introducing the Python libvirt Module

The Python libvirt module provides a set of classes and routines for manipulating guest domains. The set of methods and routines is extensive and covers a large percentage of the libvirt library.

The Python libvirt module is fully documented and available at `http://libvirt.org/docs/libvirt-appdev-guide-python/en-US/html/`.

New Problems Introduced by Using Guest Domains

By implementing guest domains as a solution to the sample problem, there are some new issues to deal with. The following are a few of these issues:

- The guest domains must be started and then stopped to perform each task. If tasks are successive in nature, then perform multiple tasks before shutting down the guest domain.
- To communicate with the guest domain via SSH, the domain name or IP address must be saved (or discovered), and the guest domain account that should be used to perform a task must be saved.
- The guest domains names to be used to perform the tasks must be saved along with any task-specific arguments or information.
- A storage mechanism must be chosen to save all the configuration information; flat files or a database are the logical choices.
- A task scheduling mechanism must be chosen.

- Other information must also be saved, or a discovery mechanism must be put into place to obtain the information. The IP address of the guest domain is the main piece of information that falls into this category. But there may be other information that is needed for a different kind of problem solution.
- A language must be chosen to program the needed steps to accomplish each task. The choice made here will have a lasting influence on how easy it is to maintain and enhance the problem solution over time. Choosing something like the C or Java language is a bad idea because it will be time-consuming and hard to implement even small changes. Using a high-level interpreted language will be much easier to implement the first time and maintain/enhance over time.

An interpreted language is also a good choice because the nature of the sample problem prevents the implemented language from reducing the time used for each task. Using C instead of something like Python will *not* reduce runtimes for the tasks. The overhead for managing the guest domains is low, and thus an interpreted language is the correct choice for implementing the solution.

Sample Solution

The remedy is to run jobs on QEMU-KVM domains using automation. To accomplish this, these issues need to be tackled:

- The list of domains and the command script needs to be stored in an easily manipulated container. This store also may need to contain other meta information concerning the domain, communications parameters, and job information.

- You need the code to read the store of information and then process the list consecutively.
- You need to create code that can restore or start a domain. You also need code that does the reverse: stores or shuts down a domain.
- To execute a command on the running domain, you need to decide on a communications method and then implement that method in code.

There are some other details you will need to make decisions about, but they will be presented in the following sections as the code is examined.

Implementing the Job Information Store

The amount of information you need to store is small, so use a comma-separated file (CSV) for the store. This also has the advantage of being easily readable from Python. Listing 12-1 shows an example of how this file is implemented in the solution.

Listing 12-1. CSV Store

```
"Fedora23-x86_64-1", "username", "command-to-be run"
"RedHat72-x86_64-1", "username", "command-to-be run"
```

The layout is simple. Each line contains a domain name and job to be run on the domain. It also contains a username. This is all the information needed by the code to iterate through the list of domains and execute the command on each domain.

The `username` is needed so that you can determine the user environment to be used to run the command. More on this field will be covered later in this chapter.

Reading the Store

The solution has been broken into two parts for reasons that will become obvious. Listing 12-2 provides the first part.

Listing 12-2. Using runJobs.py

```
#!/usr/bin/python -u

from __future__ import print_function
import argparse
import csv
import runJobOnDomain

# setup
"""Run the set of jobs defined in the CSV input file.
"""
__version__ = '1.0'
debug = False

parser = argparse.ArgumentParser( \
           description='Proess runJobs command line arguments.', \
           version=__version__)
parser.add_argument('--ifile', help='the csv input file.')
args = parser.parse_args()
# get the arguments
ifile = args.ifile
if debug == True:
    print('The command line arguments were:')
    print('ifile =', ifile)
with open(ifile, 'rb') as csvfile:
    reader = csv.reader(csvfile, delimiter=',', quotechar='"')
    for row in reader:
        domain = row[0]
        userid = row[1]
        cmd = row[2]
        runJobOnDomain.runJob(domain, userid, cmd)
exit(0)
```

This Python script was created to run on Python 2.7. If run on a newer version, you will need to rework lines 14 to 18. The argument parser was reworked in Python 3.*x*, and this version of the code will fail on newer versions of Python.

The code is simple. The input file is obtained from a command-line parameter. The input file is then read using a Python `csv.reader` one line at a time. When all the information on a line of the input file has been gathered, the script then calls the `runJobOnDomain.runJob` method to actually perform the job specified by the entry in the CSV file.

Using runJobOnDemand.py

The script presented here can be run as a stand-alone program or called from the `runJobs.py` script. When run as a stand-alone program, it obtains the parameters it needs from the command line. This allows the script to run a single command on a domain without having to create a CSV file for a single entry. It should now be obvious why the processing was split into two steps.

The solution will utilize SSH to run the command on the domain (Listing 12-3). While it is easy to implement, it does introduce a new requirement. The host user account must have generated a set of public/private keys to be used for the SSH connection. The public key on the host must be stored on the domain. All this is to prevent a password prompt when accessing a domain. A password prompt will prevent the command from running when the scripts are used in an automated environment.

Listing 12-3. Using runJobOnDemand.py

```
#!/usr/bin/python -u

from __future__ import print_function
import argparse
import os
import time
import libvirt

# setup
"""A function for running an SSH command on an active
   or inactive domain.
"""
__version__ = '1.0'
debug = False
uri = 'qemu:///system'
startupWait = 60  # in seconds

def toIPAddrType(addrType):
    if addrType == libvirt.VIR_IP_ADDR_TYPE_IPV4:
        return "ipv4"
    elif addrType == libvirt.VIR_IP_ADDR_TYPE_IPV6:
        return "ipv6"

def runJob(domain_name, userid, cmd, restoreFile='', open_connx=None):
    # get the connection
    if open_connx == None:
        connx = libvirt.open(uri)
    else:
        connx = open_connx
    # get the domain
    domain = connx.lookupByName(domain_name)
    if domain == None:
        if open_connx == None:
            connx.close()
        return
    # start/restore the domain if necessary
    wasActive = domain.isActive()
    if debug == True:
        print('wasActive = '+str(wasActive))
    if restoreFile != '' and wasActive == False:
        if debug == True:
            print('Restoring domain')
        domain.restore(restoreFile)
    elif restoreFile != '' and wasActive == True:
        if debug == True:
            print('Domain was active')
        pass
    elif wasActive == False:
        if debug == True:
            print('Starting domain -- sleeping')
        if domain.create() < 0:
            print('Error: Domain is not active or never started.')
            return(-1)
        time.sleep(startupWait)
    else:
        if debug == True:
            print('Domain was active')
        pass
    # make sure the domain is running
    if domain.isActive() == False:
        print('Error: Domain is not active or never started.')
        return(-1)
    # get the ip address of the domain
```

```
ifaces = domain.interfaceAddresses( \
         libvirt.VIR_DOMAIN_INTERFACE_ADDRESSES_SRC_AGENT, 0)
usableIpaddr = None
for (name, val) in ifaces.iteritems():
    if val['addrs']:
        for ipaddr in val['addrs']:
            if debug == True:
                print(str(ipaddr))
            if ipaddr['addr'][0:8] == '192.168.' and \
             toIPAddrType(ipaddr['type']) == 'ipv4':
                usableIpaddr = ipaddr['addr']
                break
            if ipaddr['addr'][0:3] == '10.' and \
                toIPAddrType(ipaddr['type']) == 'ipv4':
                usableIpaddr = ipaddr['addr']
                break
            # currently we ignore ipv6 addresses
# run the SSH command
if usableIpaddr == None:
    print('Error: No usable ipaddress was found.')
    return(-1)
sshcmd = 'ssh %s@%s %s' % (userid, usableIpaddr, cmd)
os.system(sshcmd)
# shutdown the domain if necessary
if wasActive == False:
    if restoreFile != '':
        domain.save(restoreFile)
    else:
        domain.shutdown()
# close the connection if necessary
if open_connx == None:
        connx.close()
    return(0)

if __name__ == "__main__":
    parser = argparse.ArgumentParser( \
        description='Proess runJobOnDomain command line
        arguments.', \
        version=__version__)
    parser.add_argument('--domain', \
                        help='the domain to be used to run
                        the job.')
    parser.add_argument('--userid', \
                        help='the SSH userid for the job.')
    parser.add_argument('--cmd', \
                      help='the SSH command to be run on
                      the domain.')
    args = parser.parse_args()
    # get the arguments
    domain = args.domain
    userid = args.userid
    cmd = args.cmd
    if debug == True:
        print('The command line arguments were:')
        print('domain = '+domain)
        print('userid = '+userid)
        print('cmd = '+cmd)
    retc = runJob(domain, userid, cmd, '')
    exit(0)
```

Let’s start our examination at line 98 because this is the point the action gets started when the script is run as a stand-alone program.

The same warnings apply for lines 100 to 108 in this script as for the `runJobs.py` script. The argument processing will need to be reworked for newer versions of Python.

After obtaining the command-line parameters, a call to `runJob` is made. This is the same call that is made from the `runJobs.py` script. Thus, the same code will be executed to perform the job command in either situation.

The `runJob` function at line 23 is the main function in this script and is where most of the work is performed.

Lines 24 to 28 get a connection to the libvirt environment, and lines 29 to 34 use the connection to get the domain. If the connection cannot be made or the domain cannot be found, the function returns to the caller.

Lines 35 to 61 contain some assumptions about how to start/restore a domain under a number of conditions. If the domain is already running, a flag is set so when the process is done using the domain, it will not try to save/shut down the domain. The process also investigates the `restoreFile` variable to see whether it should restore the domain instead of starting it. Currently, the `restoreFile` variable is not used in these scripts, but the processing options are implemented if you want to enhance the scripts to use it.

Lines 63 to 79 implement the new libvirt `interfaceAddresses` method call to the domain to obtain the list of network interfaces. One of these interfaces will be used to make an SSH call to implement the job command. The code as implemented looks for the first IPv4 address that starts with 192.168. or 10. and then uses that interface. If this is not correct for your environment, then you will need to modify this code.

Lines 81 to 85 call SSH to send the command to the domain where it will be processed. Note that the interface used is the one picked using the previous code section. The user ID for the SSH command is obtained from the function arguments.

Lastly, lines 87 to 94 shut down/save the domain if needed. Next, the connection is closed if needed.

The previous examples show how to access a guest domain to perform functions from a Python program running on the main host. I hope that the examples contain enough information so that you can modify, extend, or use it in other ways to run programs on the guest domain.

Summary

This chapter showed you how to write a Python program that uses SSH to access a guest domain and run a specific program in that domain. The main program will run on the host, and a separate program will run on the guest domain and possibly return information to the main host program. I hope this program has supplied you with enough ideas and examples to help you in designing your own programs to perform the services and functions needed in your environment.

CHAPTER 13

Storing Information About Virtual Machines

There are two categories of information that need to be obtained and stored to be able to manipulate and use guest domains.

- The information needed by libvirt to access and manipulate a guest domain
- The information necessary to access the guest domain via SSH and run the programs necessary to accomplish the required tasks

This chapter discusses the requirements necessary for the information to be stored as well as some options for storing that information.

Deciding What Information Should Be Stored

Here is a list of some basic information that should be stored:

- The guest domain name (for libvirt access)
- The IP address or DNS name of the guest domain (for SSH access)

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_13

- The username for the guest domain (for SSH access)
- The command (or commands) to be run in the guest domain to perform tasks (for SSH)

There are many other items that an administrator might want to store such as database names and access information, command arguments (these might be dynamic), time/date information for scheduling, and many more.

Scheduling when tasks are to be performed is a major design decision. This also might include grouping guest domains to run a task on each of them via either a script or separate cron jobs. There will be more discussion on this topic in Chapter 12.

Another consideration is whether guest domains may run multiple tasks at the same time or only consecutively. This has major consequences because if multiple tasks can access a guest domain simultaneously, then mechanisms must be set up so that one task cannot shut down a domain while another task is still running. This usually means setting up a global counter so that any task can determine whether it is safe to shut down a guest domain.

Using Simple Text Files

From a Python perspective, probably the easiest text file format to store information in is a comma-separated value (CSV) file. Listing 13-1 shows an example of a CSV file storing a minimal amount of information.

Listing 13-1. An Example CSV Project File

```
RHEL7.2-x86_64-1,192.186.1.166,buildid,/home/buildid/buildmyproject
CentOS6.6-i386-1,192.168.1.172,buildid,/home/buildid/buildmyproject
Ubuntu15.10-x86_64-1,192.168.1.70,buildid,/home/buildid/buildmyproject
```

This file includes fields for the guest domain name, IP address, user ID, and command to be performed. This should be enough information to activate the guest domain, run the SSH command to perform the task, and then shut down the domain.

Next, you need a Python class to store the information from the CSV file, as shown in Listing 13-2. Be sure to save the example as `Example_5.py` so that it will not cause a syntax problem with the Python interpreter.

Listing 13-2. An Example Python Class to Store Guest Domain Information

```
# Example-5.py

class guest_domain ():
    """This class stores information about a single libvirt
    guest domain."""
    objectname = 'guest_domain'
    domain_name = ''
    ipaddress = ''
    userid = ''
    cmd = ''
    def __init__ (self, domain_name, ipaddress, userid, cmd):
        self.domain_name = domain_name
        self.ipaddress = ipaddress
        self.userid = userid
        self.cmd = cmd
        return
```

Now that you can store the guest domain information, you need a way to read the CSV file and store that information in the Python class you have defined. Listing 13-3 shows how to perform that function. Be sure to save the example as `Example_6.py` so that it will not cause a syntax problem with the Python interpreter.

Listing 13-3. An Example Python Class to Read the CSV File and Store Guest Domain Information

```
# Example-6.py

import csv
import Example_5

def get_domains(filename):
    guest_domains = list()
    with open(filename) as csvfile:
        fieldnames = ['domain_name', 'ipaddress', 'userid', 'cmd']
        reader = csv.DictReader(csvfile, fieldnames)
        for row in reader:
            domain_name = row['domain_name']
            ipaddress = row['ipaddress']
            userid = row['userid']
            cmd = row['cmd']
            gd = Example_5.guest_domain(domain_name, ipaddress, \
                                            userid, cmd)
            guest_domains.append(gd)
    return guest_domains
```

Now that you have the basic structures in place to store and read the guest domain information, you need to test these structures to make sure they provide valid information (Listing 13-4).

Listing 13-4. Test the CSV and Guest Domain Structures

```
# Example-7.py

import csv
import Example_6

guest_domains = Example_6.get_domains('Example-myprojectbuild.csv')
```

```
for gd in guest_domains:
    print(gd.domain_name + ',' + gd.ipaddress + ',' + gd.userid
    + ',' \
    + gd.cmd)
```

This example should essentially print out the same thing as the contents of the CSV file.

You now have the basic mechanisms in place for storing and reading the guest domain and other information that will be needed to manipulate the guest domains and perform tasks in each guest domain.

Using a Simple Database

For more robust situations, a database will be a more appropriate solution for storing the required data. Also, a database allows a much wider choice of field types and the use of additional fields for other data. This will be especially useful for cloud implementations.

To accommodate multiple jobs that might be run on a single VM, the guest domain data and the job data are divided into two separate tables. The minimal table definitions will look something like Listing 13-5.

Listing 13-5. Minimal Table Definitions

```
-- Example-8.sql

CREATE TABLE IF NOT EXISTS domains (
    domain_name varchar(80),
    ipaddress varchar(15) DEFAULT NULL,
    PRIMARY KEY(domain_name)
    );

CREATE TABLE IF NOT EXISTS jobs (
    domain_name varchar(80),
    job_name varchar(80),
    userid varchar(20),
    cmd varchar(256),
    PRIMARY KEY(domain_name)
    );
```

A new field named `job_name` has been added so multiple jobs for a single guest domain can be distinguished from each other. It now becomes obvious that there are many other fields that could be useful as well.

- To start a job at a particular time, some scheduling and frequency fields could be added.
- For cloud implementations, some fields may need to be added to the domains table for ownership and accounting purposes.
- To keep track of resources, it might be necessary to add additional tables to hold that information.

Assuming the tables described earlier are defined in a MariaDB database called `mybuilddb`, you can extract all the jobs for a guest domain by using the Python code in Listing 13-6.

Listing 13-6. Fetching Jobs from the Database

```
# Example-9.py

import mysql.connector as mariadb

mariadb_connection = mariadb.connect(user='python_user', \
                                     password='some_pass', \
                                     database='mybuilddb')
cursor = mariadb_connection.cursor()

cursor.execute("SELECT domain_name,userid,cmd FROM domains,
jobs " + \
 "WHERE domains.domain_name=%s " + \
 "AND domains.domain_name=jobs.domain_name", \
 ("RHEL7.2-x86_64-1",))

for job_name, userid, cmd in cursor:
    print("Job name: {}, Userid: {}, Cmd: {}").format(job_name, \
```

Summary

This chapter presented a single example that uses many of the concepts introduced in previous chapters. The goal has been to present it in such a way that it can be easily modified to apply to your situation. In this case, you can easily modify the example to use another database or other storage mechanism. Also, the concepts presented here should give you some idea of how other programs can be created to solve use cases in your environment.

CHAPTER 14

Securing Virtual Machines

In this chapter, you'll quickly review host and guest domain security.

As you know, a *host* is a computer connected to one or more guest domains. The number of guest domains is limited only by the capacity of the server. The domains may be on a private virtual network or a shared network via a shared network adapter on the host.

While guest domain security will be similar to the host's, there are some additional items that apply. In most cases, the OS should be hardened just like the host. However, there may be both exceptions and additional hardening necessary. For instance, if VNC is used to access the guest domain, then you will need to add some additional security and open the firewall for the VNC connection. Other security items may also apply, so make sure you plan accordingly.

Host Security

The security of the host should be treated just like any other server on the network. However, the libvirtd daemon is a new item that will need to be mixed in with all the other security items. Here are some items to consider when planning for a host that will have guest domains installed:

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1_14

- Access to accounts on the host that have libvirt group privileges should be highly restricted because *all* guest domains will be accessible to any user with libvirt group privileges.
- VNC access should be turned off to the host. It allows too many possibilities for users to change configurations.
- Allow external access to the libvirtd port only if necessary. It is needed only if remote libvirt administration is needed.
- When planning remote access to the server, try to use TLS certificates if at all possible (se Chapter 3). They are easy to work with and provide fine-grained access privileges.
- Ensure that job information and job scripts are secured with the correct access permissions and that only specific users may run or modify the files and databases.

At this point, I would normally show a few examples of both good and bad security configurations. But one of the things I've learned by reading security audits over the years is that it is really hard to say that this technique is good and that technique is bad. The environment usually determines what is good or bad. In other words, a good technique in one environment may be irrelevant in another. A good consultant will tailor their recommendations to the environment.

Guest Domain Security

Guest domain security is tricky. Much is dependent on whether the domain is using a shared network adapter and the kind of network the adapter it is attached to. The more public the network of the guest domain, the more secure the setup for the domain will need to be.

Here are some items to consider when planning for guest domains:

- Port forwarding from the host to a guest domain can be tricky, and it can expose the guest domain to potential user escalation problems. Try to avoid using port forwarding if possible.
- If the guest domain is using a real shared network adapter, it will be exposed to all the same threats as that network. Plan the configuration of the guest domain firewall and exposed applications accordingly.
- Allow access to the libvirtd port only if necessary (and in most cases it is never necessary to allow port access on a guest domain).
- Be careful of storing guest domain disk images on shared storage. It makes them vulnerable to modification if the access privileges are incorrect. It can also make disk access on the guest domain much slower.

Summary

Hosts are computers that are connected to one or more guest domains on a private virtual network or a shared network. Guest domain security is similar to a host's, but there may be exceptions and additional hardening necessary.

APPENDIX A

Libvirt XML Schemas

This appendix covers the XML schemas used by libvirt. Each major section in the appendix describes a single libvirt domain. Everything needed to completely describe all the elements of a libvirt domain are contained in the schema.

The Domain Schema

The Domain schema completely describes a libvirt domain, i.e., a virtual machine. The domain can be any of the supported libvirt types including QEMU/KVM, XEN, and so on. Obviously, some of the schema sections apply to only some of the supported domain types, so it makes sense that you need to include only the parts of the schema that apply to your domain. The same applies to devices and features. If your domain does not support or have certain devices or features, then simply do not include those sections.

The Domain XML schema consists of a large number of XML elements (Listing A-1). The documentation for this schema is quite extensive, but in most cases the uses of the elements in the schema are obvious. If you need documentation, refer to `https://libvirt.org/formatdomain.html` for more specific information about each element and attribute.

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1

Listing A-1. The Domain Schema

```
<!--General metadata -->

<domain type='kvm' id='1'>
  <name>MyGuest</name>
  <uuid>4dea22b3-1d52-d8f3-2516-782e98ab3fa0</uuid>
  <genid>43dc0cf8-809b-4adb-9bea-a9abb5f3d90e</genid>
  <title>A short description - title - of the domain</title>
  <description>Some human readable description</description>
  <metadata>
    <app1:foo xmlns:app1="http://app1.org/app1/">..</app1:foo>
    <app2:bar xmlns:app2="http://app1.org/app2/">..</app2:bar>
  </metadata>
  <bootloader>/usr/bin/pygrub</bootloader>
  <bootloader_args>--append single</bootloader_args>

  <!-- BIOS bootloader -->

  <os>
    <type>hvm</type>
    <loader readonly='yes' secure='no' type='rom'>
        /usr/lib/xen/boot/hvmloader</loader>
    <nvram template='/usr/share/OVMF/OVMF_VARS.fd'>
        /var/lib/libvirt/nvram/guest_VARS.fd</nvram>
    <boot dev='hd'/>
    <boot dev='cdrom'/>
    <bootmenu enable='yes' timeout='3000'/>
    <smbios mode='sysinfo'/>
    <bios useserial='yes' rebootTimeout='0'/>
  </os>
```

```
<!-- Host bootloader -->

<bootloader>/usr/bin/pygrub</bootloader>
<bootloader_args>--append single</bootloader_args>

<!-- Direct kernel boot -->

<os>
  <type>hvm</type>
  <loader>/usr/lib/xen/boot/hvmloader</loader>
  <kernel>/root/f8-i386-vmlinuz</kernel>
  <initrd>/root/f8-i386-initrd</initrd>
  <cmdline>console=ttyS0 ks=http://example.com/f8-i386/os/
  </cmdline>
  <dtb>/root/ppc.dtb</dtb>
  <acpi>
    <table type='slic'>/path/to/slic.dat</table>
  </acpi>
</os>

<!-- Container boot -->

<os>
  <type arch='x86_64'>exe</type>
  <init>/bin/systemd</init>
  <initarg>--unit</initarg>
  <initarg>emergency.service</initarg>
  <initenv name='MYENV'>some value</initenv>
  <initdir>/my/custom/cwd</initdir>
  <inituser>tester</inituser>
  <initgroup>1000</initgroup>
</os>
```

```
  <idmap>
    <uid start='0' target='1000' count='10'/>
    <gid start='0' target='1000' count='10'/>
  </idmap>

  <!-- SMBIOS System Information -->

  <os>
  <smbios mode='sysinfo'/>
  </os>
  <sysinfo type='smbios'>
    <bios>
      <entry name='vendor'>LENOVO</entry>
    </bios>
    <system>
      <entry name='manufacturer'>Fedora</entry>
      <entry name='product'>Virt-Manager</entry>
      <entry name='version'>0.9.4</entry>
    </system>
    <baseBoard>
      <entry name='manufacturer'>LENOVO</entry>
      <entry name='product'>20BE0061MC</entry>
      <entry name='version'>0B98401 Pro</entry>
      <entry name='serial'>W1KS427111E</entry>
    </baseBoard>
    <chassis>
      <entry name='manufacturer'>Dell Inc.</entry>
      <entry name='version'>2.12</entry>
      <entry name='serial'>65X0XF2</entry>
      <entry name='asset'>40000101</entry>
      <entry name='sku'>Type3Sku1</entry>
    </chassis>
    <oemStrings>
      <entry>myappname:some arbitrary data</entry>
      <entry>otherappname:more arbitrary data</entry>
    </oemStrings>
  </sysinfo>

  <!-- CPU Allocation -->

  <vcpu placement='static' cpuset="1-4,^3,6" current="1">2</vcpu>
  <vcpus>
    <vcpu id='0' enabled='yes' hotpluggable='no' order='1'/>
    <vcpu id='1' enabled='no' hotpluggable='yes'/>
  </vcpus>

  <!-- IOThreads Allocation -->

  <iothreads>4</iothreads>
  <iothreadids>
    <iothread id="2"/>
    <iothread id="4"/>
    <iothread id="6"/>
    <iothread id="8"/>
  </iothreadids>

 <!-- CPU Tuning -->

  <cputune>
    <vcpupin vcpu="0" cpuset="1-4,^2"/>
    <vcpupin vcpu="1" cpuset="0,1"/>
    <vcpupin vcpu="2" cpuset="2,3"/>
    <vcpupin vcpu="3" cpuset="0,4"/>
    <emulatorpin cpuset="1-3"/>
    <iothreadpin iothread="1" cpuset="5,6"/>
    <iothreadpin iothread="2" cpuset="7,8"/>
    <shares>2048</shares>
    <period>1000000</period>
    <quota>-1</quota>
    <global_period>1000000</global_period>
    <global_quota>-1</global_quota>
    <emulator_period>1000000</emulator_period>
    <emulator_quota>-1</emulator_quota>
    <iothread_period>1000000</iothread_period>
    <iothread_quota>-1</iothread_quota>
    <vcpusched vcpus='0-4,^3' scheduler='fifo' priority='1'/>
    <iothreadsched iothreads='2' scheduler='batch'/>
    <cachetune vcpus='0-3'>
      <cache id='0' level='3' type='both' size='3' unit='MiB'/>
      <cache id='1' level='3' type='both' size='3' unit='MiB'/>
      <monitor level='3' vcpus='1'/>
      <monitor level='3' vcpus='0-3'/>
    </cachetune>
    <cachetune vcpus='4-5'>
      <monitor level='3' vcpus='4'/>
      <monitor level='3' vcpus='5'/>
    </cachetune>
    <memorytune vcpus='0-3'>
      <node id='0' bandwidth='60'/>
    </memorytune>
  </cputune>

  <!-- Memory Allocation -->

  <maxMemory slots='16' unit='KiB'>1524288</maxMemory>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
```

```
<!-- Memory Backing -->

<memoryBacking>
  <hugepages>
    <page size="1" unit="G" nodeset="0-3,5"/>
    <page size="2" unit="M" nodeset="4"/>
  </hugepages>
  <nosharepages/>
  <source type="file|anonymous|memfd"/>
  <access mode="shared|private"/>
  <allocation mode="immediate|ondemand"/>
</memoryBacking>

<!-- Memory Tuning -->

<memtune>
  <hard_limit unit='G'>1</hard_limit>
  <soft_limit unit='M'>128</soft_limit>
  <swap_hard_limit unit='G'>2</swap_hard_limit>
  <min_guarantee unit='bytes'>67108864</min_guarantee>
</memtune>

<!-- NUMA Node Tuning -->

<numatune>
  <memory mode="strict" nodeset="1-4,^3"/>
  <memnode cellid="0" mode="strict" nodeset="1"/>
  <memnode cellid="2" mode="preferred" nodeset="2"/>
</numatune>

<!-- Block I/O Tuning -->

<blkiotune>
  <weight>800</weight>
  <device>
```

```
      <path>/dev/sda</path>
      <weight>1000</weight>
    </device>
    <device>
      <path>/dev/sdb</path>
      <weight>500</weight>
      <read_bytes_sec>10000</read_bytes_sec>
      <write_bytes_sec>10000</write_bytes_sec>
      <read_iops_sec>20000</read_iops_sec>
      <write_iops_sec>20000</write_iops_sec>
    </device>
  </blkiotune>

  <!-- Resource partitioning -->

  <resource>
    <partition>/virtualmachines/production</partition>
  </resource>

  <!-- CPU model and topology -->

  <cpu match='exact'>
    <model fallback='allow'>core2duo</model>
    <vendor>Intel</vendor>
    <topology sockets='1' cores='2' threads='1'/>
    <cache level='3' mode='emulate'/>
    <feature policy='disable' name='lahf_lm'/>
  </cpu>
  <cpu mode='host-model'>
    <model fallback='forbid'/>
    <topology sockets='1' cores='2' threads='1'/>
  </cpu>
```

```
<cpu mode='host-passthrough'>
  <cache mode='passthrough'/>
  <feature policy='disable' name='lahf_lm'/>
</cpu>
<cpu>
  <topology sockets='1' cores='2' threads='1'/>
</cpu>
<cpu>
  <numa>
    <cell id='0' cpus='0-3' memory='512000' unit='KiB'
    discard='yes'/>
    <cell id='1' cpus='4-7' memory='512000' unit='KiB'
    memAccess='shared'/>
  </numa>
  <numa>
    <cell id='0' cpus='0,4-7' memory='512000' unit='KiB'>
      <distances>
        <sibling id='0' value='10'/>
        <sibling id='1' value='21'/>
        <sibling id='2' value='31'/>
        <sibling id='3' value='41'/>
      </distances>
    </cell>
    <cell id='1' cpus='1,8-10,12-15' memory='512000' unit='KiB'
        memAccess='shared'>
      <distances>
        <sibling id='0' value='21'/>
        <sibling id='1' value='10'/>
        <sibling id='2' value='21'/>
        <sibling id='3' value='31'/>
      </distances>
    </cell>
```

```
      <cell id='2' cpus='2,11' memory='512000' unit='KiB'
      memAccess='shared'>
        <distances>
          <sibling id='0' value='31'/>
          <sibling id='1' value='21'/>
          <sibling id='2' value='10'/>
          <sibling id='3' value='21'/>
        </distances>
      </cell>
      <cell id='3' cpus='3' memory='512000' unit='KiB'>
        <distances>
          <sibling id='0' value='41'/>
          <sibling id='1' value='31'/>
          <sibling id='2' value='21'/>
          <sibling id='3' value='10'/>
        </distances>
      </cell>
    </numa>
  </cpu>

  <!-- Events configuration -->

  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <on_lockfailure>poweroff</on_lockfailure>

  <!-- Power Management -->

  <pm>
    <suspend-to-disk enabled='no'/>
    <suspend-to-mem enabled='yes'/>
  </pm>
```

```
<!-- Hypervisor features -->

<features>
  <pae/>
  <acpi/>
  <apic/>
  <hap/>
  <privnet/>
  <hyperv>
    <relaxed state='on'/>
    <vapic state='on'/>
    <spinlocks state='on' retries='4096'/>
    <vpindex state='on'/>
    <runtime state='on'/>
    <synic state='on'/>
    <reset state='on'/>
    <vendor_id state='on' value='KVM Hv'/>
    <frequencies state='on'/>
    <reenlightenment state='on'/>
    <tlbflush state='on'/>
    <ipi state='on'/>
    <evmcs state='on'/>
  </hyperv>
  <kvm>
    <hidden state='on'/>
  </kvm>
  <pvspinlock state='on'/>
  <gic version='2'/>
  <ioapic driver='qemu'/>
  <hpt resizing='required'>
    <maxpagesize unit='MiB'>16</maxpagesize>
  </hpt>
```

```
    <vmcoreinfo state='on'/>
    <smm state='on'>
      <tseg unit='MiB'>48</tseg>
    </smm>
    <htm state='on'/>
  </features>

  <!-- Time keeping -->

  <clock offset='localtime'>
    <timer name='rtc' tickpolicy='catchup' track='guest'>
      <catchup threshold='123' slew='120' limit='10000'/>
    </timer>
    <timer name='pit' tickpolicy='delay'/>
  </clock>

  <!-- Performance monitoring events -->

  <perf>
    <event name='cmt' enabled='yes'/>
    <event name='mbmt' enabled='no'/>
    <event name='mbml' enabled='yes'/>
    <event name='cpu_cycles' enabled='no'/>
    <event name='instructions' enabled='yes'/>
    <event name='cache_references' enabled='no'/>
    <event name='cache_misses' enabled='no'/>
    <event name='branch_instructions' enabled='no'/>
    <event name='branch_misses' enabled='no'/>
    <event name='bus_cycles' enabled='no'/>
    <event name='stalled_cycles_frontend' enabled='no'/>
    <event name='stalled_cycles_backend' enabled='no'/>
    <event name='ref_cpu_cycles' enabled='no'/>
    <event name='cpu_clock' enabled='no'/>
    <event name='task_clock' enabled='no'/>
    <event name='page_faults' enabled='no'/>
    <event name='context_switches' enabled='no'/>
    <event name='cpu_migrations' enabled='no'/>
    <event name='page_faults_min' enabled='no'/>
    <event name='page_faults_maj' enabled='no'/>
    <event name='alignment_faults' enabled='no'/>
    <event name='emulation_faults' enabled='no'/>
  </perf>

  <!-- Devices -->

  <devices>
    <emulator>/usr/lib/xen/bin/qemu-dm</emulator>
  </devices>
  <devices>
    <disk type='file'>
      <alias name='ua-myDisk'/>
    </disk>
    <interface type='network' trustGuestRxFilters='yes'>
      <alias name='ua-myNIC'/>
    </interface>
    ...
  </devices>

  <!-- Hard drives, floppy disks, CDROMs -->

  <devices>
    <disk type='file' snapshot='external'>
      <driver name="tap" type="aio" cache="default"/>
      <source file='/var/lib/xen/images/fv0' startupPolicy=
      'optional'>
        <seclabel relabel='no'/>
      </source>
```

```
      <target dev='hda' bus='ide'/>
      <iotune>
        <total_bytes_sec>10000000</total_bytes_sec>
        <read_iops_sec>400000</read_iops_sec>
        <write_iops_sec>100000</write_iops_sec>
      </iotune>
      <boot order='2'/>
      <encryption type='...'>
        ...
      </encryption>
      <shareable/>
      <serial>
        ...
      </serial>
    </disk>
    <disk type='network'>
      <driver name="qemu" type="raw" io="threads" ioeventfd="on"
          event_idx="off"/>
      <source protocol="sheepdog" name="image_name">
        <host name="hostname" port="7000"/>
      </source>
      <target dev="hdb" bus="ide"/>
      <boot order='1'/>
      <transient/>
      <address type='drive' controller='0' bus='1' unit='0'/>
    </disk>
    <disk type='network'>
      <driver name="qemu" type="raw"/>
      <source protocol="rbd" name="image_name2">
        <host name="hostname" port="7000"/>
        <snapshot name="snapname"/>
        <config file="/path/to/file"/>
        <auth username='myuser'>
          <secret type='ceph' usage='mypassid'/>
        </auth>
      </source>
      <target dev="hdc" bus="ide"/>
    </disk>
    <disk type='block' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <target dev='hdd' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='network' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <source protocol="http" name="url_path">
        <host name="hostname" port="80"/>
      </source>
      <target dev='hde' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='network' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <source protocol="https" name="url_path">
        <host name="hostname" port="443"/>
      </source>
      <target dev='hdf' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='network' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <source protocol="ftp" name="url_path">
        <host name="hostname" port="21"/>
      </source>
```

```
      <target dev='hdg' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='network' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <source protocol="ftps" name="url_path">
        <host name="hostname" port="990"/>
      </source>
      <target dev='hdh' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='network' device='cdrom'>
      <driver name='qemu' type='raw'/>
      <source protocol="tftp" name="url_path">
        <host name="hostname" port="69"/>
      </source>
      <target dev='hdi' bus='ide' tray='open'/>
      <readonly/>
    </disk>
    <disk type='block' device='lun'>
      <driver name='qemu' type='raw'/>
      <source dev='/dev/sda'>
        <reservations managed='no'>
          <source type='unix' path='/path/to/qemu-pr-helper'
          mode='client'/>
        </reservations>
      <target dev='sda' bus='scsi'/>
      <address type='drive' controller='0' bus='0' target='3'
      unit='0'/>
    </disk>
    <disk type='block' device='disk'>
      <driver name='qemu' type='raw'/>
      <source dev='/dev/sda'/>
      <geometry cyls='16383' heads='16' secs='63' trans='lba'/>
      <blockio logical_block_size='512' physical_block_size='4096'/>
      <target dev='hdj' bus='ide'/>
    </disk>
    <disk type='volume' device='disk'>
      <driver name='qemu' type='raw'/>
      <source pool='blk-pool0' volume='blk-pool0-vol0'/>
      <target dev='hdk' bus='ide'/>
    </disk>
    <disk type='network' device='disk'>
      <driver name='qemu' type='raw'/>
      <source protocol='iscsi' name='iqn.2013-07.com.
      example:iscsi-nopool/2'>
        <host name='example.com' port='3260'/>
        <auth username='myuser'>
          <secret type='iscsi' usage='libvirtiscsi'/>
        </auth>
      </source>
      <target dev='vda' bus='virtio'/>
    </disk>
    <disk type='network' device='lun'>
      <driver name='qemu' type='raw'/>
      <source protocol='iscsi' name='iqn.2013-07.com.
      example:iscsi-nopool/1'>
        <host name='example.com' port='3260'/>
        <auth username='myuser'>
          <secret type='iscsi' usage='libvirtiscsi'/>
        </auth>
      </source>
      <target dev='sdb' bus='scsi'/>
    </disk>
```

```
    <disk type='network' device='lun'>
      <driver name='qemu' type='raw'/>
      <source protocol='iscsi' name='iqn.2013-07.com.
      example:iscsi-nopool/0'>
        <host name='example.com' port='3260'/>
        <initiator>
          <iqn name='iqn.2013-07.com.example:client'/>
        </initiator>
      </source>
      <target dev='sdb' bus='scsi'/>
    </disk>
    <disk type='volume' device='disk'>
      <driver name='qemu' type='raw'/>
      <source pool='iscsi-pool' volume='unit:0:0:1' mode='host'/>
      <target dev='vdb' bus='virtio'/>
    </disk>
    <disk type='volume' device='disk'>
      <driver name='qemu' type='raw'/>
      <source pool='iscsi-pool' volume='unit:0:0:2' mode='direct'/>
      <target dev='vdc' bus='virtio'/>
    </disk>
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2' queues='4'/>
      <source file='/var/lib/libvirt/images/domain.qcow'/>
      <backingStore type='file'>
        <format type='qcow2'/>
        <source file='/var/lib/libvirt/images/snapshot.qcow'/>
        <backingStore type='block'>
          <format type='raw'/>
          <source dev='/dev/mapper/base'/>
          <backingStore/>
        </backingStore>
      </backingStore>
      <target dev='vdd' bus='virtio'/>
    </disk>
  </devices>

  <!-- Filesystems -->

  <devices>
    <filesystem type='template'>
      <source name='my-vm-template'/>
      <target dir='/'/>
    </filesystem>
    <filesystem type='mount' accessmode='passthrough'>
      <driver type='path' wrpolicy='immediate'/>
      <source dir='/export/to/guest'/>
      <target dir='/import/from/host'/>
      <readonly/>
    </filesystem>
    <filesystem type='file' accessmode='passthrough'>
      <driver name='loop' type='raw'/>
      <driver type='path' wrpolicy='immediate'/>
      <source file='/export/to/guest.img'/>
      <target dir='/import/from/host'/>
      <readonly/>
    </filesystem>
  </devices>

  <!-- Controllers -->

  <devices>
    <controller type='ide' index='0'/>
    <controller type='virtio-serial' index='0' ports='16'
    vectors='4'/>
    <controller type='virtio-serial' index='1'>
```

```
      <address type='pci' domain='0x0000' bus='0x00'
      slot='0x0a'
          function='0x0'/>
    </controller>
    <controller type='scsi' index='0' model='virtio-scsi'>
      <driver iothread='4'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x0b'
          function='0x0'/>
    </controller>
  </devices>
  <devices>
    <controller type='usb' index='0' model='ich9-ehci1'>
      <address type='pci' domain='0' bus='0' slot='4'
      function='7'/>
    </controller>
    <controller type='usb' index='0' model='ich9-uhci1'>
      <master startport='0'/>
      <address type='pci' domain='0' bus='0' slot='4' function='0'
          multifunction='on'/>
    </controller>
  </devices>
  <devices>
    <controller type='pci' index='0' model='pci-root'/>
    <controller type='pci' index='1' model='pci-bridge'>
      <address type='pci' domain='0' bus='0' slot='5' function='0'
          multifunction='off'/>
    </controller>
  </devices>
  <devices>
    <controller type='pci' index='0' model='pcie-root'/>
    <controller type='pci' index='1' model='dmi-to-pci-bridge'>
      <address type='pci' domain='0' bus='0' slot='0xe'
      function='0'/>
    </controller>
    <controller type='pci' index='2' model='pci-bridge'>
      <address type='pci' domain='0' bus='1' slot='1'
      function='0'/>
    </controller>
  </devices>

  <!-- Device leases -->

  <devices>
    ...
    <lease>
      <lockspace>somearea</lockspace>
      <key>somekey</key>
      <target path='/some/lease/path' offset='1024'/>
    </lease>
    ...
  </devices>

  <!-- USB / PCI / SCSI devices -->

  <devices>
    <hostdev mode='subsystem' type='usb'>
      <source startupPolicy='optional'>
        <vendor id='0x1234'/>
        <product id='0xbeef'/>
      </source>
      <boot order='2'/>
    </hostdev>
  </devices>
  <devices>
```

```
    <hostdev mode='subsystem' type='pci' managed='yes'>
      <source>
        <address domain='0x0000' bus='0x06' slot='0x02'
        function='0x0'/>
      </source>
      <boot order='1'/>
      <rom bar='on' file='/etc/fake/boot.bin'/>
    </hostdev>
  </devices>
  <devices>
    <hostdev mode='subsystem' type='scsi' sgio='filtered'
    rawio='yes'>
      <source>
        <adapter name='scsi_host0'/>
        <address bus='0' target='0' unit='0'/>
      </source>
      <readonly/>
      <address type='drive' controller='0' bus='0' target='0'
      unit='0'/>
    </hostdev>
  </devices>
  <devices>
    <hostdev mode='subsystem' type='scsi'>
      <source protocol='iscsi' name='iqn.2014-08.com.
      example:iscsi-nopool/1'>
        <host name='example.com' port='3260'/>
        <auth username='myuser'>
          <secret type='iscsi' usage='libvirtiscsi'/>
        </auth>
      </source>
      <address type='drive' controller='0' bus='0' target='0'
      unit='0'/>
    </hostdev>
  </devices>
  <devices>
    <hostdev mode='subsystem' type='scsi_host'>
      <source protocol='vhost' wwpn='naa.50014057667280d8'/>
    </hostdev>
  </devices>
  <devices>
    <hostdev mode='subsystem' type='mdev' model='vfio-pci'>
    <source>
      <address uuid='c2177883-f1bb-47f0-914d-32a22e3a8804'/>
    </source>
    </hostdev>
    <hostdev mode='subsystem' type='mdev' model='vfio-ccw'>
    <source>
      <address uuid='9063cba3-ecef-47b6-abcf-3fef4fdcad85'/>
    </source>
    <address type='ccw' cssid='0xfe' ssid='0x0' devno='0x0001'/>
    </hostdev>
  </devices>

  <!-- Block / character devices -->

  <hostdev mode='capabilities' type='storage'>
    <source>
      <block>/dev/sdf1</block>
    </source>
  </hostdev>
  <hostdev mode='capabilities' type='misc'>
    <source>
      <char>/dev/input/event3</char>
    </source>
  </hostdev>
```

```
<hostdev mode='capabilities' type='net'>
  <source>
    <interface>eth0</interface>
  </source>
</hostdev>

<!-- Redirected devices -->

<devices>
  <redirdev bus='usb' type='tcp'>
    <source mode='connect' host='localhost' service='4000'/>
    <boot order='1'/>
  </redirdev>
  <redirfilter>
    <usbdev class='0x08' vendor='0x1234' product='0xbeef'
    version='2.56'
        allow='yes'/>
    <usbdev allow='no'/>
  </redirfilter>
</devices>

<!-- Smartcard devices -->

<devices>
  <smartcard mode='host'/>
  <smartcard mode='host-certificates'>
    <certificate>cert1</certificate>
    <certificate>cert2</certificate>
    <certificate>cert3</certificate>
    <database>/etc/pki/nssdb/</database>
  </smartcard>
  <smartcard mode='passthrough' type='tcp'>
    <source mode='bind' host='127.0.0.1' service='2001'/>
    <protocol type='raw'/>
    <address type='ccid' controller='0' slot='0'/>
  </smartcard>
  <smartcard mode='passthrough' type='spicevmc'/>
</devices>
<devices>
  <interface type='direct' trustGuestRxFilters='yes'>
    <source dev='eth0'/>
    <mac address='52:54:00:5d:c7:9e'/>
    <boot order='1'/>
    <rom bar='off'/>
  </interface>
</devices>

<!-- Network interfaces -->

<devices>
  <interface type='network'>
    <source network='default'/>
  </interface>
  ...
  <interface type='network'>
    <source network='default' portgroup='engineering'/>
    <target dev='vnet7'/>
    <mac address="00:11:22:33:44:55"/>
    <virtualport>
      <parameters instanceid='09b11c53-8b5c-4eeb-8f00-
      d84eaa0aaa4f'/>
    </virtualport>
  </interface>
</devices>
```

```
<!-- Bridge to LAN -->

<devices>
  ...
  <interface type='bridge'>
    <source bridge='br0'/>
  </interface>
  <interface type='bridge'>
    <source bridge='br1'/>
    <target dev='vnet7'/>
    <mac address="00:11:22:33:44:55"/>
  </interface>
  <interface type='bridge'>
    <source bridge='ovsbr'/>
    <virtualport type='openvswitch'>
      <parameters profileid='menial'
          interfaceid='09b11c53-8b5c-4eeb-8f00-d84eaa0aaa4f'/>
    </virtualport>
  </interface>
  ...
</devices>
<devices>
  ...
  <interface type='bridge'>
    <source bridge='br0'/>
  </interface>
  <interface type='bridge'>
    <source bridge='br1'/>
    <target dev='vnet7'/>
    <mac address="00:11:22:33:44:55"/>
  </interface>
  <interface type='bridge'>
    <source bridge='midonet'/>
    <virtualport type='midonet'>
      <parameters interfaceid='0b2d64da-3d0e-431e-afdd-
      804415d6ebbb'/>
    </virtualport>
  </interface>
  ...
</devices>

<!-- Userspace SLIRP stack -->

<devices>
  <interface type='user'/>
  ...
  <interface type='user'>
    <mac address="00:11:22:33:44:55"/>
    <ip family='ipv4' address='172.17.2.0' prefix='24'/>
    <ip family='ipv6' address='2001:db8:ac10:fd01::'
    prefix='64'/>
  </interface>
</devices>

<!-- Generic ethernet connection -->

<devices>
  <interface type='ethernet'/>
  ...
  <interface type='ethernet'>
    <target dev='vnet7'/>
    <script path='/etc/qemu-ifup-mynet'/>
  </interface>
</devices>
```

```
  <devices>
    ...
    <interface type='direct' trustGuestRxFilters='no'>
      <source dev='eth0' mode='vepa'/>
    </interface>
  </devices>
  <devices>
    ...
    <interface type='direct'>
      <source dev='eth0.2' mode='vepa'/>
      <virtualport type="802.1Qbg">
        <parameters managerid="11" typeid="1193047"
        typeidversion="2"
            instanceid="09b11c53-8b5c-4eeb-8f00-d84eaa0aaa4f"/>
      </virtualport>
    </interface>
  </devices>
  <devices>
    ...
    <interface type='direct'>
      <source dev='eth0' mode='private'/>
      <virtualport type='802.1Qbh'>
        <parameters profileid='finance'/>
      </virtualport>
    </interface>
  </devices>
  <devices>
    <interface type='hostdev' managed='yes'>
      <driver name='vfio'/>
      <source>
        <address type='pci' domain='0x0000' bus='0x00' slot='0x07'
            function='0x0'/>
      </source>
      <mac address='52:54:00:6d:90:02'/>
      <virtualport type='802.1Qbh'>
        <parameters profileid='finance'/>
      </virtualport>
    </interface>
  </devices>
  <devices>
    <interface type='mcast'>
      <mac address='52:54:00:6d:90:01'/>
      <source address='230.0.0.1' port='5558'/>
    </interface>
  </devices>
  <devices>
    <interface type='server'>
      <mac address='52:54:00:22:c9:42'/>
      <source address='192.168.0.1' port='5558'/>
    </interface>
    ...
    <interface type='client'>
      <mac address='52:54:00:8b:c9:51'/>
      <source address='192.168.0.1' port='5558'/>
    </interface>
  </devices>
  <devices>
    <interface type='udp'>
      <mac address='52:54:00:22:c9:42'/>
      <source address='127.0.0.1' port='11115'>
        <local address='127.0.0.1' port='11116'/>
      </source>
    </interface>
  </devices>
```

```
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
    <model type='ne2k_pci'/>
  </interface>
</devices>
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
    <model type='virtio'/>
    <driver name='vhost' txmode='iothread' ioeventfd='on'
    event_idx='off'
        queues='5' rx_queue_size='256' tx_queue_size='256'>
      <host csum='off' gso='off' tso4='off' tso6='off'
      ecn='off' ufo='off'
          mrg_rxbuf='off'/>
      <guest csum='off' tso4='off' tso6='off' ecn='off'
      ufo='off'/>
    </driver>
    </interface>
</devices>
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
    <model type='virtio'/>
    <backend tap='/dev/net/tun' vhost='/dev/vhost-net'/>
    <driver name='vhost' txmode='iothread' ioeventfd='on'
    event_idx='off'
        queues='5'/>
    <tune>
      <sndbuf>1600</sndbuf>
    </tune>
  </interface>
</devices>
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
  </interface>
</devices>
<devices>
  <interface type='network'>
    <source network='default'/>
    <guest dev='myeth'/>
  </interface>
</devices>
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
    <boot order='1'/>
  </interface>
</devices>
<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet1'/>
    <rom bar='on' file='/etc/fake/boot.bin'/>
  </interface>
</devices>
```

```
  <devices>
    ...
    <interface type='bridge'>
      <source bridge='br0'/>
      <backenddomain name='netvm'/>
    </interface>
    ...
  </devices>
  <devices>
    <interface type='network'>
      <source network='default'/>
      <target dev='vnet0'/>
      <bandwidth>
        <inbound average='1000' peak='5000' floor='200'
        burst='1024'/>
        <outbound average='128' peak='256' burst='256'/>
      </bandwidth>
    </interface>
  </devices>
  <devices>
    <interface type='bridge'>
      <vlan>
        <tag id='42'/>
      </vlan>
      <source bridge='ovsbr0'/>
      <virtualport type='openvswitch'>
        <parameters interfaceid='09b11c53-8b5c-4eeb-8f00-
        d84eaa0aaa4f'/>
      </virtualport>
    </interface>
    <interface type='bridge'>
      <vlan trunk='yes'>
        <tag id='42'/>
        <tag id='123' nativeMode='untagged'/>
      </vlan>
      ...
    </interface>
  </devices>
  <devices>
    <interface type='network'>
      <source network='default'/>
      <target dev='vnet0'/>
      <link state='down'/>
    </interface>
  </devices>
  <devices>
    <interface type='network'>
      <source network='default'/>
      <target dev='vnet0'/>
      <mtu size='1500'/>
    </interface>
  </devices>
  <devices>
    <interface type='network'>
      <source network='default'/>
      <target dev='vnet0'/>
      <coalesce>
        <rx>
          <frames max='7'/>
        </rx>
      </coalesce>
    </interface>
  </devices>
```

```
<!-- IP configuration -->

<devices>
  <interface type='network'>
    <source network='default'/>
    <target dev='vnet0'/>
    <ip address='192.168.122.5' prefix='24'/>
    <ip address='192.168.122.5' prefix='24'
    peer='10.0.0.10'/>
    <route family='ipv4' address='192.168.122.0' prefix='24'
        gateway='192.168.122.1'/>
    <route family='ipv4' address='192.168.122.8'
    gateway='192.168.122.1'/>
  </interface>
  ...
  <hostdev mode='capabilities' type='net'>
    <source>
      <interface>eth0</interface>
    </source>
    <ip address='192.168.122.6' prefix='24'/>
    <route family='ipv4' address='192.168.122.0' prefix='24'
    gateway='192.168.122.1'/>
    <route family='ipv4' address='192.168.122.8'
    gateway='192.168.122.1'/>
  </hostdev>
  ...
</devices>
<devices>
  <interface type='ethernet'>
    <source/>
      <ip address='192.168.123.1' prefix='24'/>
      <ip address='10.0.0.10' prefix='24' peer='192.168.122.5'/>
      <route family='ipv4' address='192.168.42.0' prefix='24'
      gateway='192.168.123.4'/>
    <source/>
    ...
  </interface>
  ...
</devices>

<!-- vhost-user interface -->

<devices>
  <interface type='vhostuser'>
    <mac address='52:54:00:3b:83:1a'/>
    <source type='unix' path='/tmp/vhost1.sock' mode='server'/>
    <model type='virtio'/>
  </interface>
  <interface type='vhostuser'>
    <mac address='52:54:00:3b:83:1b'/>
    <source type='unix' path='/tmp/vhost2.sock'
    mode='client'>
      <reconnect enabled='yes' timeout='10'/>
    </source>
    <model type='virtio'/>
    <driver queues='5'/>
  </interface>
</devices>

<!-- Traffic filtering with NWFilter -->

<devices>
  <interface ...>
    ...
    <filterref filter='clean-traffic'/>
  </interface>
```

```
    <interface ...>
      ...
      <filterref filter='myfilter'>
        <parameter name='IP' value='104.207.129.11'/>
        <parameter name='IP6_ADDR' value='2001:19f0:300:2102::'/>
        <parameter name='IP6_MASK' value='64'/>
        ...
      </filterref>
    </interface>
  </devices>

  <!-- Input devices -->

  <devices>
    <input type='mouse' bus='usb'/>
    <input type='keyboard' bus='usb'/>
    <input type='mouse' bus='virtio'/>
    <input type='keyboard' bus='virtio'/>
    <input type='tablet' bus='virtio'/>
    <input type='passthrough' bus='virtio'>
      <source evdev='/dev/input/event1/>'
    </input>
  </devices>

  <!-- Hub devices -->

  <devices>
    <hub type='usb'/>
  </devices>

  <!-- Graphical framebuffers -->

  <devices>
    <graphics type='sdl' display=':0.0'/>
    <graphics type='vnc' port='5904' sharePolicy='allow-
    exclusive'>
      <listen type='address' address='1.2.3.4'/>
    </graphics>
    <graphics type='rdp' autoport='yes' multiUser='yes' />
    <graphics type='desktop' fullscreen='yes'/>
    <graphics type='spice'>
      <listen type='network' network='rednet'/>
    </graphics>
  </devices>

  <!-- Video devices -->

  <devices>
    <video>
      <model type='vga' vram='16384' heads='1'>
        <acceleration accel3d='yes' accel2d='yes'/>
      </model>
    </video>
  </devices>

  <!-- Consoles, serial, parallel & channel devices -->

  <devices>
    <parallel type='pty'>
      <source path='/dev/pts/2'/>
      <target port='0'/>
    </parallel>
    <serial type='pty'>
      <source path='/dev/pts/3'/>
      <target port='0'/>
    </serial>
```

```
    <serial type='file'>
      <source path='/tmp/file' append='on'>
        <seclabel model='dac' relabel='no'/>
      </source>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <source path='/dev/pts/4'/>
      <target port='0'/>
    </console>
    <channel type='unix'>
      <source mode='bind' path='/tmp/guestfwd'/>
      <target type='guestfwd' address='10.0.2.1' port='4600'/>
    </channel>
  </devices>
  <devices>
    <parallel type='pty'>
      <source path='/dev/pts/2'/>
      <target port='0'/>
    </parallel>
  </devices>
  <devices>
    <!-- Serial port -->
    <serial type='pty'>
      <source path='/dev/pts/3'/>
      <target port='0'/>
    </serial>
  </devices>
  <devices>
    <!-- USB serial port -->
    <serial type='pty'>
      <target type='usb-serial' port='0'>
        <model name='usb-serial'/>
      </target>
      <address type='usb' bus='0' port='1'/>
    </serial>
  </devices>
  <devices>
    <!-- Serial console -->
    <console type='pty'>
      <source path='/dev/pts/2'/>
     <target type='serial' port='0'/>
    </console>
  </devices>
  <devices>
    <!-- KVM virtio console -->
    <console type='pty'>
      <source path='/dev/pts/5'/>
      <target type='virtio' port='0'/>
    </console>
  </devices>
  <devices>
    <console type='pty'>
      <target type='serial'/>
    </console>
    <console type='pty'>
      <target type='virtio'/>
    </console>
  </devices>
  <devices>
    <serial type='pty'/>
  </devices>
```

```
  <devices>
    <console type='pty'/>
  </devices>
  <devices>
    <serial type='pty'/>
    <console type='pty'/>
  </devices>
  <devices>
    <channel type='unix'>
      <source mode='bind' path='/tmp/guestfwd'/>
      <target type='guestfwd' address='10.0.2.1' port='4600'/>
    </channel>
    <!-- KVM virtio channel -->
    <channel type='pty'>
      <target type='virtio' name='arbitrary.virtio.serial.port.
      name'/>
    </channel>
    <channel type='unix'>
      <source mode='bind' path='/var/lib/libvirt/qemu/
      f16x86_64.agent'/>
      <target type='virtio' name='org.qemu.guest_agent.0'
      state='connected'/>
    </channel>
    <channel type='spicevmc'>
      <target type='virtio' name='com.redhat.spice.0'/>
    </channel>
  </devices>
  <devices>
    <console type='stdio'>
      <target port='1'/>
    </console>
  </devices>
```

```
<devices>
  <serial type="file">
    <source path="/var/log/vm/vm-serial.log"/>
    <target port="1"/>
  </serial>
</devices>
<devices>
  <serial type='vc'>
    <target port="1"/>
  </serial>
</devices>
  <devices>
  <serial type='null'>
    <target port="1"/>
  </serial>
</devices>
<devices>
  <serial type="pty">
    <source path="/dev/pts/3"/>
    <target port="1"/>
  </serial>
</devices>
<devices>
  <serial type="dev">
    <source path="/dev/ttyS0"/>
    <target port="1"/>
  </serial>
</devices>
<devices>
  <serial type="pipe">
    <source path="/tmp/mypipe"/>
```

```
      <target port="1"/>
    </serial>
  </devices>
  <devices>
    <serial type="tcp">
      <source mode="connect" host="0.0.0.0" service="2445"/>
      <protocol type="raw"/>
      <target port="1"/>
    </serial>
  </devices>
  <devices>
    <serial type="tcp">
      <source mode="bind" host="127.0.0.1" service="2445"/>
      <protocol type="raw"/>
      <target port="1"/>
    </serial>
  </devices>
  <devices>
    <serial type="tcp">
      <source mode="connect" host="0.0.0.0" service="2445"/>
      <protocol type="telnet"/>
      <target port="1"/>
    </serial>
    ...
    <serial type="tcp">
      <source mode="bind" host="127.0.0.1" service="2445"/>
      <protocol type="telnet"/>
      <target port="1"/>
    </serial>
  </devices>
```

```
<devices>
  <serial type="tcp">
    <source mode='connect' host="127.0.0.1" service="5555"
    tls="yes"/>
    <protocol type="raw"/>
    <target port="0"/>
  </serial>
</devices>
<devices>
  <serial type="udp">
    <source mode="bind" host="0.0.0.0" service="2445"/>
    <source mode="connect" host="0.0.0.0" service="2445"/>
    <target port="1"/>
  </serial>
</devices>
<devices>
  <serial type="unix">
    <source mode="bind" path="/tmp/foo"/>
    <target port="1"/>
  </serial>
</devices>
<devices>
  <serial type="spiceport">
    <source channel="org.qemu.console.serial.0"/>
    <target port="1"/>
  </serial>
</devices>
<devices>
  <serial type="nmdm">
    <source master="/dev/nmdm0A" slave="/dev/nmdm0B"/>
  </serial>
</devices>
```

```
  <devices>
    <sound model='es1370'/>
  </devices>
  <devices>
    <sound model='ich6'>
      <codec type='micro'/>
    </sound>
  </devices>
  <devices>
    <watchdog model='i6300esb'/>
  </devices>
  <devices>
    <watchdog model='i6300esb' action='poweroff'/>
  </devices>
  <devices>
    <memballoon model='virtio'/>
  </devices>
  <devices>
    <memballoon model='virtio'>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x02'
          function='0x0'/>
      <stats period='10'/>
      <driver iommu='on' ats='on'/>
    </memballoon>
  </devices>
  <devices>
    <rng model='virtio'>
      <rate period="2000" bytes="1234"/>
      <backend model='random'>/dev/random</backend>
      <!-- OR -->
      <backend model='egd' type='udp'>
        <source mode='bind' service='1234'/>
        <source mode='connect' host='1.2.3.4' service='1234'/>
      </backend>
    </rng>
  </devices>

  <!-- TPM device -->

  <devices>
    <tpm model='tpm-tis'>
      <backend type='passthrough'>
        <device path='/dev/tpm0'/>
      </backend>
    </tpm>
  </devices>
  <devices>
    <tpm model='tpm-tis'>
      <backend type='emulator' version='2.0'>
      </backend>
    </tpm>
  </devices>

  <!-- NVRAM device -->

  <devices>
    <nvram>
      <address type='spapr-vio' reg='0x3000'/>
    </nvram>
  </devices>
  <devices>
    <panic model='hyperv'/>
    <panic model='isa'>
      <address type='isa' iobase='0x505'/>
    </panic>
  </devices>
```

```
<!-- Shared memory devic -->

<devices>
  <shmem name='my_shmem0'>
    <model type='ivshmem-plain'/>
    <size unit='M'>4</size>
  </shmem>
  <shmem name='shmem_server'>
    <model type='ivshmem-doorbell'/>
    <size unit='M'>2</size>
    <server path='/tmp/socket-shmem'/>
    <msi vectors='32' ioeventfd='on'/>
  </shmem>
</devices>

<!-- Memory devices -->

<devices>
  <memory model='dimm' access='private' discard='yes'>
    <target>
      <size unit='KiB'>524287</size>
      <node>0</node>
    </target>
  </memory>
  <memory model='dimm'>
    <source>
      <pagesize unit='KiB'>4096</pagesize>
      <nodemask>1-3</nodemask>
    </source>
    <target>
      <size unit='KiB'>524287</size>
      <node>1</node>
    </target>
  </memory>
  <!-- IOMMU devices -->

 <devices>
  <iommu model='intel'>
    <driver intremap='on'/>
  </iommu>
</devices>

  <memory model='nvdimm'>
    <source>
      <path>/tmp/nvdimm</path>
      <alignsize unit='KiB'>2048</alignsize>
    </source>
    <target>
      <size unit='KiB'>524288</size>
      <node>1</node>
      <label>
        <size unit='KiB'>128</size>
      </label>
      <readonly/>
    </target>
  </memory>
  <memory model='nvdimm'>
    <source>
      <path>/dev/dax0.0</path>
      <pmem/>
    </source>
    <target>
      <size unit='KiB'>524288</size>
      <node>1</node>
      <label>
        <size unit='KiB'>128</size>
      </label>
    </target>
  </memory>
</devices>

<!-- IOMMU devices -->

<devices>
  <iommu model='intel'>
    <driver intremap='on'/>
  </iommu>
</devices>

<!-- Vsock -->

<devices>
  <vsock model='virtio'>
    <cid auto='no' address='3'/>
  </vsock>
</devices>

<!-- Security label -->

<seclabel type='dynamic' model='selinux'/>
<seclabel type='dynamic' model='selinux'>
  <baselabel>system_u:system_r:my_svirt_t:s0</baselabel>
</seclabel>
<seclabel type='static' model='selinux' relabel='no'>
  <label>system_u:system_r:svirt_t:s0:c392,c662</label>
</seclabel>
<seclabel type='static' model='selinux' relabel='yes'>
  <label>system_u:system_r:svirt_t:s0:c392,c662</label>
</seclabel>
<seclabel type='none'/>
<!-- Key Wrap -->

<keywrap>
  <cipher name='aes' state='off'/>
</keywrap>

<!-- Launch Security -->

<launchSecurity type='sev'>
  <policy>0x0001</policy>
  <cbitpos>47</cbitpos>
  <reducedPhysBits>1</reducedPhysBits>
  <dhCert>RBBBSDDD=FDDCCCDDDG</dhCert>
  <session>AAACCCDD=FFFCCCDSDS</session>
</launchSecurity>
</domain>
```

The following are some general rules for the schema:

- Some attributes on some elements can be safely omitted.
- Devices not available to the domain can be safely omitted.
- Some devices can be configured in different ways. Use the method that closely matches the device.
- In many cases, libvirt can discover and configure devices on its own. If the discovery and config are correct, there is no need to configure the device via XML.
- It is possible to add attributes and elements later based on device availability.

Example Paravirtualized XEN Schemas

The following are some example XML configurations for Xen guest domains. For full details of the available options, consult the libvirt domain XML documentation.

Paravirtualized Guest Bootloader

Using a bootloader allows a paravirtualized guest to be booted using a kernel stored inside its virtual disk image (Listing A-2).

Listing A-2. Paravirtualized Guest Bootloader

```
<domain type='xen'>
  <name>fc8</name>
  <bootloader>/usr/bin/pygrub</bootloader>
  <os>
    <type>linux</type>
  </os>
  <memory>131072</memory>
  <vcpu>1</vcpu>
  <devices>
    <disk type='file'>
      <source file='/var/lib/xen/images/fc4.img'/>
      <target dev='sda1'/>
    </disk>
    <interface type='bridge'>
      <source bridge='xenbr0'/>
      <mac address='aa:00:00:00:00:11'/>
      <script path='/etc/xen/scripts/vif-bridge'/>
    </interface>
    <console tty='/dev/pts/5'/>
  </devices>
</domain>
```

Paravirtualized Guest Direct Kernel Boot

To install paravirtualized guests, it is typical to boot the domain using a kernel and initrd stored in the host OS (Listing A-3).

Listing A-3. Paravirtualized Direct Kernel Boot

```
<domain type='xen'>
  <name>fc8</name>
  <os>
    <type>linux</type>
    <kernel>/var/lib/xen/install/vmlinuz-fedora8-x86_64
    </kernel>
    <initrd>/var/lib/xen/install/initrd-vmlinuz-fedora8-x86_64
    </initrd>
    <cmdline> kickstart=http://example.com/myguest.ks </cmdline>
  </os>
  <memory>131072</memory>
  <vcpu>1</vcpu>
  <devices>
    <disk type='file'>
      <source file='/var/lib/xen/images/fc4.img'/>
      <target dev='sda1'/>
    </disk>
    <interface type='bridge'>
      <source bridge='xenbr0'/>
      <mac address='aa:00:00:00:00:11'/>
      <script path='/etc/xen/scripts/vif-bridge'/>
    </interface>
    <graphics type='vnc' port='-1'/>
    <console tty='/dev/pts/5'/>
  </devices>
</domain>
```

Fully Virtualized Guest BIOS Boot

Fully virtualized guests use the emulated BIOS to boot off the primary hard disk, CD-ROM, or network PXE ROM (Listing A-4) .

Listing A-4. Fully Virtualized Guest BIOS Boot

```
<domain type='xen' id='3'>
  <name>fv0</name>
  <uuid>4dea22b31d52d8f32516782e98ab3fa0</uuid>
  <os>
    <type>hvm</type>
    <loader>/usr/lib/xen/boot/hvmloader</loader>
    <boot dev='hd'/>
  </os>
  <memory>524288</memory>
  <vcpu>1</vcpu>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <features>
    <pae/>
    <acpi/>
    <apic/>
  </features>
  <clock sync="localtime"/>
  <devices>
    <emulator>/usr/lib/xen/bin/qemu-dm</emulator>
    <interface type='bridge'>
      <source bridge='xenbr0'/>
      <mac address='00:16:3e:5d:c7:9e'/>
      <script path='vif-bridge'/>
    </interface>
    <disk type='file'>
      <source file='/var/lib/xen/images/fv0'/>
      <target dev='hda'/>
    </disk>
    <disk type='file' device='cdrom'>
      <source file='/var/lib/xen/images/fc5-x86_64-boot.iso'/>
      <target dev='hdc'/>
      <readonly/>
    </disk>
    <disk type='file' device='floppy'>
      <source file='/root/fd.img'/>
      <target dev='fda'/>
    </disk>
    <graphics type='vnc' port='5904'/>
  </devices>
</domain>
```

Fully Virtualized Guest Direct Kernel Boot

With Xen 3.2.0 or newer, it is possible to bypass the BIOS and directly boot a Linux kernel and initrd as a fully virtualized domain (Listing A-5). This allows for complete automation of the OS installation, for example, using the Anaconda kickstart support to install the domain.

Listing A-5. Fully Virtualized Guest Direct Kernel Boot

```
<domain type='xen' id='3'>
  <name>fv0</name>
  <uuid>4dea22b31d52d8f32516782e98ab3fa0</uuid>
  <os>
    <type>hvm</type>
    <loader>/usr/lib/xen/boot/hvmloader</loader>
    <kernel>/var/lib/xen/install/vmlinuz-fedora8-x86_64</kernel>
    <initrd>/var/lib/xen/install/initrd-vmlinuz-
    fedora8-x86_64</initrd>
    <cmdline> kickstart=http://example.com/myguest.ks </cmdline>
  </os>
  <memory>524288</memory>
  <vcpu>1</vcpu>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>restart</on_crash>
  <features>
    <pae/>
    <acpi/>
    <apic/>
  </features>
  <clock sync="localtime"/>
  <devices>
    <emulator>/usr/lib/xen/bin/qemu-dm</emulator>
    <interface type='bridge'>
      <source bridge='xenbr0'/>
      <mac address='00:16:3e:5d:c7:9e'/>
      <script path='vif-bridge'/>
    </interface>
    <disk type='file'>
      <source file='/var/lib/xen/images/fv0'/>
      <target dev='hda'/>
    </disk>
    <disk type='file' device='cdrom'>
      <source file='/var/lib/xen/images/fc5-x86_64-boot.iso'/>
      <target dev='hdc'/>
      <readonly/>
    </disk>
    <disk type='file' device='floppy'>
      <source file='/root/fd.img'/>
      <target dev='fda'/>
    </disk>
    <graphics type='vnc' port='5904'/>
  </devices>
</domain>
```

Example Paravirtualized KVM and Qemu Schemas

Paravirtualized schemas are also supported by specifying the emulation to be used.

QEMU Emulated Guest on x86_64

The XML in Listing A-6 specifies that the QEMU emulator is to be used to support the domain.

Listing A-6. QEMU Emulated Guest on x86_64

```
<domain type='qemu'>
  <name>QEMU-fedora-i686</name>
  <uuid>c7a5fdbd-cdaf-9455-926a-d65c16db1809</uuid>
  <memory>219200</memory>
  <currentMemory>219200</currentMemory>
  <vcpu>2</vcpu>
  <os>
    <type arch='i686' machine='pc'>hvm</type>
    <boot dev='cdrom'/>
  </os>
  <devices>
```

```
    <emulator>/usr/bin/qemu-system-x86_64</emulator>
    <disk type='file' device='cdrom'>
      <source file='/home/user/boot.iso'/>
      <target dev='hdc'/>
      <readonly/>
    </disk>
    <disk type='file' device='disk'>
      <source file='/home/user/fedora.img'/>
      <target dev='hda'/>
    </disk>
    <interface type='network'>
      <source network='default'/>
    </interface>
    <graphics type='vnc' port='-1'/>
  </devices>
</domain>
```

KVM Hardware Accelerated Guest on i686

Listing A-7 specifies KVM as the emulator for the guest domain.

Listing A-7. KVM Hardware-Accelerated Guest on i686

```
<domain type='kvm'>
  <name>demo2</name>
  <uuid>4dea24b3-1d52-d8f3-2516-782e98a23fa0</uuid>
  <memory>131072</memory>
  <vcpu>1</vcpu>
  <os>
    <type arch="i686">hvm</type>
  </os>
  <clock sync="localtime"/>
  <devices>
    <emulator>/usr/bin/qemu-kvm</emulator>
    <disk type='file' device='disk'>
      <source file='/var/lib/libvirt/images/demo2.img'/>
      <target dev='hda'/>
    </disk>
    <interface type='network'>
      <source network='default'/>
      <mac address='24:42:53:21:52:45'/>
    </interface>
    <graphics type='vnc' port='-1' keymap='de'/>
  </devices>
</domain>
```

The Networks Schema

The root element required for all virtual networks is named network and has no configurable attributes. (However, since 0.10.0, there is one optional read-only attribute; when examining the live configuration of a network, the attribute connections, if present, specify the number of guest interfaces currently connected via this network.) The network XML format has been available since 0.3.0.

The Network XML schema consists of a number of XML elements. The documentation for this schema is extensive but in most cases the uses of the elements in the schema are obvious. If you need documentation, refer to https://libvirt.org/formatnetwork.html for more specific information on each element and attribute.

The Network Schema Details

Listing A-8 shows the first elements that provide basic metadata about the virtual network.

Listing A-8. General Metadata

```
<network ipv6='yes' trustGuestRxFilters='no'>
  <name>default</name>
  <uuid>3e3fce45-4f53-4fa7-bb32-11f34168b82b</uuid>
  <metadata>
    <app1:foo xmlns:app1="http://app1.org/app1/">..</app1:foo>
    <app2:bar xmlns:app2="http://app1.org/app2/">..</app2:bar>
  </metadata>
```

The Connectivity Schema

The next set of elements control how a virtual network is provided with connectivity to the physical LAN (if at all). See Listing A-9.

Listing A-9. Connectivity

```
  <bridge name="virbr0" stp="on" delay="5"
  macTableManager="libvirt"/>
<mtu size="9000"/>
<domain name="example.com" localOnly="no"/>
<forward mode="nat" dev="eth0"/>
```

Listing A-10 shows how to specify network forwarding.

Listing A-10. Forwarding

```
  <forward mode='passthrough'>
    <interface dev='eth10'/>
    <interface dev='eth11'/>
    <interface dev='eth12'/>
    <interface dev='eth13'/>
    <interface dev='eth14'/>
  </forward>
```

Listing A-11 shows how to specify network passthrough.

Listing A-11. Passthrough with a Specific Device

```
  <forward mode='passthrough'>
    <pf dev='eth0'/>
  </forward>
```

Listing A-12 shows how to specify forwarding for a specific driver.

Listing A-12. Forwarding Using a Specific Driver

```
  <forward mode='hostdev' managed='yes'>
    <driver name='vfio'/>
    <address type='pci' domain='0' bus='4' slot='0' function='1'/>
    <address type='pci' domain='0' bus='4' slot='0' function='2'/>
    <address type='pci' domain='0' bus='4' slot='0' function='3'/>
  </forward>
```

Listing A-13 shows how to specify forwarding for a managed device.

Listing A-13. Forwarding with a Managed Device

```
  <forward mode='hostdev' managed='yes'>
    <pf dev='eth0'/>
  </forward>
```

Listing A-14 shows how to specify forwarding with a specific device with limits.

Listing A-14. Forwarding a Specific Device with Limits

```
  <forward mode='nat' dev='eth0'/>
  <bandwidth>
    <inbound average='1000' peak='5000' burst='5120'/>
    <outbound average='128' peak='256' burst='256'/>
  </bandwidth>
```

Listing A-15 shows how to forward a specific device with limits.

Listing A-15. Forwarding a Specific Device with Limits

```
  <forward mode='nat' dev='eth0'/>
  <bandwidth>
    <inbound average='1000' peak='5000' burst='5120'/>
    <outbound average='128' peak='256' burst='256'/>
  </bandwidth>
```

Setting a VLAN Tag (on Supported Network Types Only)

Listing A-16 shows how to set the VLAN tag on networks. The VLAN tag specifies how the virtual network interacts with the main host's real network connection.

Listing A-16. Setting a VLAN Tag

```
<network>
  <name>ovs-net</name>
  <forward mode='bridge'/>
  <bridge name='ovsbr0'/>
  <virtualport type='openvswitch'>
```

```
    <parameters interfaceid='09b11c53-8b5c-4eeb-8f00-
    d84eaa0aaa4f'/>
  </virtualport>
  <vlan trunk='yes'>
    <tag id='42' nativeMode='untagged'/>
    <tag id='47'/>
  </vlan>
  <portgroup name='dontpanic'>
    <vlan>
      <tag id='42'/>
    </vlan>
  </portgroup>
</network>
```

Portgroups

Listing A-17 shows the port interaction with the host's ports.

Listing A-17. Portgroups

```
<network>
  <name>ovs-net</name>
  <forward mode='bridge'/>
  <bridge name='ovsbr0'/>
  <virtualport type='openvswitch'>
    <parameters interfaceid='09b11c53-8b5c-4eeb-8f00-
    d84eaa0aaa4f'/>
  </virtualport>
  <vlan trunk='yes'>
    <tag id='42' nativeMode='untagged'/>
    <tag id='47'/>
  </vlan>
  <portgroup name='dontpanic'>
    <vlan>
      <tag id='42'/>
    </vlan>
  </portgroup>
</network>
```

Static Routes

Listing A-18 shows how static routes can interface with the host's real network.

Listing A-18. Static Routes

```
  <ip address="192.168.122.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.122.128" end="192.168.122.254"/>
    </dhcp>
  </ip>
  <route address="192.168.222.0" prefix="24"
  gateway="192.168.122.2"/>
  <ip family="ipv6" address="2001:db8:ca2:2::1" prefix="64"/>
  <route family="ipv6" address="2001:db8:ca2:3::" prefix="64"
  gateway="2001:db8:ca2:2::2"/>
  <route family="ipv6" address="2001:db9:4:1::" prefix="64"
  gateway="2001:db8:ca2:2::3" metric='2'/>
```

Addressing

Listing A-19 shows how to set up static routes to the real network.

Listing A-19. Static Routes

```
<mac address='00:16:3E:5D:C7:9E'/>
<domain name="example.com"/>
<dns>
  <txt name="example" value="example value"/>
  <forwarder addr="8.8.8.8"/>
  <forwarder domain='example.com' addr="8.8.4.4"/>
  <forwarder domain='www.example.com'/>
  <srv service='name' protocol='tcp' domain='test-domain-
  name' target='.'
    port='1024' priority='10' weight='10'/>
  <host ip='192.168.122.2'>
    <hostname>myhost</hostname>
    <hostname>myhostalias</hostname>
  </host>
</dns>
<ip address="192.168.122.1" netmask="255.255.255.0"
localPtr="yes">
  <dhcp>
    <range start="192.168.122.100" end="192.168.122.254"/>
    <host mac="00:16:3e:77:e2:ed" name="foo.example.com"
    ip="192.168.122.10"/>
    <host mac="00:16:3e:3e:a9:1a" name="bar.example.com"
    ip="192.168.122.11"/>
  </dhcp>
</ip>
<ip family="ipv6" address="2001:db8:ca2:2::1" prefix="64"
localPtr="yes"/>
<route family="ipv6" address="2001:db9:ca1:1::" prefix="64"
 gateway="2001:db8:ca2:2::2"/>
```

Example Configurations

The following sections review some of the more frequently used network configurations.

NAT-Based Network

This example is the so-called "default" virtual network. It is provided and enabled out of the box for all libvirt installations. This is a configuration that allows the guest OS to get outbound connectivity regardless of whether the host uses Ethernet, wireless, dial-up, or VPN networking without requiring any specific admin configuration. In the absence of host networking, it at least allows guests to talk directly to each other. See Listing A-20.

Listing A-20. The Default Virtual Network Configuration

```
<network>
  <name>default</name>
  <bridge name="virbr0"/>
  <forward mode="nat"/>
  <ip address="192.168.122.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.122.2" end="192.168.122.254"/>
    </dhcp>
  </ip>
  <ip family="ipv6" address="2001:db8:ca2:2::1" prefix="64"/>
</network
```

Listing A-21 is a variation of the previous example that adds an IPv6 DHCP range definition.

Listing A-21. A Variation of the Default Network Configuration

```
<network>
  <name>default</name>
  <bridge name="virbr0"/>
  <forward mode="nat"/>
  <ip address="192.168.122.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.122.2" end="192.168.122.254"/>
    </dhcp>
  </ip>
  <ip family="ipv6" address="2001:db8:ca2:2::1" prefix="64"/>
</network
```

Routed Network Config

This is a variant on the default network that routes traffic from the virtual network to the LAN without applying any NAT. It requires that the IP address range be preconfigured in the routing tables of the router on the host network. Listing A-22 further specifies that guest traffic may go out only via the `eth1` host network device.

Listing A-22. A Routed Default Network Configuration

```
<network>
  <name>local</name>
  <bridge name="virbr1"/>
  <forward mode="route" dev="eth1"/>
  <ip address="192.168.122.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.122.2" end="192.168.122.254"/>
    </dhcp>
  </ip>
  <ip family="ipv6" address="2001:db8:ca2:2::1" prefix="64"/>
</network>
```

Listing A-23 is another IPv6 variation. Instead of a DHCP range being specified, this example has a couple of IPv6 host definitions. Note that most of the DHCP host definitions use an `id` (client ID or DUID) since this has proven to be a more reliable way of specifying the interface and its association with an IPv6 address. The first is a DUID-LLT, the second a DUID-LL, and the third a DUID-UUID. Since 1.0.3.

Listing A-23. A Routed IPv6 Network Configuration

```
<network>
  <name>local6</name>
  <bridge name="virbr1"/>
  <forward mode="route" dev="eth1"/>
  <ip address="192.168.122.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.122.2" end="192.168.122.254"/>
    </dhcp>
  </ip>
  <ip family="ipv6" address="2001:db8:ca2:2::1" prefix="64">
    <dhcp>
      <host name="paul" ip="2001:db8:ca2:2:3::1"/>
      <host id="0:1:0:1:18:aa:62:fe:0:16:3e:44:55:66"
      ip="2001:db8:ca2:2:3::2"/>
      <host id="0:3:0:1:0:16:3e:11:22:33" name="ralph"
      ip="2001:db8:ca2:2:3::3"/>
      <host id="0:4:7e:7d:f0:7d:a8:bc:c5:d2:13:32:11:ed:16:
      ea:84:63"
        name="badbob" ip="2001:db8:ca2:2:3::4"/>
    </dhcp>
  </ip>
</network>
```

Listing A-24 is yet another IPv6 variation. This variation has only IPv6 defined with DHCPv6 on the primary IPv6 network. A static link is defined for a second IPv6 network, which will not be directly visible on the bridge interface, but there will be a static route defined for this network via the specified gateway. Note that the gateway address must be directly reachable via (on the same subnet as) one of the `<ip>` addresses defined for this `<network>`. Since 1.0.6.

Listing A-24. Another Routed IPv6 Network Configuration

```
<network>
  <name>net7</name>
  <bridge name="virbr7"/>
  <forward mode="route"/>
  <ip family="ipv6" address="2001:db8:ca2:7::1" prefix="64">
    <dhcp>
      <range start="2001:db8:ca2:7::100" end="2001:db8:ca2::1ff"/>
      <host id="0:4:7e:7d:f0:7d:a8:bc:c5:d2:13:32:11:ed:16:
      ea:84:63"
        name="lucas" ip="2001:db8:ca2:2:3::4"/>
    </dhcp>
  </ip>
  <route family="ipv6" address="2001:db8:ca2:8::" prefix="64"
  gateway="2001:db8:ca2:7::4"/>
</network>
```

Isolated Network Config

This variant provides a completely isolated private network for guests. The guests can talk to each other and the host OS but cannot reach any other machines on the LAN because of the omission of the forward element in the XML description. Listing A-25 shows an entire class C network being specified, but it is possible to start with a smaller range.

Listing A-25. Isolated Network

```
<network>
  <name>private</name>
  <bridge name="virbr2"/>
  <ip address="192.168.152.1" netmask="255.255.255.0">
    <dhcp>
      <range start="192.168.152.2" end="192.168.152.254"/>
    </dhcp>
  </ip>
  <ip family="ipv6" address="2001:db8:ca2:3::1" prefix="64"/>
</network>
```

Isolated IPv6 Network Config

This variation of an isolated network defines only IPv6. Note that most of the DHCP host definitions use an ID (a client ID or DUID) since this has proven to be a more reliable way of specifying the interface and its association with an IPv6 address (Listing A-26). The first is a DUID-LLT, the second a DUID-LL, and the third a DUID-UUID. Since version 1.0.3.

Listing A-26. Isolated IPv6 Network

```
<network>
  <name>sixnet</name>
  <bridge name="virbr6"/>
  <ip family="ipv6" address="2001:db8:ca2:6::1" prefix="64">
    <dhcp>
      <host name="peter" ip="2001:db8:ca2:6:6::1"/>
      <host id="0:1:0:1:18:aa:62:fe:0:16:3e:44:55:66"
      ip="2001:db8:ca2:6:6::2"/>
      <host id="0:3:0:1:0:16:3e:11:22:33" name="dariusz"
      ip="2001:db8:ca2:6:6::3"/>
      <host id="0:4:7e:7d:f0:7d:a8:bc:c5:d2:13:32:11:ed:16:
      ea:84:63"
        name="anita" ip="2001:db8:ca2:6:6::4"/>
    </dhcp>
  </ip>
</network>
```

Using an Existing Host Bridge

Since 0.9.4.

Listing A-27 shows how to use a preexisting host bridge, called `br0`. The guests will effectively be directly connected to the physical network. (In other words, their IP addresses will all be on the subnet of the physical network, and there will be no restrictions on inbound or outbound connections.)

Listing A-27. Using an Existing Host Bridge

```
<network>
  <name>host-bridge</name>
  <forward mode="bridge"/>
  <bridge name="br0"/>
</network>
```

Using a macvtap "Direct" Connection

Since version 0.9.4, QEMU and KVM requires Linux kernel 2.6.34 or newer. Listing A-28 shows how to use macvtap to connect to the physical network directly through one of a group of physical devices (without using a host bridge device). As with the host bridge network, the guests will effectively be directly connected to the physical network so their IP addresses will all be on the subnet of the physical network, and there will be no restrictions on inbound or outbound connections. Note that because of a limitation in the implementation of macvtap, these connections do not allow communication directly between the host and the guests. If you require this, you will either need the attached physical switch to be operating in a mirroring mode (so that all traffic coming to the switch is reflected back to the host's interface) or provide alternate means for this communication (e.g., a second interface on each guest that is connected to an isolated network). The other forward modes that use macvtap (private, vepa, and passthrough) would be used in a similar fashion.

Listing A-28. Using a macvtap "Direct" Connection

```
<network>
  <name>direct-macvtap</name>
  <forward mode="bridge">
    <interface dev="eth20"/>
    <interface dev="eth21"/>
    <interface dev="eth22"/>
    <interface dev="eth23"/>
    <interface dev="eth24"/>
  </forward>
</network>
```

Network Config with No Gateway Address

A valid network definition can contain no IPv4 or IPv6 addresses. Such a definition can be used for a "very private" or "very isolated" network since it will not be possible to communicate with the virtualization host via this network. However, this virtual network interface can be used for communication between virtual guest systems. This works for IPv4 and (since 1.0.1) IPv6. However, `ipv6='yes'` must be added for guest-to-guest IPv6 communication (Listing A-29).

Listing A-29. Network Config with No Gateway Address

```
<network ipv6='yes'>
  <name>nogw</name>
  <uuid>7a3b7497-1ec7-8aef-6d5c-38dff9109e93</uuid>
  <bridge name="virbr2" stp="on" delay="0"/>
  <mac address='00:16:3E:5D:C7:9E'/>
</network>
```

Storage Pool XML

Although all storage pool back ends share the same public APIs and XML format, they have varying levels of capabilities. Some may allow the creation of volumes; others may allow only the use of preexisting volumes. Some may have constraints on volume size or placement.

The top-level tag for a storage pool document is `pool`. It has a single attribute type, which is one of `dir`, `fs`, `netfs`, `disk`, `iscsi`, `logical`, `scsi` (all since 0.4.1), `mpath` (since 0.7.1), `rbd` (since 0.9.13), `sheepdog` (since 0.10.0), `gluster` (since 1.2.0), `zfs` (since 1.2.8), or `vstorage` (since 3.1.0). This corresponds to the storage back-end drivers listed later in this appendix.

General Metadata

Listing A-30 shows how to specify the metadata for a storage pool.

Listing A-30. Storage Pool Metadata

```
<pool type="iscsi">
  <name>virtimages</name>
  <uuid>3e3fce45-4f53-4fa7-bb32-11f34168b82b</uuid>
  <allocation>10000000</allocation>
  <capacity>50000000</capacity>
  <available>40000000</available>
```

Source Elements

The following examples show how to specify the source elements for a storage pool.

Specifically, Listing A-31 shows how to specify an iSCSI device for the storage pool.

Listing A-31. Using an iSCSI Device for the Storage Pool

```
<source>
  <host name="iscsi.example.com"/>
  <device path="iqn.2013-06.com.example:iscsi-pool"/>
  <auth type='chap' username='myname'>
    <secret usage='mycluster_myname'/>
  </auth>
  <vendor name="Acme"/>
  <product name="model"/>
</source>
```

Listing A-32 shows how to specify a file path for the storage pool.

Listing A-32. Using a File Path for the Storage Pool

```
<source>
  <device path='/dev/mapper/mpatha' part_separator='no'/>
  <format type='gpt'/>
</source>
```

Listing A-33 shows how to use a SCSI device for the storage pool.

Listing A-33. Using a SCSI Device for the Storage Pool

```
<source>
  <adapter type='scsi_host' name='scsi_host1'/>
</source>
```

Listing A-34 shows how to use a SCSI LUN for the storage pool.

Listing A-34. Using a SCSI LUN for the Storage Pool

```
<source>
  <adapter type='scsi_host'>
    <parentaddr unique_id='1'>
      <address domain='0x0000' bus='0x00' slot='0x1f' addr='0x2'/>
    </parentaddr>
  </adapter>
</source>
```

Listing A-35 shows how to use a SCSI host for the storage pool.

Listing A-35. Using a SCSI Host for the Storage Pool

```
<source>
  <adapter type='fc_host' parent='scsi_host5'
  wwnn='20000000c9831b4b'        wwpn='10000000c9831b4b'/>
</source>
```

Target Elements

A single target element is contained within the top-level pool element for some types of pools (pool types `dir`, `fs`, `netfs`, `logical`, `disk`, `iscsi`, `scsi`, `mpath`, `zfs`). This tag is used to describe the mapping of the storage pool into the host filesystem. It can contain the child elements shown in Listing A-36.

Listing A-36. Specifying a Single Target in the Top-Level Element

```
  <target>
    <path>/dev/disk/by-path</path>
    <permissions>
      <owner>107</owner>
      <group>107</group>
      <mode>0744</mode>
      <label>virt_image_t</label>
    </permissions>
  </target>
</pool>
```

Storage Volume XML

A storage volume will generally be either a file or a device node; since 1.2.0, an optional output-only attribute type lists the actual type (`file`, `block`, `dir`, `network`, `netdir`, or `ploop`), which is also available from `virStorageVolGetInfo()`. The storage volume XML format has been available since 0.4.1.

Storage Volume XML

A storage volume will generally be either a file or a device node; since 1.2.0, an optional output-only attribute type lists the actual type (`file`, `block`, `dir`, `network`, `netdir`, or `ploop`), which is also available from `virStorageVolGetInfo()`. The storage volume XML format has been available since 0.4.1 (Listing A-37).

Listing A-37. How to Specify a Storage Volume

```
  <volume type='file'>
  <name>sparse.img</name>
  <key>/var/lib/xen/images/sparse.img</key>
  <allocation>0</allocation>
  <capacity unit="T">1</capacity>
```

Target Elements

A single target element is contained within the top-level volume element. This tag is used to describe the mapping of the storage volume into the host file system. It can contain the child elements in Listing A-38.

Listing A-38. Specifying a Single Target Element for the Storage Volume

```
<target>
  <path>/var/lib/virt/images/sparse.img</path>
  <format type='qcow2'/>
  <permissions>
    <owner>107</owner>
    <group>107</group>
    <mode>0744</mode>
    <label>virt_image_t</label>
  </permissions>
  <timestamps>
    <atime>1341933637.273190990</atime>
    <mtime>1341930622.047245868</mtime>
    <ctime>1341930622.047245868</ctime>
  </timestamps>
  <encryption type='...'>
    ...
  </encryption>
  <compat>1.1</compat>
  <nocow/>
  <features>
    <lazy_refcounts/>
  </features>
</target>
```

Backing Store Element

A single backingStore element is contained within the top-level volume element. This tag is used to describe the optional copy-on-write, backing store for the storage volume. It can contain the child elements in Listing A-39.

Listing A-39. Specifying a Single Backing Store for the Storage Volume

```
  <backingStore>
    <path>/var/lib/virt/images/master.img</path>
    <format type='raw'/>
    <permissions>
      <owner>107</owner>
      <group>107</group>
      <mode>0744</mode>
      <label>virt_image_t</label>
    </permissions>
  </backingStore>
</volume>
```

Volume Examples

The following examples show some complete examples for specifying a storage volume.

Listing A-40 specifies a file path for the storage volume.

Listing A-40. Specifying a File Path for the Storage Volume

```
<pool type="dir">
  <name>virtimages</name>
  <target>
    <path>/var/lib/virt/images</path>
  </target>
</pool>
```

Listing A-41 specifies an iSCSI device for the storage volume.

Listing A-41. Specifying an iSCSI Device for the Storage Volume

```
<pool type="iscsi">
  <name>virtimages</name>
  <source>
    <host name="iscsi.example.com"/>
    <device path="iqn.2013-06.com.example:iscsi-pool"/>
    <auth type='chap' username='myuser'>
      <secret usage='libvirtiscsi'/>
    </auth>
  </source>
  <target>
    <path>/dev/disk/by-path</path>
  </target>
</pool>
```

Listing A-42 specifies an ummounted file image for the storage volume.

Listing A-42. Specifying a File Image for the Storage Volume

```
<volume>
  <name>sparse.img</name>
  <allocation>0</allocation>
  <capacity unit="T">1</capacity>
  <target>
    <path>/var/lib/virt/images/sparse.img</path>
    <permissions>
      <owner>107</owner>
      <group>107</group>
      <mode>0744</mode>
      <label>virt_image_t</label>
    </permissions>
  </target>
</volume>
```

Listing A-43 specifies an encrypted storage file for the storage volume.

Listing A-43. Specifying an Encrypted File Image for the Storage Volume

```
<volume>
  <name>MyLuks.img</name>
  <capacity unit="G">5</capacity>
  <target>
    <path>/var/lib/virt/images/MyLuks.img</path>
    <format type='raw'/>
    <encryption format='luks'>
      <secret type='passphrase' uuid='f52a81b2-424e-490c-823d-
      6bd4235bc572'/>
    </encryption>
  </target>
</volume>
```

Element and Attribute Overview

As new virtualization engine support gets added to libvirt and to handle cases like QEMU supporting a variety of emulations, a query interface has been added in 0.2.1. This query interface allows you to list the set of supported virtualization capabilities on the host.

```
connect.GetCapabilities()
```

It should be noted at this point that the following XML is returned from libvirt APIs and not passed to it. However, that does not prevent you from modifying this XML and then passing it back to libvirt to change the existing configuration.

Host Capabilities

Listing A-44 provides an example of all the possible elements of `<host/>`.

Listing A-44. Specifying the Capabilities of the Main Host System

```
<capabilities>
  <host>
    <cpu>
      <arch>x86_64</arch>
      <features>
        <vmx/>
      </features>
      <model>core2duo</model>
      <vendor>Intel</vendor>
      <topology sockets="1" cores="2" threads="1"/>
      <feature name="lahf_lm"/>
      <feature name='xtpr'/>
      ...
    </cpu>
    <power_management>
      <suspend_mem/>
      <suspend_disk/>
      <suspend_hybrid/>
    </power_management>
  </host>
</capabilities>
```

Guest Capabilities

Listing A-45 provides an example of all the possible elements of `<guest/>`.

Listing A-45. Specifying a Guest Domain's Capabilities

```
<capabilities>
  <guest>
    <os_type>hvm</os_type>
    <arch name="i686">
      <wordsize>32</wordsize>
      <domain type="xen"></domain>
      <emulator>/usr/lib/xen/bin/qemu-dm</emulator>
      <machine>pc</machine>
      <machine>isapc</machine>
      <loader>/usr/lib/xen/boot/hvmloader</loader>
    </arch>
    <features>
      <cpuselection/>
      <deviceboot/>
    </features>
  </guest>
</capabilities>
```

Node Device XML Format

There are several libvirt functions, all with the prefix `virNodeDevice`, that deal with the management of host devices that can be handed to guests via passthrough as `<hostdev>` elements in the domain XML. These devices are represented as a hierarchy, where a device on a bus has a parent of the bus controller device; the root of the hierarchy is the node named `computer`.

When represented in XML, a node device uses the top-level device element, with the elements in Listing A-46 present according to the type of device.

Listing A-46. Specifying the Node Device Capabilities

```
<device>
  <name>computer</name>
  <capability type='system'>
    <product>2241B36</product>
    <hardware>
      <vendor>LENOVO</vendor>
      <version>ThinkPad T500</version>
      <serial>R89055N</serial>
      <uuid>c9488981-5049-11cb-9c1c-993d0230b4cd</uuid>
    </hardware>
    <firmware>
      <vendor>LENOVO</vendor>
      <version>6FET82WW (3.12 )</version>
      <release_date>11/26/2009</release_date>
    </firmware>
  </capability>
</device>

<device>
  <name>net_eth1_00_27_13_6a_fe_00</name>
  <parent>pci_0000_00_19_0</parent>
  <capability type='net'>
    <interface>eth1</interface>
    <address>00:27:13:6a:fe:00</address>
    <capability type='80203'/>
  </capability>
</device>
```

```
<device>
  <name>pci_0000_02_00_0</name>
  <path>/sys/devices/pci0000:00/0000:00:04.0/0000:02:00.0</path>
  <parent>pci_0000_00_04_0</parent>
  <driver>
    <name>igb</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>2</bus>
    <slot>0</slot>
    <function>0</function>
    <product id='0x10c9'>82576 Gigabit Network Connection
    </product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <capability type='virt_functions'>
      <address domain='0x0000' bus='0x02' slot='0x10'
      function='0x0'/>
      <address domain='0x0000' bus='0x02' slot='0x10'
      function='0x2'/>
      <address domain='0x0000' bus='0x02' slot='0x10'
      function='0x4'/>
      <address domain='0x0000' bus='0x02' slot='0x10'
      function='0x6'/>
      <address domain='0x0000' bus='0x02' slot='0x11'
      function='0x0'/>
      <address domain='0x0000' bus='0x02' slot='0x11'
      function='0x2'/>
      <address domain='0x0000' bus='0x02' slot='0x11'
      function='0x4'/>
    </capability>
    <iommuGroup number='12'>
      <address domain='0x0000' bus='0x02' slot='0x00'
      function='0x0'/>
      <address domain='0x0000' bus='0x02' slot='0x00'
      function='0x1'/>
    </iommuGroup>
    <pci-express>
      <link validity='cap' port='1' speed='2.5' width='1'/>
      <link validity='sta' speed='2.5' width='1'/>
    </pci-express>
  </capability>
</device>
```

Snapshot XML Format

There are several types of snapshots.

- **Disk snapshot**: The contents of disks (whether a subset or all disks associated with the domain) are saved at a given point of time and can be restored to that state. On a running guest, a disk snapshot is likely to be only crash-consistent rather than clean. (That is, it represents the state of the disk on a sudden power outage and may need fsck or journal replays to be made consistent.) On an inactive guest, a disk snapshot is clean if the disks were clean when the guest was last shut down. Disk snapshots exist in two forms: internal (file formats such as qcow2 track both the snapshot and changes since the snapshot in a single file) and external (the snapshot is one file, and the changes since the snapshot are in another file).

- **Memory state (or VM state)**: This tracks only the state of RAM and all other resources in use by the VM. If the disks are unmodified between the time a VM state snapshot is taken and restored, then the guest will resume in a consistent state; but if the disks are modified externally in the meantime, this is likely to lead to data corruption.
- **System checkpoint**: This is a combination of disk snapshots for all disks as well as VM memory state, which can be used to resume the guest from where it left off with symptoms similar to hibernation (that is, TCP connections in the guest may have timed out, but no files or processes are lost).

libvirt can manage all three types of snapshots. For now, VM state (memory) snapshots are created only by the `save()`, `saveFlags`, and `managedSave()` methods, and they are restored via the `restore()`, `create()`, and `createWithFlags()` methods (as well as via domain autostart). With managed snapshots, libvirt tracks all information internally; with save images, the user tracks the snapshot file, but libvirt provides functions such as `saveImageGetXMLDesc()` to work with those files.

System checkpoints are created by `snapshotCreateXML()` with no flags, and disk snapshots are created by the same function with the `VIR_DOMAIN_SNAPSHOT_CREATE_DISK_ONLY` flag; in both cases, they are restored by the `revertToSnapshot()` function. For these types of snapshots, libvirt tracks each snapshot as a separate `snapshotPtr` object and maintains a tree relationship of which snapshots descended from an earlier point in time.

Attributes of libvirt snapshots are stored as child elements of the `domainsnapshot` element. At snapshot creation time, normally only the name, description, and disks elements are settable; the rest of the fields are ignored on creation and will be filled in by libvirt for informational purposes by snapshotGetXMLDesc(). However, when redefining a snapshot (since 0.9.5), with the VIR_DOMAIN_SNAPSHOT_CREATE_REDEFINE flag of snapshotCreateXML(), all of the XML described here is relevant.

Snapshots are maintained in a hierarchy. A domain can have a current snapshot, which is the most recent snapshot compared to the current state of the domain (although a domain might have snapshots without a current snapshot, if snapshots have been deleted in the meantime). Creating or reverting to a snapshot sets that snapshot as current, and the prior current snapshot is the parent of the new snapshot. Branches in the hierarchy can be formed by reverting to a snapshot with a child and then creating another snapshot.

Snapshot XML Samples

Listing A-47 uses XML to create a disk snapshot of just vda on a qemu domain with two disks.

Listing A-47. An Example of a Disk Snapshot

```
<domainsnapshot>
  <description>Snapshot of OS install and updates</description>
  <disks>
    <disk name='/path/to/old'>
      <source file='/path/to/new'/>
    </disk>
    <disk name='vdb' snapshot='no'/>
  </disks>
</domainsnapshot>
```

Listing A-48 shows the result in XML similar to this from snapshotGetXMLDesc().

Listing A-48. Sample Guest Domain Snapshot

```
<domainsnapshot>
  <name>1270477159</name>
  <description>Snapshot of OS install and updates</description>
  <state>running</state>
  <creationTime>1270477159</creationTime>
  <parent>
    <name>bare-os-install</name>
  </parent>
  <memory snapshot='no'/>
  <disks>
    <disk name='vda' snapshot='external'>
      <driver type='qcow2'/>
      <source file='/path/to/new'/>
    </disk>
    <disk name='vdb' snapshot='no'/>
  </disks>
  <domain>
    <name>fedora</name>
    <uuid>93a5c045-6457-2c09-e56c-927cdf34e178</uuid>
    <memory>1048576</memory>
    ...
    <devices>
      <disk type='file' device='disk'>
        <driver name='qemu' type='raw'/>
        <source file='/path/to/old'/>
        <target dev='vda' bus='virtio'/>
      </disk>
      <disk type='file' device='disk' snapshot='external'>
        <driver name='qemu' type='raw'/>
        <source file='/path/to/old2'/>
        <target dev='vdb' bus='virtio'/>
      </disk>
      ...
    </devices>
  </domain>
</domainsnapshot>
```

With that snapshot created, `/path/to/old` is the read-only backing file to the new active file `/path/to/new`. The `<domain>` element within the snapshot `xml` records the state of the domain just before the snapshot; a call to `getXMLDesc()` will show that the domain has been changed to reflect the snapshot (Listing A-49).

Listing A-49. Sample XML from the getXMLDesc() Method

```
<domain>
  <name>fedora</name>
  <uuid>93a5c045-6457-2c09-e56c-927cdf34e178</uuid>
  <memory>1048576</memory>
  ...
  <devices>
    <disk type='file' device='disk'>
      <driver name='qemu' type='qcow2'/>
      <source file='/path/to/new'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <disk type='file' device='disk' snapshot='external'>
      <driver name='qemu' type='raw'/>
      <source file='/path/to/old2'/>
      <target dev='vdb' bus='virtio'/>
    </disk>
    ...
  </devices>
</domain>
```

The Domain Capabilities XML Format

Sometimes, when a new domain is to be created, it may become handy to know the capabilities of the hypervisor so the correct combination of devices and drivers is used. For example, when a management application is considering the mode for a host device's passthrough, there are several options depending not only on host but on the hypervisor in question. If the hypervisor is QEMU, then it needs to be more recent to support VFIO, while legacy KVM is achievable just fine with older qemus.

The main difference between the connection method `getCapabilities` and the emulator capabilities API is that the former one focuses more on the host capabilities (e.g., NUMA topology, security models in effect, etc.), while the latter one specializes on the hypervisor capabilities.

While the Driver Capabilities section provides the host capabilities (e.g., NUMA topology, security models in effect, etc.), the "Domain Capabilities" section provides the hypervisor-specific capabilities for management applications to query and make decisions regarding what to utilize.

The Domain Capabilities section can provide information such as the correct combination of devices and drivers that are supported. Knowing which host and hypervisor-specific options are available or supported will allow the management application to choose an appropriate mode for a passthrough host device as well as which adapter to utilize.

Element and Attribute Overview

Listing A-50 shows the new query interface that was added to the `virConnect` APIs to retrieve the XML listing of the set of domain capabilities (since 1.2.7).

Listing A-50. The Method for Retieving Domain Capabilities

```
getDomainCapabilities()
```

The root element that the emulator capability XML document starts with has the name `<domainCapabilities>`. It contains at least four direct child elements (Listing A-51).

Listing A-51. A Sample Domain Capabilities XML Output

```
<domainCapabilities>
  <path>/usr/bin/qemu-system-x86_64</path>
  <domain>kvm</domain>
  <machine>pc-i440fx-2.1</machine>
  <arch>x86_64</arch>
  ...
</domainCapabilities>
```

CPU Allocation

Before any device's capability occurs, there might be information on domain-wide capabilities, e.g., virtual CPUs (Listing A-52).

Listing A-52. Retrieving CPU Capabilities

```
<domainCapabilities>
  ...
  <vcpu max='255'/>
  ...
</domainCapabilities>
```

BIOS Bootloader

Sometimes users might want to tweak some BIOS knobs or use UEFI. For cases like that, the `os` element exposes what values can be passed to its children (Listing A-53) .

Listing A-53. Retrieving BIOS or UEFI Capabilities from a Guest Domain

```
<domainCapabilities>
  ...
  <os supported='yes'>
    <loader supported='yes'>
      <value>/usr/share/OVMF/OVMF_CODE.fd</value>
      <enum name='type'>
        <value>rom</value>
        <value>pflash</value>
      </enum>
      <enum name='readonly'>
        <value>yes</value>
        <value>no</value>
      </enum>
    </loader>
  </os>
  ...
<domainCapabilities>
```

CPU Configuration

Each CPU mode understood by libvirt is described with a mode element, which tells whether the particular mode is supported and provides (when applicable) more details about it (Listing A-54).

Listing A-54. Retrieving CPU Configuration of a Guest Domain

```
<domainCapabilities>
  ...
  <cpu>
    <mode name='host-passthrough' supported='yes'/>
    <mode name='host-model' supported='yes'>
      <model fallback='allow'>Broadwell</model>
      <vendor>Intel</vendor>
      <feature policy='disable' name='aes'/>
      <feature policy='require' name='vmx'/>
    </mode>
    <mode name='custom' supported='yes'>
      <model usable='no'>Broadwell</model>
      <model usable='yes'>Broadwell-noTSX</model>
      <model usable='no'>Haswell</model>
      ...
    </mode>
  </cpu>
  ...
<domainCapabilities>
```

IO Threads

The `<iothread>` elements indicates whether I/O threads are supported (Listing A-55).

Listing A-55. Retrieving iothread Capabilities for a Guest Domain

```
<domainCapabilities>
  ...
  <iothread supported='yes'/>
  ...
<domainCapabilities>
```

Reported capabilities are expressed as an enumerated list of available options for each element or attribute. For example, the `<disk/>` element has an attribute device that can support the values `<disk>`, `<cdrom>`, `<floppy>`, and `<lun>`.

Hard Drives, Floppy Disks, and CD-ROMs

Disk capabilities are exposed under the disk element, as shown in Listing A-56.

Listing A-56. Retrieving Drive Capabilities from a Guest Domain

```
<domainCapabilities>
  ...
  <devices>
    <disk supported='yes'>
      <enum name='diskDevice'>
        <value>disk</value>
        <value>cdrom</value>
        <value>floppy</value>
        <value>lun</value>
      </enum>
      <enum name='bus'>
        <value>ide</value>
        <value>fdc</value>
        <value>scsi</value>
        <value>virtio</value>
        <value>xen</value>
        <value>usb</value>
        <value>sata</value>
        <value>sd</value>
      </enum>
    </disk>
    ...
  </devices><domainCapabilities>
```

Graphical Framebuffers

Graphics device capabilities are exposed under the `graphics` element (Listing A-57).

Listing A-57. Retriving Graphical Device Capabilities from a Guest Domain

```
<domainCapabilities>
  ...
  <devices>
    <graphics supported='yes'>
      <enum name='type'>
        <value>sdl</value>
        <value>vnc</value>
        <value>spice</value>
      </enum>
    </graphics>
    ...
  </devices>  </devices><domainCapabilities>
```

Video Device

Video device capabilities are exposed under the `video` element (Listing A-58).

Listing A-58. Retrieving Video Capabilities from a Guest Domain

```
<domainCapabilities>
  ...
  <devices>
    <video supported='yes'>
      <enum name='modelType'>
        <value>vga</value>
        <value>cirrus</value>
        <value>vmvga</value>
        <value>qxl</value>
        <value>virtio</value>
      </enum>
    </video>
    ...
  </devices>
<domainCapabilities>
```

Host Device Assignment

Some host devices can be passed through to a guest (e.g., USB, PCI, and SCSI), but only if the code in Listing A-59 is enabled.

Listing A-59. Retrieving Host Assignments to the Guest Domain

```
<domainCapabilities>
  ...
  <devices>
    <hostdev supported='yes'>
      <enum name='mode'>
        <value>subsystem</value>
        <value>capabilities</value>
      </enum>
      <enum name='startupPolicy'>
        <value>default</value>
        <value>mandatory</value>
        <value>requisite</value>
        <value>optional</value>
      </enum>
      <enum name='subsysType'>
        <value>usb</value>
        <value>pci</value>
        <value>scsi</value>
      </enum>
      <enum name='capsType'>
        <value>storage</value>
        <value>misc</value>
        <value>net</value>
      </enum>
      <enum name='pciBackend'>
        <value>default</value>
        <value>kvm</value>
        <value>vfio</value>
        <value>xen</value>
      </enum>
    </hostdev>
  </devices>
<domainCapabilities>
```

Features

One more set of XML elements describes the supported features and their capabilities. All `<features>` elements occur as children of the main `features` element (Listing A-60).

Listing A-60. Retrieving Supported Features from the Guest Domain

```
<domainCapabilities>
  ...
  <features>
    <gic supported='yes'>
      <enum name='version'>
        <value>2</value>
        <value>3</value>
      </enum>
    </gic>
    <vmcoreinfo supported='yes'/>
    <genid supported='yes'/>
    <sev>
      <cbitpos>47</cbitpos>
      <reduced-phys-bits>1</reduced-phys-bits>
    </sev>
  </features>
<domainCapabilities>
```

Reported capabilities are expressed as an enumerated list of possible values for each element or attribute. For example, the `<gic>` element has an attribute version that can support the value 2 or 3.

For information about the purpose of each feature, see the relevant section in the domain XML documentation.

Network Filters

This section provides an introduction to libvirt's network filters, including their goals, concepts, and XML format.

Goals and Background

The goal of the network filtering XML is to enable administrators of a virtualized system to configure and enforce network traffic filtering rules on virtual machines and manage the parameters of network traffic that virtual machines are allowed to send or receive. The network traffic filtering rules are applied on the host when a virtual machine is started. Since the filtering rules cannot be circumvented from within the virtual machine, it makes them mandatory from the point of view of a virtual machine user.

The network filter subsystem allows each virtual machine's network traffic filtering rules to be configured individually on a per-interface basis. The rules are applied on the host when the virtual machine is started and can be modified while the virtual machine is running. The latter can be achieved by modifying the XML description of a network filter.

Multiple virtual machines can make use of the same generic network filter. When such a filter is modified, the network traffic filtering rules of all running virtual machines that reference this filter are updated.

Network filtering support has been available since version 0.8.1 (QEMU, KVM).

Concepts

The network traffic filtering subsystem enables configuration of network traffic filtering rules on individual network interfaces that are configured for certain types of network configurations. Supported network types are as follows:

- Network
- Ethernet (must be used in bridging mode)
- Bridge

The interface XML is used to reference a top-level filter. In Listing A-61, the interface description references the filter `clean-traffic`.

Listing A-61. Specifying Network Filters

```
...
<devices>
  <interface type='bridge'>
    <mac address='00:16:3e:5d:c7:9e'/>
    <filterref filter='clean-traffic'/>
  </interface>
</devices>
...
```

Network filters are written in XML and may either contain references to other filters, contain rules for traffic filtering, or hold a combination of both. The referenced filter `clean-traffic` is a filter that only contains references to other filters and no actual filtering rules. Since references to other filters can be used, a tree of filters can be built. The `clean-traffic` filter can be viewed using the command `virsh nwfilter-dumpxml clean-traffic`.

As previously mentioned, a single network filter can be referenced by multiple virtual machines. Since interfaces will typically have individual parameters associated with their respective traffic filtering rules, the rules described in a filter XML can be parameterized with variables. In this case, the variable name is used in the filter XML, and the name and value are provided at the place where the filter is referenced. In Listing A-62, the interface description has been extended with the parameter IP and a dotted IP address as the value.

Listing A-62. Specifying IP-Specific Network Filters

```
...
<devices>
  <interface type='bridge'>
    <mac address='00:16:3e:5d:c7:9e'/>
    <filterref filter='clean-traffic'>
      <parameter name='IP' value='10.0.0.1'/>
    </filterref>
  </interface>
</devices>
...
```

In this particular example, the `clean-traffic` network traffic filter will be instantiated with the IP address parameter 10.0.0.1 and enforce that the traffic from this interface will always be using 10.0.0.1 as the source IP address, which is one of the purposes of this particular filter.

Filtering Chains

Filtering rules are organized in filter chains. These chains can be thought of as having a tree structure with packet filtering rules as entries in individual chains (branches).

Packets start their filter evaluation in the root chain and can then continue their evaluation in other chains, return from those chains into the root chain, or be dropped or accepted by a filtering rule in one of the traversed chains.

libvirt's network filtering system automatically creates individual root chains for every virtual machine's network interface on which the user chooses to activate traffic filtering. The user may write filtering rules that either are directly instantiated in the root chain or may create protocol-specific filtering chains for efficient evaluation of protocol-specific rules. The following chains exist:

- `root`
- `mac` (since 0.9.8)
- `stp` (Spanning Tree Protocol) (since 0.9.8)
- `vlan` (802.1Q) (since 0.9.8)
- `arp`, `rarp`
- `ipv4`
- `ipv6`

Since 0.9.8, multiple chains evaluating the `mac`, `stp`, `vlan`, `arp`, `rarp`, `ipv4`, and `ipv6` protocols can be created using the protocol name only as a prefix in the chain's name. This, for example, allows chains with the names arp-xyz or arp-test to be specified and have the ARP protocol packets evaluated in those chains.

The filter in Listing A-63 shows an example of filtering ARP traffic in the arp chain.

Listing A-63. Specifying Filtering ARP Traffic in the arp Chain

```
<filter name='no-arp-spoofing' chain='arp' priority='-500'>
  <uuid>f88f1932-debf-4aa1-9fbe-f10d3aa4bc95</uuid>
  <rule action='drop' direction='out' priority='300'>
    <mac match='no' srcmacaddr='$MAC'/>
  </rule>
  <rule action='drop' direction='out' priority='350'>
    <arp match='no' arpsrcmacaddr='$MAC'/>
  </rule>
  <rule action='drop' direction='out' priority='400'>
    <arp match='no' arpsrcipaddr='$IP'/>
  </rule>
  <rule action='drop' direction='in' priority='450'>
    <arp opcode='Reply'/>
    <arp match='no' arpdstmacaddr='$MAC'/>
  </rule>
  <rule action='drop' direction='in' priority='500'>
    <arp match='no' arpdstipaddr='$IP'/>
  </rule>
  <rule action='accept' direction='inout' priority='600'>
    <arp opcode='Request'/>
  </rule>
  <rule action='accept' direction='inout' priority='650'>
    <arp opcode='Reply'/>
  </rule>
  <rule action='drop' direction='inout' priority='1000'/>
</filter>
```

The consequence of putting ARP-specific rules in the `arp` chain, rather than, for example, in the `root` chain, is that packets for any other protocol than ARP do not need to be evaluated by ARP-specific rules. This improves the efficiency of the traffic filtering. However, one must then pay attention to only put filtering rules for the given protocol into the chain since any other rules will not be evaluated; in other words, an IPv4 rule will not be evaluated in the `arp` chain since no IPv4 protocol packets will traverse the `arp` chain.

Filtering Chain Priorities

All chains are connected to the root chain. The order in which those chains are accessed is influenced by the priority of the chain. Table A-1 shows the chains that can be assigned a priority and their default priorities.

Table A-1. *Filtering Chain Priorities*

Chain (Prefix)	Default Priority
`stp`	-810
`mac`	-800
`stp`	-810
`vlan`	-750
`ipv4`	-700
`ipv6`	-600
`arp`	-500
`rarp`	-400

A chain with a lower-priority value is accessed before one with a higher value. Since 0.9.8, the previously listed chains can be assigned custom priorities by writing a value in the range `[-1000, 1000]` into the priority (XML) attribute in the filter node. The previous example filter shows the default priority of -500 for `arp` chains.

Usage of Variables in Filters

Two variables names have so far been reserved for usage by the network traffic filtering subsystem: `MAC` and `IP`.

`MAC` is the MAC address of the network interface. A filtering rule that references this variable will automatically be instantiated with the MAC address of the interface. This works without the user having to explicitly provide the `MAC` parameter. Even though it is possible to specify the `MAC` parameter similar to the `IP` parameter shown previously, it is discouraged since libvirt knows what MAC address an interface will be using.

The parameter `IP` represents the IP address that the operating system inside the virtual machine is expected to use on the given interface. The `IP` parameter is special in so far as the libvirt daemon will try to determine the IP address (and thus the `IP` parameter's value) that is being used on an interface if the parameter is not explicitly provided but referenced. For current limitations on IP address detection, consult the Chapter 7.

The previously shown network filter `no-arp-spoofing` is an example of a network filter with XML referencing the `MAC` and `IP` variables.

Note that referenced variables are always prefixed with the dollar (`$`) sign. The format of the value of a variable must be of the type expected by the filter attribute in the XML. In the previous example, the `IP` parameter must hold a dotted IP address in decimal number format. Failure to provide the correct value type will result in the filter not being instantiatable and will prevent a virtual machine from starting or the interface from attaching when hot plugging is used. The types that are expected for each XML attribute are shown in Listing A-64.

Since version 0.9.8, variables can contain lists of elements; for example, the variable `IP` can contain multiple IP addresses that are valid on a particular interface. Listing A-64 shows the notation for providing multiple elements for the `IP` variable.

Listing A-64. Specifying an IP Filter for Specific Network Addresses

```
...
<devices>
  <interface type='bridge'>
    <mac address='00:16:3e:5d:c7:9e'/>
    <filterref filter='clean-traffic'>
      <parameter name='IP' value='10.0.0.1'/>
      <parameter name='IP' value='10.0.0.2'/>
      <parameter name='IP' value='10.0.0.3'/>
    </filterref>
  </interface>
</devices>
...
```

This then allows filters to enable multiple IP addresses per interface. Therefore, with the list of IP addresses shown previously, the rule in Listing A-65 will create three individual filtering rules, one for each IP address.

Listing A-65. Specifying an IP Filter for Multiple IP Addresses

```
...
<rule action='accept' direction='in' priority='500'>
  <tcp srpipaddr='$IP'/>
</rule>
...
```

Since version 0.9.10, it is possible to access individual elements of a variable holding a list of elements. A filtering rule like that in Listing A-66 accesses the second element of the variable `DSTPORTS`.

Listing A-66. Specifying an IP Filter for Specific Ports

```
...
<rule action='accept' direction='in' priority='500'>
  <udp dstportstart='$DSTPORTS[1]'/>
</rule>
...
```

Since 0.9.10, it is possible to create filtering rules that instantiate all combinations of rules from different lists using the notation of `$VARIABLE[@<iterator ID>]`. The rule in Listing A-67 allows a virtual machine to receive traffic on a set of ports, which are specified in `DSTPORTS`, from the set of source IP address specified in `SRCIPADDRESSES`. The rule generates all combinations of elements of the variable `DSTPORT` with those of `SRCIPADDRESSES` by using two independent iterators to access their elements.

Listing A-67. Specify a Filter for a Range of Ports

```
...
<rule action='accept' direction='in' priority='500'>
  <ip srcipaddr='$SRCIPADDRESSES[@1]'
dstportstart='$DSTPORTS[@2]'/>
</rule>
...
```

Listing A-68 assigns concrete values to `SRCIPADDRESSES` and `DSTPORTS`.

Listing A-68. Specifying a Filter for Concrete Values

```
SRCIPADDRESSES = [ 10.0.0.1, 11.1.2.3 ]
DSTPORTS = [ 80, 8080 ]
```

Accessing the variables using `$SRCIPADDRESSES[@1]` and `$DSTPORTS[@2]` would then result in all combinations of addresses and ports being created (Listing A-69).

Listing A-69. Result of Specifying a Filter for a Range of Concrete Addresses

```
10.0.0.1, 80
10.0.0.1, 8080
11.1.2.3, 80
11.1.2.3, 8080
```

Accessing the same variables using a single iterator, for example by using the notation `$SRCIPADDRESSES[@1]` and `$DSTPORTS[@1]`, would result in parallel access to both lists and result in the combinations shown in Listing A-70.

Listing A-70. Result of Specifying a Filter for a Range of Concrete Addresses

```
10.0.0.1, 80
11.1.2.3, 8080
```

Further, the notation of `$VARIABLE` is shorthand for `$VARIABLE[@0]`. The former notation always assumes an iterator with an ID of 0.

Automatic IP Address Detection

The detection of IP addresses used on a virtual machine's interface is automatically activated if the variable `IP` is referenced but no value has been assigned to it. Since 0.9.13, the variable `CTRL_IP_LEARNING` can be used to specify the IP address learning method to use. Valid values are `any`, `dhcp`, `and none`.

The value `any` means that libvirt may use any packet to determine the address in use by a virtual machine, which is the default behavior if the variable `CTRL_IP_LEARNING` is not set. This method will detect only a single IP address on an interface. Once a VM's IP address has been detected, its IP network traffic will be locked to that address, if for example IP address spoofing is prevented by one of its filters. In that case, the user of the VM will not be able to change the IP address on the interface inside the VM, which would be considered IP address spoofing. When a VM is migrated to another host or resumed after a suspend operation, the first packet sent by the VM will again determine the IP address it can use on a particular interface.

A value of `dhcp` specifies that libvirt should honor only DHCP server-assigned addresses with valid leases. This method supports the detection and usage of multiple IP addresses per interface. When a VM is resumed after a suspend operation, still-valid IP address leases are applied to its filters. Otherwise, the VM is expected to again use DHCP to obtain new IP addresses. The migration of a VM to another physical host requires that the VM again runs the DHCP protocol.

Using `CTRL_IP_LEARNING=dhcp` (DHCP snooping) provides additional antispoofing security, especially when combined with a filter allowing only trusted DHCP servers to assign addresses. To enable this, set the variable `DHCPSERVER` to the IP address of a valid DHCP server and provide filters that use this variable to filter incoming DHCP responses.

When DHCP snooping is enabled and the DHCP lease expires, the VM will no longer be able to use the IP address until it acquires a new, valid lease from a DHCP server. If the VM is migrated, it must get a new valid DHCP lease to use an IP address (e.g., by bringing the VM interface down and up again).

Note that automatic DHCP detection listens to the DHCP traffic the VM exchanges with the DHCP server of the infrastructure. To avoid denial-of-service attacks on libvirt, the evaluation of those packets is rate-limited, meaning that a VM sending an excessive number of DHCP packets per second on an interface will not have all of those packets evaluated, and thus filters may not get adapted. Normal DHCP client behavior is assumed to send a low number of DHCP packets per second. Further, it is important to set up appropriate filters on all VMs in the infrastructure to avoid them being able to send DHCP packets. Therefore, either VMs must be prevented from sending UDP and TCP traffic from port 67 to port 68 or the `DHCPSERVER` variable should be used on all VMs to restrict DHCP server messages to be allowed to originate only from trusted DHCP servers. At the same time, antispoofing prevention must be enabled on all VMs in the subnet.

If `CTRL_IP_LEARNING` is set to `none`, libvirt does not do IP address learning, and referencing `IP` without assigning it an explicit value is an error.

The XML in Listing A-71 provides an example for the activation of IP address learning using the DHCP snooping method.

Listing A-71. Activating a Specific IP Address

```
<interface type='bridge'>
  <source bridge='virbr0'/>
  <filterref filter='clean-traffic'>
    <parameter name='CTRL_IP_LEARNING' value='dhcp'/>
  </filterref>
</interface>
```

Reserved Variables

Table A-2 lists reserved variables in use by libvirt.

Table A-2. *Libvirt Reserved Variables*

Variable Name	Semantics
`MAC`	The MAC address of the interface
`IP`	The list of IP addresses in use by an interface
`IPV6`	Not currently implemented: the list of IPV6 addresses in use by an interface
`DHCPSERVER`	The list of IP addresses of trusted DHCP servers
`DHCPSERVER6`	Not currently implemented: the list of IPv6 addresses of trusted DHCP servers
`CTRL_IP_LEARNING`	The choice of the IP address detection mode

Element and Attribute Overview

The root element required for all network filters is named `filter` with two possible attributes. The name attribute provides a unique name of the given filter. The `chain` attribute is optional but allows certain filters to be better organized for more efficient processing by the firewall subsystem of the underlying host. Currently the system supports only the chains `root`, `ipv4`, `ipv6`, `arp`, and `rarp`.

References to Other Filters

Any filter may hold references to other filters. Individual filters may be referenced multiple times in a filter tree, but references between filters must not introduce loops (directed acyclic graph).

Listing A-72 shows the XML of the `clean-traffic` network filter referencing several other filters.

Listing A-72. Specifying a clean-traffic Filter Referencing Other Filters

```
<filter name='clean-traffic'>
  <uuid>6ef53069-ba34-94a0-d33d-17751b9b8cb1</uuid>
  <filterref filter='no-mac-spoofing'/>
  <filterref filter='no-ip-spoofing'/>
  <filterref filter='allow-incoming-ipv4'/>
  <filterref filter='no-arp-spoofing'/>
  <filterref filter='no-other-l2-traffic'/>
  <filterref filter='qemu-announce-self'/>
</filter>
```

To reference another filter, the XML node `filterref` needs to be provided inside a filter node. This node must have the attribute filter whose value contains the name of the filter to be referenced.

New network filters can be defined at any time and may contain references to network filters that are not known to libvirt yet. However, once a virtual machine is started or a network interface referencing a filter is to be hot plugged, all network filters in the filter tree must be available. Otherwise, the virtual machine will not start or the network interface cannot be attached.

Filter Rules

Listing A-73 shows a simple example of a network traffic filter implementing a rule to drop traffic if the IP address (provided through the value of the variable `IP`) in an outgoing IP packet is not the expected one, thus preventing IP address spoofing by the VM.

Listing A-73. Using Filter Rules

```
<filter name='no-ip-spoofing' chain='ipv4'>
  <uuid>fce8ae33-e69e-83bf-262e-30786c1f8072</uuid>
  <rule action='drop' direction='out' priority='500'>
    <ip match='no' srcipaddr='$IP'/>
  </rule>
</filter>
```

A traffic filtering rule starts with the rule node. This node may contain up to three attributes.

- `action`: Mandatory; must be `drop` (matching the rule silently discards the packet with no further analysis), `reject` (matching the rule generates an ICMP reject message with no further analysis) since 0.9.0, `accept` (matching the rule accepts the packet with no further analysis), `return` (matching the rule passes this filter but returns control to the calling filter for further analysis) since 0.9.7, or `continue` (matching the rule goes on to the next rule for further analysis) since 0.9.7.
- `direction`: Mandatory; must be either `in`, `out`, or `inout` if the rule is for incoming, outgoing, or incoming and outgoing traffic.
- `priority`: Optional; the priority of the rule controls the order in which the rule will be instantiated relative to other rules. Rules with lower value will be instantiated before rules with higher values. Valid values are in the range of 0 to 1000. Since 0.9.8, this has been extended to cover the range of -1000 to 1000. If this attribute is not provided, priority 500 will automatically be assigned. Note that filtering rules in the root chain are

sorted with filters connected to the root chain following their priorities. This allows you to interleave filtering rules with access to filter chains. (See also the "Filtering Chain Priorities" section.)

- `statematch`: Optional; possible values are 0 or `false` to turn the underlying connection state matching off; the default is `true`. Also read the "Network Filters" section.

The previous example indicates that the traffic of type `ip` will be associated with the chain `ipv4`, and the rule will have priority 500. If for example another filter is referenced whose traffic of type `ip` is also associated with the chain `ipv4`, then that filter's rules will be ordered relative to the priority 500 of the shown rule.

A rule may contain a single rule for the filtering of traffic. The previous example shows that traffic of type `ip` is to be filtered.

Supported Protocols

The following sections list the protocols that are supported by the network filtering subsystem. The type of traffic a rule is supposed to filter on is provided in the rule node as a nested node. Depending on the traffic type that a rule is filtering, the attributes are different. The previous example showed the single attribute `srcipaddr` that is valid inside the IP traffic filtering node. The following sections show what attributes are valid and what type of data they are expecting. The following datatypes are available:

- `UINT8`: 8-bit integer; range 0–255.
- `UINT16`: 16-bit integer; range 0–65535.
- `MAC_ADDR`: MAC address in dotted decimal format, i.e., 00:11:22:33:44:55.
- `MAC_MASK`: MAC address mask in MAC address format, i.e., FF:FF:FF:FC:00:00.

- IP_ADDR: IP address in dotted decimal format, i.e., 10.1.2.3.
- IP_MASK: IP address mask in either dotted decimal format (255.255.248.0) or CIDR mask (0–32).
- IPV6_ADDR: IPv6 address in numbers format, i.e., FFFF::1.
- IPV6_MASK: IPv6 mask in numbers format (FFFF:FFFF:FC00::) or CIDR mask (0–128).
- STRING: A string.
- BOOLEAN: true, yes, 1 or false, no, 0'5.
- IPSETFLAGS: The source and destination flags of the ipset described by up to six src or dst elements selecting features from either the source or the destination part of the packet header. Here's an example: src,src,dst. The number of selectors to provide here depends on the type of ipset that is referenced.

Every attribute except for those of type IP_MASK or IPV6_MASK can be negated using the match attribute with the value no. Multiple negated attributes may be grouped together. The XML fragment in Listing A-74 shows such an example using abstract attributes.

Listing A-74. Negating Attributes

```
...
<rule action='drop' direction='in'>
  <protocol match='no' attribute1='value1'
attribute2='value2'/>
  <protocol attribute3='value3'/>
</rule>
...
```

Rules perform a logical AND evaluation on all values of the given protocol attributes. Thus, if a single attribute's value does not match the one given in the rule, the whole rule will be skipped during evaluation. Therefore, in the previous example, incoming traffic will be dropped only if the protocol property `attribute1` does not match `value1` and the protocol property `attribute2` does not match `value2` and the protocol property `attribute3` matches `value3`.

MAC (Ethernet)

Protocol ID: `mac` (Table A-3)

Table A-3. *mac Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcmacmask`	`MAC_MASK`	Mask applied to MAC address of sender
`dstmacaddr`	`MAC_ADDR`	MAC address of destination
`dstmacmask`	`MAC_MASK`	Mask applied to MAC address of destination
`protocolid`	`MAC_ADDR`	MAC address of sender
`MAC`	`UINT16` (`0x600-0xffff`), `STRING`	`UINT16` (`0x600-0xffff`), `STRING`
`comment` (since 0.8.5)	`STRING`	Text with a maximum of 256 characters

Note: Rules of this type should go into the `root` chain.

Listing A-75 shows that the valid strings for `protocolid` are `arp`, `rarp`, `ipv4`, and `ipv6`.

Listing A-75. Valid Strings for protocolid

```
...
<rule action='drop' direction='in'>
  <protocol match='no' attribute1='value1'
attribute2='value2'/>
  <protocol attribute3='value3'/>
</rule>
...
```

VLAN (802.1Q) (Since 0.9.8)

Protocol ID: vlan (Table A-4)

Table A-4. *vlan Variables and Their Expected Values*

Variable Name	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
vlanid	UINT16 (0x0-0xfff, 0–4095)	VLAD ID
encap-protocol	UINT16 (0x03c-0xfff), String	Encapsulated layer 3 protocol ID
comment (since 0.8.5)	STRING	Text with a maximum of 256 characters

Note: Rules of this type should go into either the `vlan` or `root` chain. Valid strings for `encap-protocol` are `arp`, `ipv4`, and `ipv6`.

STP (Spanning Tree Protocol) (Since 0.9.8)

Protocol ID: `stp` (Table A-5)

Table A-5. *stp Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcmacmask`	`MAC_MASK`	Mask applied to MAC address of sender
`type`	`UINT8`	Bridge Protocol Data Unit (BPDU) type
`flags`	`UINT8`	BPDU flag
`root-priorityvlanid`	`UINT16`	Root priority (range start)
`root-priority-hi`	`UINT16`	Root priority (range end)
`root-address`	`MAC_ADDRESS`	Root MAC address
`root-address-mask`	`MAC_MASK`	Root MAC address mask
`root-cost`	`UINT32`	Root path cost (range start)
`root-cost-hi`	`UINT32`	Root path cost range end
`sender-priority`	`UINT16`	Sender priority (range start)
`sender-priority-hi`	`UINT16`	Sender priority range end
`sender-address`	`MAC_ADDRESS6`	BPDU sender MAC address
`sender-address-mask`	`MAC_MASK`	BPDU sender MAC address mask
`port`	`UIMT16`	Port identifier (range start)

(*continued*)

Table A-5. (*continued*)

Variable Name	Datatype	Semantics
`port-hi`	`UIMT16`	Port identifier range end)
`msg-age`	`UIMT16`	Message age timer (range start)
`msg-age-hi`	`UIMT16`	Message age timer range end
`max-age`	`UIMT16`	Maximum age timer (range start)
`max-age-hi`	`UIMT16`	Maximum age timer range end
`hello-time`	`UIMT16`	Hello time timer (range start)
`hello-time-hi`	`UIMT16`	Hello time timer range end
`forward-delay`	`UIMT16`	Forward delay (range start)
`forward-delay`	`UIMT16`	Forward delay range end
`forward-delay-hi`	`UIMT16`	Forward delay range end
`comment`	`STRING`	Text with maximum of 256 characters

Note: Rules of this type should go into either the `vlan` or the `root` chain.

ARP/RARP

Protocol ID: `apr` or `rarp` (Table A-6)

***Table A-6.** apr or rarp Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcmacmask`	`MAC_MASK`	Mask applied to MAC address of sender
`dstmacaddr`	`MAC_ADDR`	MAC address of destination
`dstmacmask`	`MAC_MASK`	Mask applied to MAC address of destination
`hwtype`	`UINT16`	Hardware type
`protocoltype`	`UINT16`	Protocol type
`opcode`	`UINT16`, `STRING`	Opcode
`arpsrcmacaddr`	`MAC_ADDR`	Source MAC address in ARP/RARP packet
`arpdstmacaddr`	`MAC_ADDR`	Destination MAC address in ARP/RARP packet
`arpsrcipaddr`	`IP_ADDR`	Source IP address in ARP/RARP packet
`arpsrcipmask` (Since 1.2.3)	`IP_MASK`	Source IP mask
`arpdstipaddr`	`IP_ADDR`	Destination IP address in ARP/RARP packet
`arpdstipmask` (Since 1.2.3)	`IP_MASK`	Destination IP mask
`comment` (Since 0.8.5)	`STRING`	Text with a maximum 256 characters
`gratuitous` (Since 0.9.2)	`BOOLEAN`	Boolean indicating whether to check for gratuitous ARP packet

Note: Rules of this type should go into either the `root` or the `arp/rarp` chain.

Valid strings for the `Opcode` field are `Request`, `Reply`, `Request_Reverse`, `Reply_Reverse`, `DRARP_Request`, `DRARP_Reply`, `DRARP_Error`, `InARP_Request`, and `ARP_NAK`.

IPv4

Protocol ID: `ip` (Table A-7)

Table A-7. *ip Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcmacmask`	`MAC_MASK`	Mask applied to MAC address of sender
`dstmacaddr`	`MAC_ADDR`	MAC address of destination
`dstmacmask`	`MAC_MASK`	Mask applied to MAC address of destination
`srcipaddr`	`IP_ADDR`	Source IP address
`srcipmask`	`IP_MASK`	Mask applied to source IP address
`dstipaddr`	`IP_ADDR`	Destination IP address
`dstipmask`	`IP_MASK`	Mask applied to destination IP address
`protocol`	`UINT8`, `STRING`	Layer 4 protocol identifier
`srcportstart`	`UINT16`	Start of range of valid source ports; requires protocol
`srcportend`	`UINT16`	End of range of valid source ports; requires protocol

(*continued*)

Table A-7. (*continued*)

Variable Name	Datatype	Semantics
`dstportstart`	`UINT16`	Start of range of valid destination ports; requires protocol
`dstportend`	`UINT16`	End of range of valid destination ports; requires protocol
`dscp`	`UINT8` (0x0-0x3f, 0 - 63)	Differentiated Services Code Point
`comment` (Since 0.8.5)	`STRING`	Text with a maximum of 256 characters

Note: Rules of this type should go into either the `root` or the `ipv4` chain.

Valid strings for `protocol` are `tcp`, `udp`, `udplite`, `esp`, `ah`, `icmp`, `igmp`, and `sctp`.

IPv6

Protocol ID: `ipv6` (Table A-8)

Table A-8. ipv6 Variables and Their Expected Values

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcmacmask`	`MAC_MASK`	Mask applied to MAC address of sender
`dstmacaddr`	`MAC_ADDR`	MAC address of destination
`dstmacmask`	`MAC_MASK`	Mask applied to MAC address of destination
`srcipaddr`	`IPV6_ADDR`	Source IP address

(*continued*)

Table A-8. (*continued*)

Variable Name	Datatype	Semantics
`srcipmask`	`IPV6_MASK`	Mask applied to source IP address
`dstipaddr`	`IPV6_ADDR`	Destination IP address
`dstipmask`	`IPV6_MASK`	Mask applied to destination IP address
`protocol`	`UINT8`	Layer 4 protocol identifier
`srcportstart`	`UINT16`	Start of range of valid source ports; requires protocol
`srcportend`	`UINT16`	End of range of valid source ports; requires protocol
`dstportstart`	`UINT16`	Start of range of valid destination ports; requires protocol
`dstportend`	`UINT16`	End of range of valid destination ports; requires protocol
`type` (Since 1.2.12)	`UINT8`	Differentiated Services Code Point
`typeend` (Since 1.2.12)	`UINT8`	ICMPv6 type end of range; requires protocol to be set to icmpv6
`code` (Since 1.2.12)	`UINT8`	ICMPv6 code; requires protocol to be set to icmpv6
`codeend` (Since 1.2.12)	`UINT8`	ICMPv6 code end of range; requires protocol to be set to icmpv6
`comment` (Since 0.8.5)	`STRING`	Text with a maximum of 256 characters

Note: Rules of this type should go into either the `root` or the `ipv6` chain.

Valid strings for `protocol` are `tcp`, `udp`, `udplite`, `esp`, `ah`, `icmpv6`, and `sctp`.

TCP/UDP/SCTP

Protocol ID: `tcp`, `udp`, `sctp` (Table A-9)

Table A-9. *tcp, udp, sctp Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcipaddr`	`IP_ADDR`	Source IP address
`srcipmask`	`IP_MASK`	Mask applied to source IP address
`dstipaddr`	`IP_ADDR`	Destination IP address
`dstipmask`	`IP_MASK`	Mask applied to destination IP address
`srcipfrom`	`IP_ADDR`	Start of range of source IP address
`srcipto`	`IP_ADDR`	End of range of source IP address
`dstipfrom`	`IP_ADDR`	Start of range of destination IP address
`dstipto`	`IP_ADDR`	End of range of destination IP address
`srcportstart`	`UINT16`	Start of range of valid source ports
`srcportend`	`UINT16`	End of range of valid source port
`dstportstart`	`UINT16`	Start of range of valid destination ports
`dstportend`	`UINT16`	End of range of valid destination ports
`dscp`	`UINT8` (`0x0-0x3f`, 0–63)	Differentiated Services Code Point
`comment` (since 0.8.5)	`STRING`	Text with a maximum of 256 characters
`state` (since 0.8.5)	`STRING`	Comma-separated list of `NEW`, `ESTABLISHED`, `RELATED`, `INVALID`, or `NONE`

(*continued*)

Table A-9. (*continued*)

Variable Name	Datatype	Semantics
`flags` (since 0.9.1)	`STRING`	TCP-only: Format of mask/flags with mask and flags, each being a comma-separated list of `SYN`, `ACK`, `URG`, `PSH`, `FIN`, `RST`, `NONE`, or `ALL6`
`ipset` (since 0.9.13)	`STRING`	The name of an IPSet managed outside of libvirt
`ipsetflags` (since 0.9.13)	`IPSETFLAGS`	Flags for the IPSet; requires `ipset` attribute

Note: The chain parameter is ignored for this type of traffic and should be either omitted or set to the `root` chain.

ICMP

Protocol ID: `icmp` (Table A-10)

Table A-10. *icmp Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcmacmask`	`MAC_MASK`	Mask applied to MAC address of sender
`dstmacaddr`	`MAC_ADDR`	MAC address of destination
`dstmacmask`	`MAC_MASK`	Mask applied to MAC address of destination
`srcipaddr`	`IP_ADDR`	Source IP address

(*continued*)

Table A-10. (*continued*)

Variable Name	Datatype	Semantics
`srcipmask`	`IP_MASK`	Mask applied to source IP address
`dstipaddr`	`IP_ADDR`	Destination IP address
`dstipmask`	`IP_MASK`	Mask applied to destination IP addresss
`srcipfromo`	`IP_ADDR`	Start of range of source IP address
`srcipto`	`IP_ADDR`	End of range of source IP address
`dstipfrom`	`IP_ADDR`	Start of range of destination IP address
`dstipto`	`IP_ADDR`	End of range of destination IP address
`type`	`UINT16`	ICMP type
`code`	UINT16	ICMP code
`dscp`	UINT8 (0x0-0x3f, 0 - 63)	Differentiated Services Code Point
`comment` (since 0.8.5)	STRING	Text with a maximum of 256 characters
`state` (since 0.8.5)	`STRING`	Comma-separated list of `NEW`, `ESTABLISHED`, `RELATED`, `INVALID`, or NONE
`ipset` (since 0.9.13)	`STRING`	The name of an IPSet managed outside of libvirt
`ipsetflags` (since 0.9.13)	`IPSETFLAGS`	Flags for the IPSet; requires `ipset` attribute

Note: The chain parameter is ignored for this type of traffic and should be either omitted or set to the `root` chain.

IGMP, ESP, AH, UDPLITE, ALL

Protocol ID: igmp, esp, ah, udplite, all (Table A-11)

Table A-11. *igmp, esp, ah, udplite, all Variables and Their Expected Values*

Variable Name	Datatype	Semantics
srcmacaddr	MAC_ADDR	MAC address of sender
srcmacmask	MAC_MASK	Mask applied to MAC address of sender
dstmacaddr	MAC_ADDR	MAC address of destination
dstmacmask	MAC_MASK	Mask applied to MAC address of destination
srcipaddr	IP_ADDR	Source IP address
srcipmask	IP_MASK	Mask applied to source IP address
dstipaddr	IP_ADDR	Destination IP address
dstipmask	IP_MASK	Mask applied to destination IP addresss
srcipfromo	IP_ADDR	Start of range of source IP address
srcipto	IP_ADDR	End of range of source IP address
dstipfrom	IP_ADDR	Start of range of destination IP address
dstipto	IP_ADDR	End of range of destination IP address
dscp	UINT8 (0x0-0x3f, 0 - 63)	Differentiated Services Code Point
comment (since 0.8.5)	STRING	Text with a maximum of 256 characters

(*continued*)

Table A-11. *(continued)*

Variable Name	Datatype	Semantics
`state` (since 0.8.5)	`STRING`	Comma-separated list of `NEW`, `ESTABLISHED`, `RELATED`, `INVALID`, or `NONE`
`ipset` (since 0.9.13)	`STRING`	The name of an IPSet managed outside of libvirt
`ipsetflags` (since 0.9.13)	`IPSETFLAGS`	Flags for the IPSet; requires `ipset` attribute

Note: The chain parameter is ignored for this type of traffic and should be either omitted or set to the `root` chain.

TCP/UDP/SCTP over IPV6

Protocol ID: `tcp-ipv6`, `udp-ipv6`, `sctp-ipv6` (Table A-12)

Table A-12. *tcp-ipv6, udp-ipv6, sctp-ipv6 Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MACV6_ADDR`	MAC address of sender
`srcipaddr`	`IPV6_ADDR`	Source IP address
`srcipmask`	`IPV6_MASK`	Mask applied to source IP address
`dstipaddr`	`IPV6_ADDR`	Destination IP address
`dstipmask`	`IPV6_MASK`	Mask applied to destination IP address
`srcipfrom`	`IPV6_ADDR`	Start of range of source IP address
`srcipto`	`IPV6_ADDR`	End of range of source IP address

(continued)

Table A-12. (*continued*)

Variable Name	Datatype	Semantics
`dstipfrom`	`IPV6_ADDR`	Start of range of destination IP address
`dstipto`	`IPV6_ADDR`	End of range of destination IP address
`srcportstart`	`UINT16`	Start of range of valid source ports
`srcportend`	`UINT16`	End of range of valid source port
`dstportstart`	`UINT16`	Start of range of valid destination ports
`dstportend`	`UINT16`	End of range of valid destination ports
`dscp`	`UINT8` (0x0-0x3f, 0–63)	Differentiated Services Code Point
`comment` (since 0.8.5)	`STRING`	Text with a maximum of 256 characters
`state` (since 0.8.5)	`STRING`	Comma-separated list of `NEW`, `ESTABLISHED`, `RELATED`, `INVALID`, or `NONE`
`flags` (since 0.9.1)	`STRING`	TCP-only: Format of mask/flags with mask and flags, with each being a comma-separated list of `SYN`, `ACK`, `URG`, `PSH`, `FIN`, `RST`, `NONE`, or `ALL6`
`ipset` (since 0.9.13)	`STRING`	The name of an IPSet managed outside of libvirt
`ipsetflags` (since 0.9.13)	`IPSETFLAGS`	Flags for the IPSet; requires `ipset` attribute

Note: The chain parameter is ignored for this type of traffic and should either be omitted or set to the `root` chain.

ICMPv6

Protocol ID: `icmpv6` (Table A-13)

Table A-13. *icmpv6 Variables and Their Expected Values*

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcmacaddr`	`MACV6_ADDR`	MAC address of sourcen
`srcmacmask`	`MACV6_MASK`	Mask applied to MAC address of source
`dstrcipaddr`	`IPV6_ADDR`	Destination IP destination
`dstipmask`	`IPV6_MASK`	Mask applied to IP destination
`srcipfrom`	`IPV6_ADDR`	Source IP address
`srcipto`	`IPV6_ADDR`	Mask applied to source IP addresss
`dstipfrom`	`IPV6_ADDR`	Start of range of destination IP address
`dstipto`	`IPV6_ADDR`	End of range of destination IP address
`dstipfrom`	`IP_ADDR`	Start of range of destination IP address
`dstipto`	`IP_ADDR`	End of range of destination IP address
`type`	`UINT16`	ICMP type
`code`	`UINT16`	ICMP code
`dscp`	`UINT8` (`0x0-0x3f`, 0–63)	Differentiated Services Code Point
`comment` (since 0.8.5)	`STRING`	Text with maximum of 256 characters

(continued)

Table A-13. (*continued*)

Variable Name	Datatype	Semantics
`state` (since 0.8.5)	`STRING`	Comma-separated list of `NEW`, `ESTABLISHED`, `RELATED`, `INVALID`, or `NONE`
`ipset` (since 0.9.13)	`STRING`	The name of an IPSet managed outside of libvirt
`ipsetflags` (since 0.9.13)	`IPSETFLAGS`	flags for the IPSet; requires `ipset` attribute

Note: The chain parameter is ignored for this type of traffic and should be either omitted or set to the `root` chain.

IGMP, ESP, AH, UDPLITE, ALL over IPv6

Protocol ID: `esp-ipv6`, `ah-ipv6`, `udplite-ipv6`, `all-ipv6` (Table A-14)

Table A-14. esp-ipv6, ah-ipv6, udplite-ipv6, all-ipv6 Variables and Their Expected Values

Variable Name	Datatype	Semantics
`srcmacaddr`	`MAC_ADDR`	MAC address of sender
`srcipaddr`	`IPV6_ADDR`	Source IP address
`srcipmask`	`IPV6_MASK`	Mask applied to source IP address
`dstipaddr`	`IPV6_ADDR`	Destination IP address
`dstipmask`	`IPV6_MASK`	Mask applied to destination IP addresss
`srcipfromo`	`IPV6_ADDR`	Start of range of source IP address

(*continued*)

Table A-14. (*continued*)

Variable Name	Datatype	Semantics
`srcipto`	`IPV6_ADDR`	End of range of source IP address
`dstipfrom`	`IPV6_ADDR`	Start of range of destination IP address
`dstipto`	`IPV6_ADDR`	End of range of destination IP address
`dscp`	`UINT8` (`0x0-0x3f`, `0–63`)	Differentiated Services Code Point
`comment` (since 0.8.5)	`STRING`	Text with maximum of 256 characters
`state` (since 0.8.5)	`STRING`	Comma-separated list of `NEW`, `ESTABLISHED`, `RELATED`, `INVALID`, or `NONE`
`ipset` (since 0.9.13)	`STRING`	The name of an IPSet managed outside of libvirt
`ipsetflags` (since 0.9.13)	`IPSETFLAGS`	Flags for the IPSet; requires `ipset` attribute

Note: The chain parameter is ignored for this type of traffic and should be either omitted or set to the `root` chain.

Advanced Filter Configuration Topics

The following sections discuss advanced filter configuration topics.

Connection Tracking

The network filtering subsystem (on Linux) makes use of the connection tracking support of iptables. This helps enforce the directionality of network traffic (state match) as well as counting and limiting the number of simultaneous connections toward a VM. As an example, if a VM has TCP port 8080 open as a server, clients may connect to the VM on port 8080. Connection tracking and enforcement of directionality then prevents the VM from initiating a connection from (TCP client) port 8080 to the host back to a remote host. More importantly, tracking helps to prevent remote attackers from establishing a connection back to a VM. For example, if the user inside the VM established a connection to port 80 on an attacker site, then the attacker will not be able to initiate a connection from TCP port 80 back toward the VM. By default, the connection state match enables connection tracking, and then enforcement of directionality of traffic is turned on.

Listing A-76 shows an example XML fragment where this feature has been turned off for incoming connections to TCP port 12345.

Listing A-76. Turning Off Port 12345

```
...
<rule direction='in' action='accept' statematch='false'>
  <tcp dstportstart='12345'/>
</rule>
...
```

This now allows incoming traffic to TCP port 12345 but also enables the initiation from (client) TCP port 12345 within the VM, which may or may not be desirable.

Limiting Number of Connections

To limit the number of connections a VM may establish, a rule must be provided that sets a limit of connections for a given type of traffic. If, for example, a VM is supposed to be allowed to ping only one other IP address at a time and is supposed to have only one active incoming SSH connection at a time, the XML fragment in Listing A-77 can be used to achieve this.

Listing A-77. Limiting Connections to a Port

```
...
<rule action='drop' direction='in' priority='400'>
  <tcp connlimit-above='1'/>
</rule>
<rule action='accept' direction='in' priority='500'>
  <tcp dstportstart='22'/>
</rule>
<rule action='drop' direction='out' priority='400'>
  <icmp connlimit-above='1'/>
</rule>
<rule action='accept' direction='out' priority='500'>
  <icmp/>
</rule>
<rule action='accept' direction='out' priority='500'>
  <udp dstportstart='53'/>
</rule>
<rule action='drop' direction='inout' priority='1000'>
  <all/>
</rule>
...
```

Note that the rule for the limit has to logically appear before the rule for accepting the traffic.

An additional rule for letting DNS traffic to port 22 go out the VM has been added to avoid SSH sessions not getting established for reasons related to DNS lookup failures by the SSH daemon. Leaving this rule out may otherwise lead to fun-filled debugging joy (because the SSH client seems to hang while trying to connect).

A lot of care must be taken with timeouts related to tracking traffic. An ICMP ping that the user may have terminated inside the VM may have a long timeout in the host's connection tracking system and therefore not allow another ICMP ping to go through for a while. Therefore, the timeouts have to be tuned in the host's sysfs (Listing A-78).

Listing A-78. Finding the Host's Timeouts

```
echo 3 > /proc/sys/net/netfilter/nf_conntrack_icmp_timeout
```

This sets the ICMP connection tracking timeout to three seconds. The effect of this is that once one ping is terminated, another one can start after three seconds. Further, I want to point out that a client that for whatever reason has not properly closed a TCP connection may cause a connection to be held open for a longer period of time, depending on what timeout the `TCP established` state timeout has been set to on the host. Also, idle connections may time out in the connection tracking system but can be reactivated once packets are exchanged. However, a newly initiated connection may force an idle connection into TCP. If the number of allowed connections is set to a too-low limit, the new connection is established and hits (not exceeds) the limit of allowed connections. For example, a key is pressed on the old SSH session, which now has become unresponsive because of its traffic being dropped. Therefore, the limit of connections should be rather high so that fluctuations in new TCP connections don't cause odd traffic behavior in relation to idle connections.

Command-Line Tools

Table A-15 lists the example network filters that are automatically installed with libvirt.

***Table A-15.** Command-Line Tools*

Name	Description
`no-arp-spoofing`	Prevent a VM from spoofing ARP traffic; this filter only allows ARP request and reply messages and enforces that those packets contain the MAC and IP addresses of the VM.
`allow-dhcpg`	Allow a VM to request an IP address via DHCP (from any DHCP server).
`allow-dhcp-server`	Allow a VM to request an IP address from a specified DHCP server. The dotted decimal IP address of the DHCP server must be provided in a reference to this filter. The name of the variable must be `DHCPSERVER`.
`no-ip-spoofing`	Prevent a VM from sending IP packets with a source IP address different from the one in the packet.
`no-ip-multicast`	Prevent a VM from sending IP multicast packets.
`clean-traffic`	Prevent MAC, IP, and ARP spoofing. This filter references several other filters as building blocks.

Note that most of these filters are only building blocks and require a combination with other filters to provide useful network traffic filtering. The most useful one in Table A-15 is the `clean-traffic` filter. This filter itself can, for example, be combined with the `no-ip-multicast` filter to prevent virtual machines from sending IP multicast traffic on top of the prevention of packet spoofing.

Writing Your Own Filters

Since libvirt provides only a couple of example networking filters, you may consider writing your own. When doing so, there are a couple of things you need to know regarding the network filtering subsystem and how it works internally. Certainly you also have to know and understand the protocols well that you want to be filtering on so that no further traffic than what you want can pass and that in fact the traffic you want to allow does pass.

The network filtering subsystem is currently available only on Linux hosts and works only for the QEMU and KVM types of virtual machines. On Linux it builds upon the support for ebtables, iptables, and ip6tables and makes use of their features. From the previous list of supported protocols, the following ones are implemented using ebtables:

- `mac`
- `stp`
- `vlan` (802.1Q)
- `arp`, `rarp`
- `ipv4`
- `ipv6mac`

All other protocols over IPv4 are supported using iptables; those over IPv6 are implemented using ip6tables.

On a Linux host, all traffic filtering instantiated by libvirt's network filter subsystem first passes through the filtering support implemented by ebtables and only then through the iptables or ip6table filters. If a filter tree has rules with the protocols `mac`, `stp`, `vlan` `arp`, `rarp`, `ipv4`, or `ipv6`, ebtables rules will automatically be instantiated.

The role of the `chain` attribute in the network filter XML is that internally a new user-defined ebtables table is created that then for example receives all ARP traffic coming from or going to a virtual machine if the chain `arp` has been specified. Further, a rule is generated in an interface's root chain that directs all IPv4 traffic into the user-defined chain. Therefore, all ARP traffic rules should then be placed into filters specifying this chain. This type of branching into user-defined tables is only supported with filtering on the ebtables layer.

Since 0.9.8, multiple chains for the same protocol can be created. For this, the name of the chain must have a prefix of one of the previously enumerated protocols. To create an additional chain for handling ARP traffic, a chain with name `arp-test` can be specified.

As an example, it is possible to filter on UDP traffic by source and destination ports using the `ip` protocol filter and specifying attributes for the protocol, source, and destination IP addresses and ports of UDP packets that are to be accepted. This allows early filtering of UDP traffic with ebtables. However, once an IP or IPv6 packet, such as a UDP packet, has passed the ebtables layer and there is at least one rule in a filter tree that instantiates the iptables or ip6tables rules, a rule to let the UDP packet pass will also be necessary to be provided for those filtering layers. This can be achieved with a rule containing an appropriate `udp` or `udp-ipv6` traffic filtering node.

Custom Filter 1

As an example, say you want to now build a filter that fulfills the following list of requirements:

- Prevents a VM's interface from MAC, IP, and ARP spoofing
- Prevents a VM's interface from MAC, IP, and ARP spoofing
- Allows the VM to send ping traffic from an interface but not let the VM be pinged on the interface
- Allows the VM to do DNS lookups (UDP toward port 53)

The requirement to prevent spoofing is fulfilled by the existing `clean-traffic` network filter; thus, you will reference this filter from your custom filter.

To enable traffic for TCP ports 22 and 80, you will add two rules to enable this type of traffic. To allow the VM to send ping traffic, you will add a rule for ICMP traffic. For simplicity reasons, you allow general ICMP traffic to be initiated from the VM, not just ICMP echo request and response messages. To then disallow all other traffic to reach or be initiated by the VM, you will then need to add a rule that drops all other traffic. Assuming the VM is called `test` and the interface you want to associate your filter with is called `eth0`, you can name your filter `test-eth0`. The result of these considerations is shown in Listing A-79.

Listing A-79. Enabling TCP Ports 22 and 80

```
<filter name='test-eth0'>
  <!-- reference the clean traffic filter to prevent
       MAC, IP and ARP spoofing. By not providing
       and IP address parameter, libvirt will detect the
       IP address the VM is using. -->
  <filterref filter='clean-traffic'/>

  <!-- enable TCP ports 22 (ssh) and 80 (http) to be reachable
-->
  <rule action='accept' direction='in'>
    <tcp dstportstart='22'/>
  </rule>

  <rule action='accept' direction='in'>
    <tcp dstportstart='80'/>
  </rule>

  <!-- enable general ICMP traffic to be initiated by the VM;
       this includes ping traffic -->
  <rule action='accept' direction='out'>
    <icmp/>
  </rule>

  <!-- enable outgoing DNS lookups using UDP -->
  <rule action='accept' direction='out'>
    <udp dstportstart='53'/>
  </rule>

  <!-- drop all other traffic -->
  <rule action='drop' direction='inout'>
    <all/>
  </rule>

</filter>
```

Note that none of the rules in the previous XML contains the IP address of the VM as either the source or destination address, yet the filtering of the traffic works correctly. The reason is that the evaluation of the rules internally happens on a per-interface basis, and the rules are evaluated based on the knowledge about which (tap) interface has sent or will receive the packet rather than what their source or destination IP address may be.

An XML fragment for a possible network interface description inside the domain XML of the test VM could then look like Listing A-80.

Listing A-80. Specifying an Interface Description

```
...
<interface type='bridge'>
  <source bridge='mybridge'/>
  <filterref filter='test-eth0'/>
</interface>
...
```

To more strictly control the ICMP traffic and enforce that only ICMP echo requests can be sent from the VM and only ICMP echo responses be received by the VM, the previous ICMP rule can be replaced with the two rules in Listing A-81.

Listing A-81. Enabling Outgoing ICMP Requests

```
<!-- enable outgoing ICMP echo requests-->
<rule action='accept' direction='out'>
  <icmp type='8'/>
</rule>

<!-- enable incoming ICMP echo replies-->
<rule action='accept' direction='in'>
  <icmp type='0'/>
</rule>
```

Custom Filter 2

In this example, you now want to build a similar filter as in the previous example but extend the list of requirements with an FTP server located inside the VM. Further, you will be using features that have been added in version 0.8.5. The requirements for this filter are as follows:

- Prevents a VM's interface from MAC, IP, and ARP spoofing
- Opens only TCP ports 22 and 80 of a VM's interface
- Allows the VM to send ping traffic from an interface but not let the VM be pinged on the interface
- Allows the VM to do DNS lookups (UDP toward port 53)
- Enable an FTP server (in active mode) to be run inside the VM

The additional requirement of allowing an FTP server to be run inside the VM maps into the requirement of allowing port 21 to be reachable for FTP control traffic as well as enabling the VM to establish an outgoing TCP connection originating from the VM's TCP port 20 back to the FTP client (FTP active mode). There are several ways this filter can be written; two solutions are presented here.

The first solution makes use of the `state` attribute of the TCP protocol that gives you a hook into the connection tracking framework of the Linux host. For the VM-initiated FTP data connection (FTP active mode), you use the `RELATED` state that allows you to detect that the VM-initiated FTP data connection is a consequence of (or "has a relationship with") an existing connection you control; thus, you want to allow it to let packets pass the firewall. The `RELATED` state, however, is valid only for the first packet of the outgoing TCP connection for the FTP data path. Afterward, the state to compare against is `ESTABLISHED`, which then applies equally to the incoming and outgoing direction. All this is related to the FTP data traffic originating from TCP port 20 of the VM. This then leads to the solution (since version 0.8.5; QEMU, KVM) shown in Listing A-82.

Listing A-82. Network Filter for FTP Traffic

```
<filter name='test-eth0'>
  <!-- reference the clean traffic filter to prevent
       MAC, IP and ARP spoofing. By not providing
       and IP address parameter, libvirt will detect the
       IP address the VM is using. -->
  <filterref filter='clean-traffic'/>

  <!-- enable TCP port 21 (ftp-control) to be reachable -->
  <rule action='accept' direction='in'>
    <tcp dstportstart='21'/>
  </rule>
```

```
<!-- enable TCP port 20 for VM-initiated ftp data connection
     related to an existing ftp control connection -->
<rule action='accept' direction='out'>
  <tcp srcportstart='20' state='RELATED,ESTABLISHED'/>
</rule>

<!-- accept all packets from client on the ftp data
connection -->
<rule action='accept' direction='in'>
  <tcp dstportstart='20' state='ESTABLISHED'/>
</rule>

<!-- enable TCP ports 22 (ssh) and 80 (http) to be reachable -->
<rule action='accept' direction='in'>
  <tcp dstportstart='22'/>
</rule>

<rule action='accept' direction='in'>
  <tcp dstportstart='80'/>
</rule>

<!-- enable general ICMP traffic to be initiated by the VM;
     this includes ping traffic -->
<rule action='accept' direction='out'>
  <icmp/>
</rule>

<!-- enable outgoing DNS lookups using UDP -->
<rule action='accept' direction='out'>
  <udp dstportstart='53'/>
</rule>
```

```
  <!-- drop all other traffic -->
  <rule action='drop' direction='inout'>
    <all/>
  </rule>

</filter>
```

Before trying a filter using the `RELATED` state, you have to make sure that the appropriate connection tracking module has been loaded into the host's kernel. Depending on the version of the kernel, you must run either one of the two commands in Listing A-83 before the FTP connection with the VM is established.

Listing A-83. modprob Commands

```
modprobe nf_conntrack_ftp   # where available  or
modprobe ip_conntrack_ftp   # if above is not available
```

If other protocols than FTP are to be used in conjunction with the `RELATED` state, their corresponding module must be loaded. Modules exist at least for the protocols `ftp`, `tftp`, `irc`, `sip`, `sctp`, and `amanda`.

The second solution makes uses the state flags of connections more than the previous solution did. In this solution, you take advantage of the fact that the `NEW` state of a connection is valid when the first packet of a traffic flow is seen. Subsequently, if the first packet of a flow is accepted, the flow becomes a connection and enters the `ESTABLISHED` state. This allows you to write a general rule for allowing packets of `ESTABLISHED` connections to reach the VM or be sent by the VM. You write specific rules for the first packets identified by the `NEW` state and for which ports they are acceptable. All packets for ports that are not explicitly accepted will be dropped, and therefore the connection will not go into the `ESTABLISHEDD` state, and any subsequent packets be dropped (Listing A-84) .

Listing A-84. A clean-traffic Filter to Prevent Spoofing

```
<filter name='test-eth0'>
  <!-- reference the clean traffic filter to prevent
       MAC, IP and ARP spoofing. By not providing
       and IP address parameter, libvirt will detect the
       IP address the VM is using. -->
  <filterref filter='clean-traffic'/>

  <!-- let the packets of all previously accepted connections
  reach the VM -->
  <rule action='accept' direction='in'>
    <all state='ESTABLISHED'/>
  </rule>

  <!-- let the packets of all previously accepted and related
  connections be sent from the VM -->
  <rule action='accept' direction='out'>
    <all state='ESTABLISHED,RELATED'/>
  </rule>

  <!-- enable traffic towards port 21 (ftp), 22 (ssh) and 80
  (http) -->
  <rule action='accept' direction='in'>
    <tcp dstportstart='21' dstportend='22' state='NEW'/>
  </rule>

  <rule action='accept' direction='in'>
    <tcp dstportstart='80' state='NEW'/>
  </rule>

  <!-- enable general ICMP traffic to be initiated by the VM;
       this includes ping traffic -->
  <rule action='accept' direction='out'>
```

```
    <icmp state='NEW'/>
  </rule>

  <!-- enable outgoing DNS lookups using UDP -->
  <rule action='accept' direction='out'>
    <udp dstportstart='53' state='NEW'/>
  </rule>

  <!-- drop all other traffic -->
  <rule action='drop' direction='inout'>
    <all/>
  </rule>

</filter>
```

Limitations

The following sections list the (current) limitations of the network filtering subsystem.

VM Migration

VM migration is supported only if the whole filter tree that is referenced by a virtual machine's top-level filter is also available on the target host. The network filter `clean-traffic`, for example, should be available on all libvirt installations of version 0.8.1 or later and thus enable migration of VMs that, for example, reference this filter. All other custom filters must be migrated using higher-layer software. It is outside the scope of libvirt to ensure that referenced filters on the source system are equivalent to those on the target system and vice versa.

Migration must occur between libvirt installations of version 0.8.1 or newer not to lose the network traffic filters associated with an interface.

VLAN Filtering on Linux

VLAN (802.1Q) packets, if sent by a virtual machine, cannot be filtered with rules for the protocol IDs `arp`, `rarp`, `ipv4`, and `ipv6` but only with the protocol IDs `mac` and `vlan`. Therefore, the example filter `clean-traffic` will not work as expected.

Index

W. D. Ashley, *Foundations of Libvirt Development*,
https://doi.org/10.1007/978-1-4842-4862-1