# libvirt 1.0

# Libvirt Application Development Guide Using Python

A guide to libvirt application development with Python

**W. David Ashley**

**Daniel Berrange**

**Chris Lalancette**

**Laine Stump**

**Daniel Veillard**

**Dani Coulson**

**David Jorm**

**Scott Radvan**

# libvirt 1.0 Libvirt Application Development Guide Using Python A guide to libvirt application development with Python Edition 1

| | |
|---|---|
| Author | W. David Ashley |
| Author | Daniel Berrange |
| Author | Chris Lalancette |
| Author | Laine Stump |
| Author | Daniel Veillard |
| Author | Dani Coulson |
| Author | David Jorm |
| Author | Scott Radvan |

This document provides a guide for libvirt application developers using python.

# Preface

## 1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

### 1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

**`Mono-spaced Bold`**

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file **`my_next_bestselling_novel`** in your current working directory, enter the **`cat my_next_bestselling_novel`** command at the shell prompt and press **`Enter`** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **`Enter`** to execute the command.
>
> Press **`Ctrl`**+**`Alt`**+**`F1`** to switch to the first virtual terminal. Press **`Ctrl`**+**`Alt`**+**`F7`** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in **`mono-spaced bold`**. For example:

> File-related classes include **`filesystem`** for file systems, **`file`** for files, and **`dir`** for directories. Each class has its own associated set of permissions.

**Proportional Bold**

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

[1] https://fedorahosted.org/liberation-fonts/

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).
>
> To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

***`Mono-spaced Bold Italic`*** or ***`Proportional Bold Italic`***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

> To connect to a remote machine using ssh, type **`ssh`** ***`username@domain.name`*** at a shell prompt. If the remote machine is **`example.com`** and your username on that machine is john, type **`ssh john@example.com`**.
>
> The **`mount -o remount`** ***`file-system`*** command remounts the named file system. For example, to remount the **`/home`** file system, the command is **`mount -o remount /home`**.
>
> To see the version of a currently installed package, use the **`rpm -q`** ***`package`*** command. It will return a result as follows: ***`package-version-release`***.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

> Publican is a *DocBook* publishing system.

## 1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in **`mono-spaced roman`** and presented thus:

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

Source-code listings are also set in **`mono-spaced roman`** but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

## 1.3. Notes and Warnings

Finally, we 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: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' won't cause data loss but may cause irritation and frustration.

**Warning**

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

## 2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a bug report at *http://libvirt.org/bugs.html*

# Introduction

Libvirt is a hypervisor-independent virtualization API and toolkit that is able to interact with the virtualization capabilities of a range of operating systems. It is free software under the GNU Lesser General Public License.

This chapter provides an introduction to libvirt and defines common terms that will be used throughout the guide.

## 1.1. Overview

Libvirt provides a common, generic and stable layer to securely manage domains on a node. As nodes may be remotely located, libvirt provides all methods required to provision, create, modify, monitor, control, migrate and stop the domains, within the limits of hypervisor support for these operations. Although multiple nodes may be accessed with libvirt simultaneously, the methods are limited to single node operations.

Libvirt is designed to work across multiple virtualization environments, which means that more common capabilities are provided as methods and functions. Due to this, certain specific capabilities may not be provided. For example, it does not provide high level virtualization policies or multi-node management features such as load balancing. However, method stability ensures that these features can be implemented on top of libvirt. To maintain this level of stability, libvirt seeks to isolate applications from the frequent changes expected at the lower level of the virtualization framework.

Libvirt is intended as a building block for higher level management tools and applications focusing on virtualization of a single node, with the only exception being domain migration between multiple node capabilities. It provides methods to enumerate, monitor and use the resources available on the managed node, including CPUs, memory, storage, networking and Non-Uniform Memory Access (NUMA) partitions. Although a management node can be located on a separate physical machine to the management program, this should only be done using secure protocols.

## 1.2. Glossary of terms

To avoid ambiguity regarding terms and concepts used in this guide, refer to the following table for their definitions.

| Term | Definition |
|---|---|
| **Domain** | An instance of an operating system (or subsystem in the case of container virtualization) running on a virtualized machine provided by the hypervisor. |
| **Hypervisor** | A layer of software allowing virtualization of a node in a set of virtual machines, which may have different configurations to the node itself. |
| **Node** | A single physical server. Nodes may be any one of many different types, and are commonly referred to by their primary purpose. Examples are storage nodes, cluster nodes, and database nodes. |
| **Storage Pool** | A collection of storage media, such as physical hard drives. A Storage Pool is sub-divided into smaller containers called Volumes, which may then be allocated to one or more Domains. |

| Term | Definition |
|---|---|
| **Volume** | A storage space, allocated from a Storage Pool. A Volume may be assigned to one or more Domains for use, and are commonly used inside Domains as virtual hard drives. |

Table 1.1. Terminology

# Architecture

This chapter describes the main principles and architecture choices behind the definition of the libvirt API and the Python libvirt module.

## 2.1. Object model

The scope of the libvirt API and the Python libvirt module is intended to extend to all functions necessary for deployment and management of virtual machines. This entails management of both the core hypervisor functions and 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 backend, allowing 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.

### 2.1.1. 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 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 defines the hypervisor to connect to, while the host part of the URI determines where it is located. Refer to *Section 3.2, “URI formats”* 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 LXC container virtualization. A connection object may 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 *Section 2.1.2, “Guest domains”*.

### 2.1.2. 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.

#### Unique identifiers

- **ID**: positive integer, unique amongst running guest domains on a single host. An inactive domain does not have an ID.
- **name**: short string, unique amongst all guest domains on a single host, both running and inactive. To ensure maximum portability between hypervisors, it is recommended that names only include alphanumeric (**`a`** - **`Z`**, **`0`** - **`9`**), hyphen ( **`-`** ) and underscore ( **`_`** ) characters.
- **UUID**: 16 unsigned bytes, guaranteed to be unique amongst 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 only be managed while it is running on the host. Once it is powered off, all trace 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.

Refer to *Chapter 4, Guest Domains* for further information about using guest domain objects.

### 2.1.3. 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 either:

- Remain isolated to the host; or
- 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:

#### Unique identifiers

- **name**: short string, unique amongst all virtual network on a single host, both running and inactive. For maximum portability between hypervisors, applications should only use the characters **`a-Z,0-9,-,_`** in names.
- **UUID**: 16 unsigned bytes, guaranteed to be unique amongst all 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 may be transient or persistent. A transient virtual network can only be managed 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 config. A transient network can be turned into a persistent network on the fly by defining a configuration for it.

After 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. For example, laptops using wireless networking.

Refer to *Chapter 6, Virtual Networks* for further information about using virtual network objects.

### 2.1.4. Storage pools

The storage pool object provides a mechanism for managing all types of storage on a host, such as local disk, logical volume group, iSCSI target, FibreChannel HBA and local/network file system. A pool refers to a quantity storage that is able to 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.

### Unique identifiers

- **name**: short string, unique amongst all storage pools on a single host, both running and inactive. For maximum portability between hypervisors applications should only rely on being able to use the characters **`a-Z,0-9,-,_`** in names.
- **UUID**: 16 unsigned bytes, guaranteed to be unique amongst all 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 may be transient, or persistent. A transient storage pool can only be managed while it is running on the host and, when powered off, all trace of it will disappear (the underlying physical storage still exists of course !). 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 config. A transient pool can be turned into a persistent pool on the fly by defining a configuration for it.

Refer to *Chapter 5, Storage Pools* for further information about using storage pool objects.

## 2.1.5. Storage volumes

The storage volume object provides management of an allocated block of storage within a pool, be it a disk partition, logical volume, 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 identifiers

### Unique identifiers

- name: short string, unique amongst all storage volumes within a storage pool. For maximum portability between implementations applications should only rely on being able to use the characters **`a-Z,0-9,-,_`** 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**: a opaque 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**: a file system path referring to the volume. The path is unique amongst all 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 *Section 5.7, “Volume overview”* for further information about using storage volume objects

## 2.1.6. 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 logical devices these provide, such as a NIC, disk, disk controller, sound card, etc. 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, though specific device types may have their own unique identifiers.

### Unique identifiers

- **name**: short string, unique amongst all 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 drivers, which implicitly removes all associated logical devices, and then assigned to a guest domain. Physical device information is also useful when working with the storage and networking APIs to determine what resources are available to configure.

## 2.2. Driver model

The libvirt library exposes a guaranteed stable API & ABI which is 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 of implementations of the public ABI, each targeting a different virtualization technology. Each implementation is referred to as a driver. When obtaining a instance of the **`virConnect`** class, the application developer can provide a URI to determine which hypervisor driver is activated.

No two virtualization technologies have exactly 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 the query certain capabilities of a hypervisor, such as the type of guest ABIs that are supported.

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 which have no native remote communication capability, libvirt provides a generic secure RPC service. This is discussed in detail later in this chapter.

### Hypervisor drivers

- **Xen**: The open source Xen hypervisor providing 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. Example local URI scheme **`xen:///`**.

- **QEMU**: 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. Example privileged URI scheme **`qemu:///system`**. Example unprivileged URI scheme **`qemu:///session`**

- **UML**: 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. Example privileged URI scheme **`uml:///system`**. Example unprivileged URI scheme **`uml:///session`**

- **OpenVZ**: 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. Example privileged URI scheme **`openvz:///system`**

- **LXC**: 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. Example privileged URI scheme **`lxc:///`**

- **Remote**: Generic secure RPC service talking to a **`libvirtd`** daemon. Encryption and authentication using a choice of TLS, x509 certificates, SASL (GSSAPI/Kerberos) and SSH tunneling. URIs follow the scheme of the desired driver, but with a hostname filled in, and a data transport name appended to the URI scheme. Example URI to talk to Xen over a TLS channel **`xen+tls://somehostname/`**. Example URI to talk to QEMU over a SASL channel **`qemu+tcp:///somehost/system`**

- **Test**: A mock driver, providing a virtual in-memory hypervisor covering all the libvirt APIs. Facilities testing of applications using libvirt, by allowing automated tests to run which exercise libvirt APIs without needing to deal with a real hypervisor Example default URI scheme **`test:///default`**. Example customized URI scheme **`test:///path/to/driver/config.xml`**

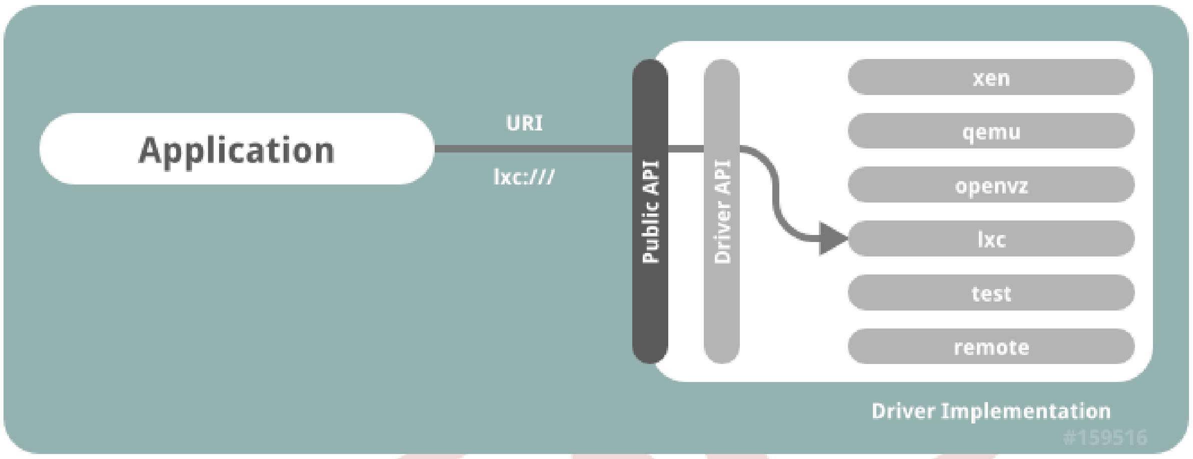

Figure 2.1. libvirt driver architecture

## 2.3. Remote management

While many virtualization technologies provide a remote management capability, libvirt does not assume this and provides a dedicated driver allowing for 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% 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 document. 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 1-to-1, 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.

### 2.3.1. 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 an default deployment this daemon will only be listening for connection on a local UNIX domain socket. This only allows for a libvirt client to use the SSH tunnel data transport. With suitable configuration of x509 certificates, or SASL credentials, the **`libvirtd`** daemon can be told to listen on a TCP socket for direct, non-tunneled client connections.

As can be seen from earlier example libvirt driver URIs, then 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 a 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 later in this document *Section 3.2.2, “Remote URIs”*.

## 2.3.2. 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.

| Transport | Description |
|---|---|
| tls | A TCP socket running the TLS protocol on the wire. This is the default data transport if none is explicitly requested, and uses a TCP connection on port 16514. At minimum it is necessary to configure the server with a 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 | 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 | A local only data transport, allowing users to connect to a **`libvirtd`** daemon running as a different user account. As it is only accessible 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 Netcat (nc) is 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 using SSH public key authentication it is recommended an ssh-agent by run to cache key credentials. GSSAPI is another useful authentication mode for the SSH transport allowing use of a pre-initialized Keberos credential cache. |

| Transport | Description |
|---|---|
| ext | 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 are satisfactory, this allows an application to provide a helper program to proxy RPC data over a custom channel. |

Table 2.1. Transports

### 2.3.3. 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.

| Scheme | Description |
|---|---|
| sasl | SASL is a 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 which 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 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. |

Table 2.2. Schemes

## 2.4. 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`**. Refer to *Table 3.3, “Extra parameters for remote URIs”* for details.

2. The server checks to ensure that only allowed clients are connected. This is performed using either:

   a. The client's IP address; or

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

   Server checking may 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(s) 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 server(s) and clients to trust certificates issues by your own CA. To do this, follow the instructions contained in the next section.

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. This setting may need to be made more or less permissive, dependent upon your requirements.

## 2.4.1. Public Key Infrastructure 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`**). |

Table 2.3. Public Key setup

# Connections

In libvirt, a connection is the underpinning of every action and object in the 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 it is interested in interacting with. A connection describes not only the type of virtualization technology that the agent wants to interact with (qemu, xen, uml, etc), but also describes any authentication methods necessary to connect to that resource.

## 3.1. Overview

The very 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 3 different functions for connecting to a resource:

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

In all three cases there is a **`name`** parameter which in fact refers to the URI of the hypervisor to connect to. The previous sections *Section 2.2, "Driver model"* and *Section 3.2.2, "Remote URIs"* provide 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 adhoc 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 outlined above is the way in which they authenticate and the resulting authorization level they provide.

### 3.1.1. 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 only succeed for connections where authentication can be done based on the credentials of the application.

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

Example 3.1. Using open

The above example opens up a read-write 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 *Section 3.2, "URI formats"*.

### 3.1.2. openReadOnly

The **`openReadOnly`** function will attempt to open a connection for read-only access. 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 relies on credentials.

```
# 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)
```

Example 3.2. Using ppenReadOnly

The above 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 *Section 3.2, “URI formats”*.

### 3.1.3. openAuth

The **`openAuth`** function is the most flexible, and effectively obsoletes the previous two functions. It takes an extra parameter providing an Python **`list`** which 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. A simple example Python program that uses **`openAuth`** with username and password credentials follows. As with **open**, this method has no scope for authentication callbacks, so relies on credentials.

```
# 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)
```

Example 3.3. Using openAuth

To test the above program, the following configuration 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 virt user has been added to the SASL database:

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

Once the above is configured, *Example 3.3, “Using openAuth”* 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.

## 3.1.4. 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 cooresponding call to the **`close`** method for each **`open`** function call.

```
# 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)
```

Example 3.4. Using close with additional references

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

## 3.2. 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 defines the hypervisor to connect to, while the host part of the URI determines where it is located.

### 3.2.1. 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 *Section 3.2.2, “Remote URIs”* for details on remote URIs.

The following drivers are currently supported:

| 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 |

Table 3.1. Supported Drivers

The following example shows how to connect to a local QEMU hypervisor using a local URI.

```
# 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)
```

Example 3.5. Connecting to a local QEMU hypervisor

### 3.2.2. Remote URIs

Remote URIs have the general form ("[...]" meaning an optional part):

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

Each component of the URI is described below.

| 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 psuedo 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. |
| 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 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. Mis-matched hostnames will guarantee authentication failures. |
| port | Rarely needed, unless SSH or libvirtd has been configured to run on a non-standard TCP port. 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 | The URI query parameters provide the mean to fine tune some aspects of the remote connection, and are discussed in depth in the next section. |

Table 3.2. URI components

Based on the information described here and with reference to the hypervisor specific URIs earlier in this document, it is now possible to illustrate some example remote access URIs.

Connect to a remote Xen hypervisor on host *`node.example.com`* using ssh tunneled data transport and 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 (NB: this is compromising your security): **`xen://node.example.com/?no_verify=1`**

Connect to the local QEMU instances over a non-standard 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 default configuration: **`test+tcp://node.example.com:5000/default`**

## Extra parameters

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

| Name | Transports | Description |
|---|---|---|
| **name** | *`any transport`* | The local hypervisor URI passed to the remote virConnectOpen function. This URI is normally formed by removing transport, hostname, port number, username and extra parameters from the remote URI, but in certain very complex cases it may be necessary to supply the name explicitly. Example: **`name=qemu:///system`** |
| **command** | ssh, ext | The external command. For ext transport this is required. For ssh the default is ssh. The PATH is searched for the command. Example: **`command=/opt/openssh/bin/ssh`** |
| **socket** | unix, ssh | The external command. For ext transport this is required. For ssh the default is **`ssh`**. The PATH 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 ssh transport, libvirt constructs an ssh command which looks like: `command -p port [-l username] hostname netcat -U socket` Where port, username, hostname can be specified as part of the remote URI, and command, netcat and socket come |

| Name | Transports | Description |
|---|---|---|
| | | from extra parameters (or sensible defaults). Example: **`netcat=/opt/netcat/bin/nc`** |
| **no_verify** | tls | Client checks of the server's certificate are disable if a non-zero 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 non-zero value, this stops ssh from asking for a password if it cannot log in to the remote machine automatically (For example, when using a ssh-agent). Use this when you don't have access to a terminal - for example in graphical programs which use libvirt. Example: **`no_tty=1`** |

Table 3.3. Extra parameters for remote URIs

The following example shows how to local to a QEMU hypervisor using a remote URI.

```
# 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)
```

Example 3.6. Connecting to a remote QEMU hypervisor

## 3.3. Capability information

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 below). If an error occurred, **`None`** will be returned instead. The following code demonstrates the use of the **`getCapabilities`** method:

```
# 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)
```

Example 3.7. Using getCapabilities

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. An example capabilities XML looks like:

```
<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>
```

Example 3.8. Example QEMU driver capabilities

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

The **`/host`** sub-document 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 the above are properly set, a temporary UUID will be generated each time that libvirtd is restarted.

The **`/host/cpu`** sub-document 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 if 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 sub-document that describes additional cpu features present on the host. As of this writing, it is only used 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 list of CPU models that libvirt currently know about are in the cpu_map.xml file.

**`/host/cpu/feature`** are 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 sub-document that describes additional cpu features present on the host. As of this writing, it is only used 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.

The **`/host/migration_features`** is an optional sub-document that describes the migration features that this driver supports on this host (if any). If this sub-document does not exist, then migration is not supported. As of this writing, the xen, qemu, and esx drivers support migration.

**`/host/migration_features/live`** XML node exists if the driver supports live migration

**`/host/migration_features/uri_transports`** is an optional sub-document that describes alternate migration connection mechanisms. These alternate connection mechanisms can be useful on multi-homed virtualization systems. For instance, the virsh migrate command might connect to the source of the migration via 10.0.0.1, and the destination of the migration via 10.0.0.2. However, due to 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`** sub-document describes the NUMA topology of the host machine; each NUMA node is represented by a **`/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 necessarily vary between different machines.

**`/host/secmodel`** is an optional sub-document that describes the security model in use on the host. **`/host/secmodel/model`** shows the name of the security model while **`/host/secmodel/doi`** shows the Domain Of Interpretation. For more information about security, please see the Security section.

Each **`/guest`** sub-document 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).

**`/guest/os_type`** is a required element that describes the type of guest.

| 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 |

Table 3.4. Guest Types

**`/guest/arch`** is the root of an XML sub-document describing various virtual hardware aspects of this guest type. It has a single attribute called "name", which can be used to refer back to this sub-document.

**`/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 (described below) 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 (described below) for guest types that use alternate loaders. At present, this is only used 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 be started with. Note that these machine types can be overridden by the **/guest/arch/domain/machine** elements (described below) 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 sub-trees (although with zero /guest/arch/domain XML sub-trees, no guests of this driver can be started). Each **/guest/arch/domain** XML sub-tree has optional <emulator>, <loader>, and <machine> elements that override the respective defaults specified above. For any of the elements that are missing, the default values are used.

The **/guest/features** optional sub-document 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.

## 3.4. Host information

There are various Python **virConnection** methods that can be used to get information about the virtualization host, including the hostname, maximum support guest CPUs, etc.

### 3.4.1. getHostname

The **getHostname** method can be used to obtain the hostname of the virtualization host as returned by gethostname(2). It invoked via a the virConnection instance and, if successful, returns a string containing the hostname. If an error occurred, NULL will be returned instead. It is the responsibility of the caller to free the memory returned from this method call. The following code demonstrates the use of **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)
```

Example 3.9. Using virConnectGetHostname

### 3.4.2. getMaxVcpus

The **getMaxVcpus** method can be used to obtain the maximum number of virtual CPUs per-guest 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 occurred, -1 is returned instead. The following code demonstrates the use of **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)
```

Example 3.10. Using virConnectGetMaxVcpus

### 3.4.3. 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 and error occurred. The Python **`list`** contains the following members:

| Member | Description |
|---|---|
| listl0] | string indicating the CPU model |
| list[1] | memory size in kilobytes |
| list[2] | the number of active CPUs |
| list[3] | expected CPU frequency (mhz) |
| list[4] | the number of NUMA nodes, 1 for uniform memory access |
| list[5] | number of CPU sockets per node |
| list[6] | number of cores per socket |
| list[7] | number of threads per core |

Table 3.5. virNodeInfo structure members

The following code demonstrates the use of **`virNodeGetInfo`**:

```
# 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)
```

Example 3.11. Using getInfo

### 3.4.4. getCellsFreeMemory

The **`getCellsFreeMemory`** method can be used to obtain the amount of free memory (in kilobytes) 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, Python **`list`** is returned with the amount of free memory in each node. On failure **`None`** is returned. The following code demonstrates the use of **`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)
```

Example 3.12. Using virNodeGetCellsFreeMemory

### 3.4.5. 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 occurred, **`None`** will be returned instead. The following code demonstrates the use of **`getType`**:

```
# 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)
```

Example 3.13. Using virConnectGetType

## 3.4.6. getVersion

The **`getVersion`** method can be used to obtain the version of the host virtualization software in use. If successful it returns a Python string with the version, otherwise it returns **`None`**. The following code demonstrates the use of **`getVersion`**:

```
# 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)
```

Example 3.14. Using virConnectGetVersion

## 3.4.7. 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`**. The following code demonstrates the use of **`getLibVersion`**:

```
# 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)
```

Example 3.15. Using virConnectGetLibVersion

## 3.4.8. 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. The following code demonstrates the use of **getURI**:

```
# 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)
```

Example 3.16. Using virConnectGetURI

### 3.4.9. isEncrypted

The **isEncrypted** method can be used to find out if a given connection is encrypted. If successful it returns 1 for an encrypted connection and 0 for an unencrypted connection. If an error occurred, -1 will be returned. The following code demonstrates the use of **isEncrypted**:

```
# 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)

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

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

Example 3.17. Using virConnectIsEncrypted

### 3.4.10. isSecure

The **isSecure** method can be used to find out if a given connection is encrypted. A connection will be classified secure if it is either encrypted or it is running on a channel which 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 occurred, -1 will be returned. The following code demonstrates the use of **isSecure**:

```
# 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)
```

Example 3.18. Using virConnectIsSecure

# Guest Domains

## 4.1. Domain Overview

A domain is an instance of an operating system running on a virtualized machine. A guest domain can refer to either a running virtual machine or a configuration which 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 the **`virDomainPtr`** object and has a number of unique identifiers:

### Unique identifiers

- **ID**: positive integer, unique amongst running guest domains on a single host. An inactive domain does not have an ID. If the host OS is a virtual domain, it is given an ID of zero by default. For example, with the Xen hypervisor, **`Dom0`** indicates a guest domain. Other domain IDs will be allocated starting at 1, and incrementing each time a new domain starts. Typically domain IDs will not be re-used until the entire ID space wraps around. The domain ID space is at least 16 bits in size, but often extends to 32 bits.

- **name**: short string, unique amongst all guest domains on a single host, both running and inactive. For maximum portability between hypervisors applications should only rely on being able to use the characters **`a-Z,0-9,-,_`** in names. Many hypervisors will store inactive domain configurations as files on disk, based on the domain name.

- **UUID**: 16 unsigned bytes, guaranteed to be unique amongst all guest domains on any host. RFC 4122 defines the format for UUIDs and provides a recommended algorithm for generating UUIDs with guaranteed uniqueness. If the host OS is itself a virtual domain, then by convention it will be given a UUID of all zeros. This is the case with the Xen hypervisor, where **`Dom0`** is a guest domain itself.

A guest domain may be transient, or persistent. A transient guest domain can only be managed while it is running on the host and, when 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 config. A transient guest can be turned into a persistent guest on the fly 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, imaginatively named, methods to do lookup 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.

```
# 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)
```

Example 4.1. Fetching a domain object from an ID

```
# 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()
```

Example 4.2. Fetching a domain object from a name

```
# 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)
```

Example 4.3. Fetching a domain object from a UUID

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

## 4.2. 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 non-overlapping, 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 amongst 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 **`int`**s.

```
# 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)
```

Example 4.4. Listing active domains

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).

```
# 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)
```

Example 4.5. Listing inactive domains

The methods for listing domains do not directly return the **`virDomain`** objects, since this may incur undue performance penalty for applications which wish to query the list of domains on a frequent basis. However, the Python libvirt module does provide the method **`listAllDomains`** which 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 which 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_SNAPSHO

```
# 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)
```

Example 4.6. Fetching all domain objects

## 4.3. Lifecycle Control

Libvirt can control the entire lifecycle of guest domains. Guest domains can 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: *Section 4.3.1, “Provisioning and Starting”*, *Section 4.3.3, “Suspend / Resume and Save / Restore”*, *Section 4.3.4, “Migration”* and *Section 4.3.5, “Autostart”*.

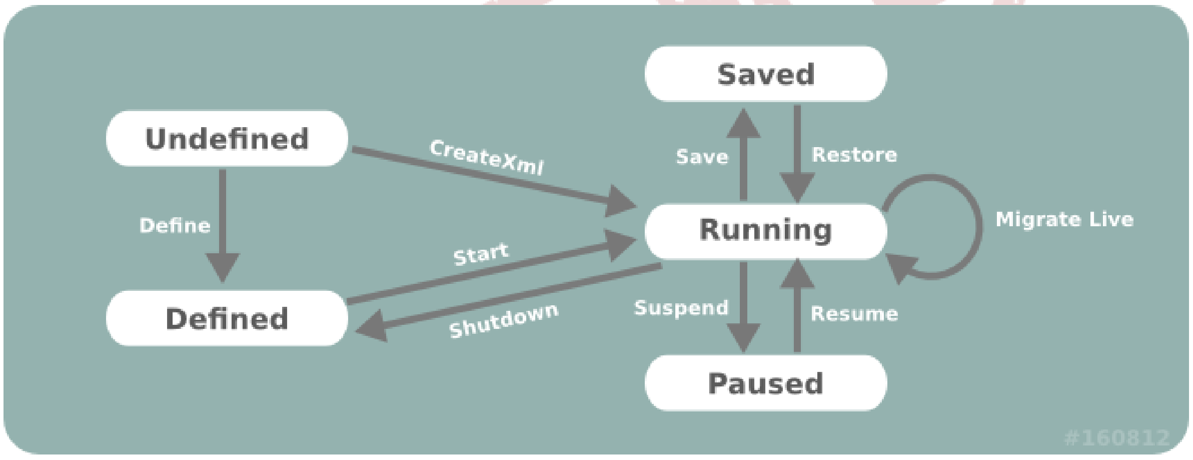

Figure 4.1. Guest domain lifecycle

### 4.3.1. 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.

#### 4.3.1.1. 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 trace 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.

#### 4.3.1.1.1. Booting a Transient Guest Domain

To boot 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 auto destroy will block attempts at migration, save-to-file, or snapshots.

```
# 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)
```

Example 4.7. Provisioning a Transient Guest Domain

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 shutdown (whether a planned shutdown, or a crash). To cope with this scenario requires use of a persistent guest.

#### 4.3.1.1.2. 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.

```
# 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)
```

Example 4.8. Defining and Booting a Persistent Guest Domain

### 4.3.1.2. 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 which 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.

```
<domain type='kvm'>
  <name>demo</name>
  <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 in the choice of initial memory allocation, since too low 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.

#### 4.3.1.2.1. CDROM/ISO image provisioning

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

The first obvious change required to the XML configuration to support CDROM installation, is to add a CDROM device. A guest domains' CDROM device can be pointed to either a host CDROM device, or to a 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 CDROM device, then the CDROM media won't be booted unless the first boot sector on the hard disk is blank. If the CDROM 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 would have the following XML chunk inserted:

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

NB, 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.

```
# 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)
```

Example 4.9. Defining and Booting a Persistent Guest Domain

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

### 4.3.1.2.2. 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 would have the following XML chunk inserted:

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

NB, this assumes the hard disk boot sector is blank initially, so that 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.

```
# 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)
```

Example 4.10. PXE Boot Provisioning

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

#### 4.3.1.2.3. 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 an 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 very easy to do 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.

```
<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>
```

Example 4.11. Kernel Boot Provisioning XML

Notice how the kernel command line provides the URL of 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 install URL. It could also be used to provide a kickstart file for 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 be '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 off, just installed, the hard disk again. The provisioning process can be started now by creating a transient guest with the first XML configuration

```
# 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.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)
```

Example 4.12. Kernel Boot Provisioning

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:

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

while a fully-virtualized guest would use:

```
<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 recreated, this time using a persistent configuration

```
# 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)
```

Example 4.13. Kernel Boot Provisioning for a Persistent Guest Domain

## 4.3.2. 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 only be stopped 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 which can, depending on what the guest OS supports, can shutdown the domain and leave the object in a usable state.

The **`destroy`** method immediately terminates the guest domain. The destroy process is analogous to pulling the plug on a physical machine.

## 4.3.3. Suspend / Resume and Save / Restore

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 execution 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 only save the memory state, no storage state is preserved. Thus when the guest is restored, the underlying guest storage must be in exactly the same state as it was when the guest was initially saved. For basic usage this implies that a guest can only be restored 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 time of saving, and explicitly revert to this storage snapshot when restoring. A future enhancement in libvirt will allow for an automated snapshot capability which saves memory and storage state in one operation.

The save operation requires the fully qualified path to a file in which the guest memory state will be saved. This filename is 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 that follows 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.

```
# 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)
```

Example 4.14. Saving a Guest Domain

Some period of time later, the saved state file can then be used to restart the guest where it left of, using the virDomainRestore 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 applications' responsibility to ensure the guest storage is in exactly the same state as it was when the save image was created

```
# 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)
```

Example 4.15. Restoring a Guest Domain

## 4.3.4. Migration

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, then instructions the domain to migrate to this connection. Both these methods can be passed a parameter to specify 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 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 can be set to **`None`**.

The forth parameter of the **`migrate`** method specifies the URI to be used as the target of the migration. A URI is only required 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.

To migrate a guest domain to a connection that is already open use the **`migrate`** method. An example follows:

```
# 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 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)
```

Example 4.16. Migrate a Domain to an Open Connection

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. An example follows:

```
# 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)
```

Example 4.17. Migrate a Domain to an Open Connection

To migrate a live guest domain to a URI use the **`migrate`** or the **`migrateToURI`** with the **`VIR_MIGRATE_LIVE`** flag set. An example follows:

```
# 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)
```

Example 4.18. Migrate a Domain to an Open Connection

### 4.3.5. Autostart

A guest domain can be configured to autostart on a particular hypervisor, either by the hypervisor itself or libvirt. In combination with 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 to regular save and restore, because paths must be known to libvirt without user input.

```
# 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)
```

Example 4.19. Set Autostart for a Domain

## 4.4. 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.

### 4.4.1. Boot Modes

Booting via the BIOS is available for hypervisors supporting full virtualization. In this case the BIOS has a boot order priority (floppy, harddisk, cdrom, network) determining where to obtain/find the boot image.

```
 ...
  <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>
  ...
```

Example 4.20. Setting the Boot Mode

### 4.4.2. 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 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 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 the current allocation 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 the **`setVcpusFlags`** methods. The number CPUs may not exceed the hypervisor maximum discussed above.

```
# 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)
```

Example 4.21. Set the number of maximum virtual cpus for a domain

Also after the domain has been created the amount of memory can be changes via the **`setMemory`** or the **`setMemoryFlags`** methods. The amount of memory should be expressed in kilobytes.

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

dom.setMemory(4096) # 4 GigaBytes

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

Example 4.22. Set the amount of memory for a domain

## 4.5. Monitoring Performance

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

### 4.5.1. Domain Block Device Performance

Disk usage statistics are provided by the **`blockStats`** method:

```
# 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)
```

Example 4.23. Get the disk block I/O statistics

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.

### 4.5.2. vCPU Performance

To obtain the individual VCPU statistics use the **`getCPUStats`** method.

```
# 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)
```

Example 4.24. Get the individual CPU statistics

The **`getCPUStats`** takes one parameter, a boolean. When **`False`** is used the statistics are reported as an aggregate of all the CPUs. Then **`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:

```
# 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)
```

Example 4.25. Get the aggregate CPU statistics

### 4.5.3. Memory Statistics

To obtain the amount of memory currently used by the domain you can use the **`memoryStats`** method.

```
# 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)
```

Example 4.26. Get the memory statistics

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

### 4.5.4. 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 the runtime). Then, look for devices/interface/target[@dev] element(s):

```
# 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)
```

Example 4.27. Get the network I/O statistics

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

## 4.6. 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

The following example shows how to obtain some basic information about the domain.

```
# 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)
```

Example 4.28. Get basic domain information from the domain's XML description

### 4.6.1. Emulator

To discover the guest domain's emulator find and display the content of the emulator XML tag.

```
# 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)
```

Example 4.29. Get domain's emulator information

The XML configuration for the Emulator is typically as follows:

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

Example 4.30. Domain Emulator XML information

### 4.6.2. Disks

To discover the guest domain's disk(s) find and display the content of the disk XML tag(s).

```
# 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)
```

Example 4.31. Get domain's disk information

The XML configuration for disks is typically as follows:

```
<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>
```

Example 4.32. Domain Disk XML information

## 4.6.3. Networking

To discover the guest domain's network interfaces find and display the interface XML tag.

```
# Example-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)
```

Example 4.33. Get domain's network interface information

The XML configuration for network interfaces is typically as follows:

```
<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>
```

Example 4.34. Domain network interface XML information

### 4.6.4. Mice, Keyboard & Tablets

To discover the guest domain's input devices find and display the input XML tags.

```
# 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)
```

Example 4.35. Get domain's input device information

The XML configuration for mouse, keyboard and tablet is typically as follows:

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

Example 4.36. Domain mouse, keyboard and tablet XML information

### 4.6.5. 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 only inherited by the guest domain at boot time. newly activated USB devices can not 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.

### 4.6.6. 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 the VT-d feature for Intel chipset, or 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 explicit 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 guest, 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 host and guest OS. 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 affect 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 will 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 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 a short example

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

Example 4.37. Get domain's input device information

## 4.7. Live Configuration Change

### 4.7.1. Block Device Jobs

Libvirt provides a 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 (eg. **`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:

1. **`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.

**`virDomainBlockJobAbort()`** can be used 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.

```
# 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


virConnectPtr conn
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)
```

Example 4.38. Get domain's input device information

# Storage Pools

Libvirt provides storage management on the physical host through storage pools and volumes.

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 the system administrator, and the volumes are assigned to VMs as block devices.

For example, 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 unfortunately 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 man virsh for more details.

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. 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 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.

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 will prefer to manage their own storage and VMs will 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, 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 backend
- Local filesystem backend

- Network filesystem backend
- Logical backend
- Disk backend
- iSCSI backend
- SCSI backend
- Multipath backend
- RBD (RADOS Block Device) backend
- Sheepdog backend
- Gluster backend
- ZFS backend

## 5.1. Overview

Storage pools are the containers for storage volumes. A system may have as many storage pools as needed and each storage pool may contain as many storage volumes as necessary.

## 5.2. Listing pools

A list of storage pool objects can be obtained using the **`listAllStoragePools`** method of the **`virConnect`** class.

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

```
# 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)
```

Example 5.1. Get the list of storage pools

## 5.3. Pool usage

There are a number of methods available in the **`virStoragePool`** class. The following example program features a number of the methods which describe some attributes of a pool.

```
# 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)
```

Example 5.2. Show the usage of some storage pool methods

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 that 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 the **`autostart`** is not set.

The **`isActive`** method indicates whether or not the user must activate the storage pool in some way. The **`create`** method can activate a storage pool.

The **`isPersistent`** method indicates whether or not a storage pool needs to be activated using **`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 one or more of the following constants:

VIR_STORAGE_XML_INACTIVE

The following example shows how to 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)
```

Example 5.3. Get the XML description of a storage pool

## 5.4. Lifecycle control

The following example shows how to create and destroy both a persistent and a non-persistent storage pool. Note that a storage pool can not be destroyed if it is in a active state. By default storage pools are created in a inactive state.

```
# 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)
```

Example 5.4. Create and destroy storage pools

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 because 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.

## 5.5. Discovering pool sources

The sources for a storage pool's sources can be discovered by examining the pool's XML description. An example program follows that prints out a pools source description attributes.

Currently the flags parameter for the **`storagePoolCreateXML`** method should always be **`0`**.

```
# 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)
```

Example 5.5. Discover a storage pool's sources

## 5.6. Pool configuration

There are a number of methods which can configure aspects of a storage pool. The main method is the **`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)
```

Example 5.6. Demonstrate the setAutostart method

## 5.7. Volume overview

Storage volumes are the basic unit of storage which 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.

## 5.8. Listing volumes

The following example program demonstrates how to list all the storage volumes contained by the "default" storage pool.

```
# 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)
```

Example 5.7. Demonstrate listing the storage volumes

## 5.9. Volume information

Information about a storage volume is obtained by using the **`info`** method. The following program shows how to list the information about each storage volume in the "default" storage pool.

```
# 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)

pool = conn.storagePoolLookupByName('default')
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)
```

Example 5.8. List storage volume information

## 5.10. Creating and deleting volumes

Storage volumes are created using the storage pool **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

```
# Example-.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)
```

Example 5.9. Create a storage volume

## 5.11. 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 very 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.

```
# Example-.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)
```

Example 5.10. Clone an existing storage volume

## 5.12. Configuring volumes

The following is an 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>
```

Example 5.11. XML description for a storage volume

# Virtual Networks

## 6.1. Overview

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 either:

- Remain isolated to the host; or
- 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 the **`virNetwork`** object and has two unique identifiers.

- name: short string, unique amongst all virtual network on a single host, both running and inactive. For maximum portability between hypervisors, applications should only use the characters a-Z,0-9,-,_ in names.
- UUID: 16 unsigned bytes, guaranteed to be unique amongst all 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 may be transient or persistent. A transient virtual network can only be managed while it is running on the host. When taken offline, all trace 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 config. A transient network can be turned into a persistent network on the fly by defining a configuration for it.

After 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. For example, laptops using wireless networking.

## 6.2. Listing networks

Virtual networks are discovered using the methods **`networkLookupByName`**, **`networkLookupByUUID`**, and **`networkLookupByUUIDString`** and **`listNetworks`**. The following example shows how to used these methods.

```
# 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)
```

Example 6.1. Discovering and finding virtual networks

## 6.3. Lifecycle control

The following example shows how to use the **`networkCreateXML`**, **`networkDefineXML`** and the **`destroy`** methods.

```
# 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)
```

Example 6.2. Creating and destroying virtual networks

## 6.4. Network configuration

The following example shows how to use the **`XMLDesc`**, **`autostart`**, **`isActive`**, **`isPersistent`** and the **`setAutostart`** methods.

```
# 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)
```

Example 6.3. Configuring virtual networks

# Network Interfaces

This section covers the management of physical network interfaces using the libvirt **`virInterface`** class.

## 7.1. Overview

The configuration of network interfaces on physical hosts can be examined and modified with methods in the **`virInterface`** class. This is useful for setting up the host to share one physical interface between multiple guest domains you want connected directly to the network (briefly - enslave a physical interface to the bridge, then create a tap device for each VM you want to share the interface), as well as for general host network interface management. In addition to physical hardware, the methods can also be used to configure bridges, bonded interfaces, and vlan interfaces.

The **`virInterface`** class is *not* used to configure virtual networks (used to conceal the guest domain's interface behind a NAT); virtual networks are instead configured using the **`virNetwork`** class described in *Chapter 6, Virtual Networks*.

Each host interface is represented by an instance of the **`virInterface`** class and each of these has a single unique identifier:

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 (eg: "eth0" or "br1").

Each interface object also has a second, non-unique 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 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 **`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.

## 7.2. 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*. Below are some examples of common interface configurations.

```
<interface type='ethernet' name='eth0'>
  <start mode='onboot'/>
  <mac address='aa:bb:cc:dd:ee:ff'/>
  <protocol family='ipv4'>
    <dhcp/>
  </protocol>
```

```
</interface>
```

Example 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'>
    <ip address="192.168.0.5" prefix="24"/>
    <route gateway="192.168.0.1"/>
  </protocol>
</interface>
```

Example 7.2. XML definition of an ethernet interface with static IP

```
<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>
```

Example 7.3. XML definition of a bridge device with eth0 and eth1 attached

```
<interface type="vlan" name="eth0.42">
  <start mode="onboot"/>
  <protocol family="ipv4">
    <dhcp peerdns="no"/>
  </protocol>
  <vlan tag="42">
    <interface name="eth0"/>
  </vlan>
</interface>
```

Example 7.4. XML definition of a vlan interface associated with eth0

## 7.3. Retrieving Information About Interfaces

### 7.3.1. Enumerating Interfaces

Once you have a connection to a host you can determine the number of interfaces on the host with **`numOfInterfaces`** and **`numOfDefinedInterfaces`** method. A list of those interfaces' names can be obtained with **`listInterfaces`** method and **`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.

Note: error handling omitted for clarity

```
# 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)
```

Example 7.5. Getting a list of active ("up") 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)
```

Example 7.6. Getting a list of inactive ("down") interfaces on a host

### 7.3.2. Obtaining a virInterface instance for an Interface

Many operations require that you have an instance of **`virInterface`**, but you may only have the name or MAC address of the interface. You can use **`interfaceLookupByName`** and **`interfaceLookupByMACString`** to get the **`virInterface`** instance in these cases.

Note: error handling omitted for clarity

```
# 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)
```

Example 7.7. Fetching the virInterface instance for a given interface name

Note: error handling omitted for clarity

```
# 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)
```

Example 7.8. Fetching the virInterface instance for a given interface MAC Address

## 7.3.3. 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.

```
# 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)
```

Example 7.9. Fetching the name and mac address 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)
```

Example 7.10. Fetching the XML configuration string from an interface object

## 7.4. 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 *Section 7.5, “Interface lifecycle management”*).

### 7.4.1. 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`**, described in *Section 7.5, “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`**.

```
# 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)
```

Example 7.11. Defining a new interface

### 7.4.2. Undefining an interface configuration

The **undefine** method completely and permanently removes the configuration for the given interface from the host's configuration files. If you want to recreate this configuration again in the future, you should invoke the **XMLDesc** method and save the string prior to the undefine.

```
# 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)
```

Example 7.12. Undefining br0 interface after saving its XML data

## 7.5. 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 using the *netcf* backend 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.

### 7.5.1. 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. *Example 7.11, "Defining a new interface"* shows typical usage of this method.

### 7.5.2. 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.

```
# 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)
```

Example 7.13. Temporarily bring down eth2, then bring it back up

# 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 returned 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 libvirt Python module defines a standard exception class **`libvirtError`** that can be subclassed to add additional functionality when raising a libvirt exception. A partial definition of the **`libvirtError`** class definition looks like:

```
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
        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]
```

Example 8.1. Libvirt module libvirtError class definition

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. Actually this is not of much use unless you subclass the **`virError`** class with your own class which would contain the needed behaviour. You should also make note of the methods which can obtain the error information. A description of these methods follows:

The method **`get_error_code`** returns the error code that was returned from the error. This is one of the data definition from the Python libvirt module. Some of the higher numbered entries from this list my 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 that for legacy reasons, but really represents which part of libvirt generated the error. This is one of the data definition from the Python libvirt module. Some of the higher numbered entries from this list my not be available in your Python libvirt module. The full list is:

```
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 definition from the Python libvirt module. The full list of levels is:

```
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 sub-sections.

## 8.1. 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. The following code demonstrates the use of **`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: '+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)

try:
    conn = libvirt.open('qemu:///system') # invalidate the parameter to force an error
except:
    report_libvirt_error()
conn.close()
```

```
exit(0)
```

Example 8.2. Using virGetLastError

## 8.2. Subclassing virError

It is possible to subclass the **`virtError`** class to add functionality. The default **`virError`** does not provide any ability to either save the error information or present the information to the user. Subclassing **`virError`** allows the programer the flexibility to add any functionality needed. An example of this follows:

```
# 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)
```

Example 8.3. Subclassing virError

## 8.3. Registering an Error Handler Function

Libvirt also supports setting up an error handler function. This is done using the libvirt function **`registerErrorHandler`**. An example of this follows:

```
# 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)
```

Example 8.4. Subclassing virError

# Event and Timer Handling

The Python libvirt module provides a complete interface for handling both events and timers. Both event 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.

## 9.1. 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`**.

An example program that uses most of these functions follows:

```
# 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()
```

Example 9.1. Provide a persistent console that survives guest reboots

## 9.2. 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 the **`VirEventAddTimout`** after the **`virEventRegisterImpl`** or the **`virEventRegisterDefaultImpl`** function has been invoked.

The timer can be removed using the **`VirEventRemoveTimout`** or updated with the **`virEventUpdateTimeout`** function after it has been added.

# Security Model

While the Python modules supplies full access to the security and secret methods, this topic is currently beyond the scope of this guide. You can look up the security and secret methods by running the following command.

```
pydoc libvirt
```

This will give you a manpage of the Python classes, methods and functions available specific to your Linux distribution.

# Debugging / Logging

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 3 key concepts:

1. 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, function name and line number indicating where the message originated from, and a formatted message.

2. Log filters - patterns and priorities which control whether or not a particular message is displayed. The format for a filter is:

   ```
   x:name
   ```

   where "x" is the minimal priority level where the match should apply, and "name" is a string to match against. The priority levels are:

   - 1 (or debug) - log all messages
   - 2 (or info) - log all non-debugging information
   - 3 (or warn) - 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 by 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
   ```

3. Log outputs - where to send the message once it has passed through filters. The format for a log output is one of the forms:

   ```
   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 by 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
```

## 11.1. Environment Variables

The desired log priority level, filters, and outputs are specified to the libvirt library through the use of environment variables:

1. **`LIBVIRT_DEBUG`** specifies the minimum priority level 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.

2. **`LIBVIRT_LOG_FILTERS`** specifies the filters to apply.

3. **`LIBVIRT_LOG_OUTPUTS`** specifies the outputs to send the message to.

To see more detailed information about what is going on with virsh, we may run it like the following:

```
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.

Example 11.1. Running virsh with environment variables

# Appendix A. Revision History

| | | |
|---|---|---|
| **Revision 1.0-1** | **Sat Aug 29 2015** | **W. David Ashley** *`w.david.ashley@gmail.com`* |

Initial release to the libvirt Development Project.

# Index

DRAFT