Application Development Guide

A guide to application development with libvirt

Daniel Berrange, Red Hat
Chris Lalancette, Red Hat
Laine Stump, Red Hat
Daniel Veillard, Red Hat
Dani Coulson, Red Hat
David Jorm, Red Hat
Scott Radvan, Red Hat

Application Development Guide: A guide to application development with libvirt

by Daniel Berrange, Chris Lalancette, Laine Stump, Daniel Veillard, Dani Coulson, David Jorm, and Scott Radvan

Abstract

This document provides a guide for application developers using libvirt.

Table of Contents

List of Figures

List of Tables

List of Examples

Preface

Document Conventions

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

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 keys 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 key, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from an individual key by the plus sign that connects each part of a key combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**+**Alt**+**F2** to switch to a virtual terminal.

The first example highlights a particular key to press. The second example highlights a key combination: a set of three keys 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 submenu titles. For example:

> Choose System → Preferences → Mouse from the main menu bar to launch Mouse Preferences. In the Buttons tab, select 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 high-

> lighted 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.

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"));
    }
}
```

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” will not cause data loss but may cause irritation and frustration.

Warning

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

We Need Feedback!

You should over ride this by creating your own local Feedback.xml file.

Chapter 1. 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.

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 APIs 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 APIs are limited to single node operations.

Libvirt is designed to work across multiple virtualization environments, which means that more common capabilities are provided as APIs. 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, API 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 APIs 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.

Glossary of terms

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

Table 1.1. Terminology

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

Term	Definition
	smaller containers called Volumes, which may then be allocated to one or more Domains.
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.

Chapter 2. Architecture

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

Object model

The scope of the libvirt API 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 APIs 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 an particular API is determined by the specific hypervisor driver in use and the capabilities of the underlying virtualization technology.

Hypervisor connections

A connection is the primary or top level object in the libvirt API. An instance of this object is required before attempting to use almost any of the APIs. 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 with the `virConnectPtr` object 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 the section called “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 the section called “Guest domains”.

Guest domains

A guest domain can refer to either a running virtual machine or a configuration that can be used to launch a virtual machine. The connection object provides APIs 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.
- 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.

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 the `virNetworkPtr` object 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 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.

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 the `virStoragePoolPtr` object 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.

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 the `virStorageVolPtr` object, 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 the section called “Volume overview” for further information about using storage volume objects

Host devices

Host devices provide a view to the hardware devices available on the host machine. This covers both the physical USB or PCI devices and 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 the `virNodeDevPtr` object, 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.

Refer to Chapter 8, *Host Devices* for further information about using host device objects.

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 `virConnectPtr` object, 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

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.

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 the section called “Remote URIs”

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.

Table 2.1. Transports

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 un-

Transport	Description
	trusted 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.
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.

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.

Table 2.2. Schemes

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

Scheme	Description
	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.

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.

Public Key Infrastructure setup

Table 2.3. Public Key setup

Location	Machine	Description	Required fields
`/etc/pki/CA/cacert.pem`	Installed on all clients and servers	CA's certificate	n/a

Location	Machine	Description	Required fields
`/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`).

Chapter 3. 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.

Overview

The very first thing a libvirt agent must do is call one of the libvirt connection functions to obtain a `virConnectPtr` handle. This handle will be used in subsequent operations. The libvirt library provides 3 different functions for connecting to a resource:

```
virConnectPtr virConnectOpen(const char *name)
virConnectPtr virConnectOpenReadOnly(const char *name)
virConnectPtr virConnectOpenAuth(const char *name, virConnectAuthPtr auth, int
```

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 the section called “Driver model” and the section called “Remote URIs” provide full details on the various URI formats that are acceptable. If the URI is NULL 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.

virConnectOpen

The `virConnectOpen` API 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 3.1. Using virConnectOpen

```
/* example ex1.c */
/* compile with: gcc -g -Wall ex1.c -o ex -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
```

```
        return 1;
    }
    virConnectClose(conn);
    return 0;
}
```

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 the section called “URI formats”.

virConnectOpenReadOnly

The `virConnectOpenReadOnly` API will attempt to open a connection for read-only access. Such a connection has a restricted set of API calls that are allowed, and is typically useful for monitoring applications that should not be allowed to make changes. As with `virConnectOpen`, this API has no scope for authentication callbacks, so relies on credentials.

Example 3.2. Using virConnectOpenReadOnly

```
/* example ex2.c */
/* compile with: gcc -g -Wall ex2.c -o ex2 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpenReadOnly("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }
    virConnectClose(conn);
    return 0;
}
```

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 the section called “URI formats”.

virConnectOpenAuth

The `virConnectOpenAuth` API is the most flexible, and effectively obsoletes the previous two APIs. It takes an extra parameter providing an instance of the `virConnectAuthPtr` struct which contains the callbacks for collecting authentication credentials from the client app. This allows libvirt to prompt for usernames, passwords, and more. The libvirt API provides an instance of this struct via the symbol `virConnectAuthPtrDefault` that implements callbacks suitable for a command line based application. Graphical applications will need to provide their own callback implementations. The flags parameter allows the application to request a read-only connection with the *`VIR_CONNECT_RO`* flag if desired. A simple example C program that uses `virConnectOpenAuth` with the default `virConnectAuthPtrDefault` is

Example 3.3. Using virConnectOpenAuth

```
/* example ex3.c */
/* compile with: gcc -g -Wall ex3.c -o ex3 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpenAuth("qemu+tcp://localhost/system", virConnectAuthPtrDefa
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu+tcp://localhost/system\
        return 1;
    }
    virConnectClose(conn);
    return 0;
}
```

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:

   ```
   # saslpasswd2 -a libvirt virt # this will prompt for a password
   ```

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

Once the above is configured, Example 3.3, “Using virConnectOpenAuth” should prompt for a username and password and allow read-write access to libvirtd.

If additional functionality is needed, a custom credential callback can also be provided as in the following program:

Example 3.4. Using a custom credential callback with virConnectOpenAuth

```
/* example ex4.c */
/* compile with: gcc -g -Wall ex4.c -o ex4 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <string.h>
```

```
static int authCreds[] = {
    VIR_CRED_AUTHNAME,
    VIR_CRED_PASSPHRASE,
};

static int authCb(virConnectCredentialPtr cred, unsigned int ncred, void *cbdata)
{
    int i;
    char buf[1024];

    for (i = 0; i < ncred; i++) {
        if (cred[i].type == VIR_CRED_AUTHNAME) {
            printf("%s: ", cred[i].prompt);
            fflush(stdout);
            fgets(buf, sizeof(buf), stdin);
            buf[strlen(buf) - 1] = '\0';
            cred[i].result = strdup(buf);
            if (cred[i].result == NULL)
                return -1;
            cred[i].resultlen = strlen(cred[i].result);
        }
        else if (cred[i].type == VIR_CRED_PASSPHRASE) {
            printf("%s: ", cred[i].prompt);
            fflush(stdout);
            fgets(buf, sizeof(buf), stdin);
            buf[strlen(buf) - 1] = '\0';
            cred[i].result = strdup(buf);
            if (cred[i].result == NULL)
                return -1;
            cred[i].resultlen = strlen(cred[i].result);
        }
    }

    return 0;
}

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virConnectAuth auth;

    auth.credtype = authCreds;
    auth.ncredtype = sizeof(authCreds)/sizeof(int);
    auth.cb = authCb;
    auth.cbdata = NULL;

    conn = virConnectOpenAuth("qemu+tcp://localhost/system", &auth, 0);
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu+tcp://localhost/system\
        return 1;
    }
    virConnectClose(conn);
    return 0;
}
```

The same configuration as Example 3.3, “Using virConnectOpenAuth” must be present in order to test Example 3.4, “Using a custom credential callback with virConnectOpenAuth”. The first thing to note here is the use of a `virConnectAuth` structure, which looks like the following:

```
struct _virConnectAuth {
    int *credtype; /* List of supported virConnectCredentialType values */
    unsigned int ncredtype;

    virConnectAuthCallbackPtr cb; /* Callback used to collect credentials */
    void *cbdata;
};
typedef struct _virConnectAuth virConnectAuth;
```

The credtype member points to an array of integers that represent the type of credentials this callback is willing to support. In Example 3.4, “Using a custom credential callback with virConnectOpenAuth” the authCreds array specifies all of the types that authCb supports; the full list of credential types is available in libvirt.h. The ncredtype member specifies the size of the credtype array. The cb member is a function pointer which specifies the callback that will be used when necessary; its signature must be:

```
typedef int (*virConnectAuthCallbackPtr)(virConnectCredentialPtr cred,
                                         unsigned int ncred,
                                         void *cbdata);
```

Finally, the cbdata member is a pointer that can point to additional data needed by the callback; in Example 3.4, “Using a custom credential callback with virConnectOpenAuth”, this is not used so it is set to NULL.

After setting up the auth structure, Example 3.4, “Using a custom credential callback with virConnectOpenAuth” goes on to use this structure in the `virConnectOpenAuth` function. When the libvirt internals require credentials, the callback in auth.cb (authCb) will be called. The cred parameter to this function is an array of `virConnectCredential` structures (described below) that libvirt needs to finish the authentication. The ncred parameter specifies the size of the cred array. Finally, the cbdata parameter is a pointer that contains the value passed in from auth.cbdata.

It is the responsibility of the auth.cb callback to examine each of the `virConnectCredential` structures and collect the necessary credentials. The `virConnectCredential` structure looks like:

```
struct _virConnectCredential {
    int type; /* One of virConnectCredentialType constants */
    const char *prompt; /* Prompt to show to user */
    const char *challenge; /* Additional challenge to show */
    const char *defresult; /* Optional default result */
    char *result; /* Result to be filled with user response (or defresult) */
    unsigned int resultlen; /* Length of the result */
};
typedef struct _virConnectCredential virConnectCredential;
```

In the case of example 4, authCb only handles VIR_CRED_AUTHNAME and VIR_CRED_PASSPHRASE, so for each of these credential types it prints out a prompt from the credential structure, collects the information into a temporary buffer, and then copies that buffer into the result and resultlen of that particular credential. Handling a credential but leaving result or resultlen as 0 is a programming error. If collection of all credentials is successful, auth.cb should return 0; otherwise, it should return -1 and libvirt will fail the connection.

virConnectClose

A connection must be released by calling `virConnectClose` when no longer required. Connections are reference counted objects, so if it is intended for a connection to be used from multiple threads at once, each additional thread should call `virConnectRef` to ensure the connection is not freed while still in use. Every extra call to `virConnectRef` must be accompanied by a corresponding call to `virConnectClose` to release the reference when no longer required. An example program that uses additional references:

Example 3.5. Using virConnectClose with additional references

```
/* example ex5.c */
/* compile with: gcc -g -Wall ex5.c -o ex5 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }
    /* now the connection has a single reference to it */

    virConnectRef(conn);

    /* now the connection has two references to it */

    virConnectClose(conn);

    /* now the connection has one reference */

    virConnectClose(conn);

    /* now the connection has no references, and has been garbage
     * collected - it should no longer be used */

    return 0;
}
```

Also note that every other object associated with a connection (virDomainPtr, virNetworkPtr, etc) will also hold a reference on the connection. To avoid leaking a connection object, applications must ensure all associated objects are also freed.

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.

Local URIs

Libvirt local URIs have one of the following forms:

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

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

The following drivers are currently supported:

Table 3.1. Supported Drivers

Driver	Description
qemu	For managing qemu and KVM guests
xen	For managing old-style (Xen 3.1 and older) Xen guests
xenapi	For managing new-style Xen guests
uml	For 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 mmanaging OpenNebula guests
phyp	For managing Power Hypervisor guests

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

Example 3.6. Connecting to a local QEMU hypervisor

```
/* example ex6.c */
/* compile with: gcc -g -Wall ex6.c -o ex6 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }
    virConnectClose(conn);
    return 0;
}
```

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.

Table 3.2. URI components

Component	Description
driver	The name of the libvirt hypervisor driver to connect to. This is the same as that used in a local URI. Some examples are `xen`, `qemu`, `lxc`, `openvz`, and `test`. As a special case, the 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.

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.

Table 3.3. Extra parameters for remote URIs

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, lib-

Name	Transports	Description
		virt constructs an ssh command which looks like: `command -p port [-l username] hostn` Where port, username, hostname can be specified as part of the remote URI, and command, netcat and socket come from extra parameters (or sensible defaults). Example: `netcat=/opt/netcat/bin/nc`
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`

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

Example 3.7. Connecting to a remote QEMU hypervisor

```
/* example ex7.c */
/* compile with: gcc -g -Wall ex7.c -o ex7 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpen("qemu+tls://host2/system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu+tls://host2/system\n");
        return 1;
    }
    virConnectClose(conn);
    return 0;
```

```
}
```

Capability information

The `virConnectGetCapabilities` API call can be used to obtain information about the capabilities of the virtualization host. It takes a connection pointer in and, if successful, returns a string containing the capabilities XML (described below). If an error occurred, NULL will be returned instead. It is the responsibility of the caller to free the memory returned from this API call. The following code demonstrates the use of `virConnectGetCapabilities`:

Example 3.8. Using virConnectGetCapabilities

```
/* example ex8.c */
/* compile with: gcc -g -Wall ex8.c -o ex8 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    char *caps;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    caps = virConnectGetCapabilities(conn);
    fprintf(stdout, "Capabilities:\n%s\n", caps);
    free(caps);

    virConnectClose(conn);
    return 0;
}
```

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

Example 3.9. Example QEMU driver capabilities

```
 <capabilities>
  <host>
    <cpu>
      <arch>x86_64</arch>
    </cpu>
    <migration_features>
      <live/>
      <uri_transports>
```

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

<guest>
  <os_type>hvm</os_type>
  <arch name='i686'>
    <wordsize>32</wordsize>
    <emulator>/usr/bin/qemu</emulator>
    <machine>pc</machine>
    <machine>isapc</machine>
    <domain type='qemu'>
    </domain>
    <domain type='kvm'>
      <emulator>/usr/bin/qemu-kvm</emulator>
    </domain>
  </arch>
  <features>
    <pae/>
    <nonpae/>
    <acpi default='on' toggle='yes'/>
    <apic default='on' toggle='no'/>
  </features>
</guest>

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

```
</capabilities>
```

(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.

Table 3.4. Guest Types

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

/guest/arch is the root of an XML 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.

Host information

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

virConnectGetHostname

The **virConnectGetHostname** API call can be used to obtain the hostname of the virtualization host as returned by gethostname(2). It takes a connection pointer in 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 API call. The following code demonstrates the use of **virConnectGetHostname**:

Example 3.10. Using virConnectGetHostname

```
/* example ex9.c */
/* compile with: gcc -g -Wall ex9.c -o ex9 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    char *host;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    host = virConnectGetHostname(conn);
    fprintf(stdout, "Hostname:%s\n", host);
    free(host);

    virConnectClose(conn);
    return 0;
}
```

virConnectGetMaxVcpus

The **virConnectGetMaxVcpus** API call can be used to obtain the maximum number of virtual CPUs per-guest the underlying virtualization technology supports. It takes a connection pointer and a virtualization "type" as input (which can be NULL), 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 **virConnectGetMaxVcpus**:

Example 3.11. Using virConnectGetMaxVcpus

```
/* example ex10.c */
/* compile with: gcc -g -Wall ex10.c -o ex10 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    int vcpus;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    vcpus = virConnectGetMaxVcpus(conn, NULL);
    fprintf(stdout, "Maximum support virtual CPUs: %d\n", vcpus);

    virConnectClose(conn);
    return 0;
}
```

virNodeGetFreeMemory

The **virNodeGetFreeMemory** API call can be used to obtain the total amount of free memory on the virtualization host. It takes a connection pointer as input, and if successful returns the amount of free memory as an unsigned long long. If an error occurred, 0 is returned instead. The following code demonstrates the use of **virNodeGetFreeMemory**:

Example 3.12. Using virNodeGetFreeMemory

```
/* example ex11.c */
/* compile with: gcc -g -Wall ex11.c -o ex11 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    unsigned long long node_free_memory;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    node_free_memory = virNodeGetFreeMemory(conn);
```

```
    fprintf(stdout, "Node free memory: %llu\n", node_free_memory);

    virConnectClose(conn);
    return 0;
}
```

virNodeGetInfo

The **virNodeGetInfo** API call can be used to obtain various information about the virtualization host. It takes a connection pointer and a virNodeInfo pointer (allocated by the caller) as input, and if successful returns 0 and fills in the virNodeInfo structure. If an error occurred, -1 is returned instead. The virNodeInfo structure contains the following members:

Table 3.5. virNodeInfo structure members

Member	Description
char model[32]	string indicating the CPU model
unsigned long memory	memory size in kilobytes
unsigned int cpus	the number of active CPUs
unsigned int mhz	expected CPU frequency
unsigned int nodes	the number of NUMA nodes, 1 for uniform memory access
unsigned int sockets	number of CPU sockets per node
unsigned int cores	number of cores per socket
unsigned int threads	number of threads per core

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

Example 3.13. Using virNodeGetInfo

```
/* example ex12.c */
/* compile with: gcc -g -Wall ex12.c -o ex12 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virNodeInfo nodeinfo;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    virNodeGetInfo(conn, &nodeinfo);

    fprintf(stdout, "Model: %s\n", nodeinfo.model);
```

```
    fprintf(stdout, "Memory size: %lukb\n", nodeinfo.memory);
    fprintf(stdout, "Number of CPUs: %u\n", nodeinfo.cpus);
    fprintf(stdout, "MHz of CPUs: %u\n", nodeinfo.mhz);
    fprintf(stdout, "Number of NUMA nodes: %u\n", nodeinfo.nodes);
    fprintf(stdout, "Number of CPU sockets: %u\n", nodeinfo.sockets);
    fprintf(stdout, "Number of CPU cores per socket: %u\n", nodeinfo.cores);
    fprintf(stdout, "Number of CPU threads per core: %u\n", nodeinfo.threads);

    virConnectClose(conn);
    return 0;
}
```

virNodeGetCellsFreeMemory

The **virNodeGetCellsFreeMemory** API call 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 a connection pointer, a preallocated array of unsigned long longs, and a maximum number of cells to retrieve data from (usually the size of the unsigned long long array). If successful, the array elements are filled in with the amount of free memory in each node, and the number of filled nodes is returned. On failure -1 is returned. The following code demonstrates the use of **virNodeGetCellsFreeMemory**:

Example 3.14. Using virNodeGetCellsFreeMemory

```
/* example ex13.c */
/* compile with: gcc -g -Wall ex13.c -o ex13 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virNodeInfo nodeinfo;
    unsigned long long *freemem;
    int i;
    int numnodes;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    /* first, get the node info.  This includes the number of nodes
     * in the host in nodeinfo.nodes
     */
    virNodeGetInfo(conn, &nodeinfo);

    /* allocate an array to hold all of the node free memory information */
    freemem = malloc(nodeinfo.nodes * sizeof(unsigned long long));

    /* fetch all the numa node free memory information from libvirt */
    numnodes = virNodeGetCellsFreeMemory(conn, freemem, 0, nodeinfo.nodes);
```

```
    for (i = 0; i < numnodes; i++)
     fprintf(stdout, "Node %d: %llukb free memory\n", i, freemem[i]);

    free(freemem);

    virConnectClose(conn);
    return 0;
}
```

virConnectGetType

The **virConnectGetType** API call can be used to obtain the type of virtualization in use on this connection. It takes a connection pointer as input, and if successful returns a string representing the type of virtualization in use. This string should *not* be freed by the caller. If an error occurred, NULL will be returned instead. The following code demonstrates the use of **virConnectGetType**:

Example 3.15. Using virConnectGetType

```
/* example ex14.c */
/* compile with: gcc -g -Wall ex14.c -o ex14 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    fprintf(stdout, "Virtualization type: %s\n", virConnectGetType(conn));

    virConnectClose(conn);
    return 0;
}
```

virConnectGetVersion

The **virConnectGetVersion** API call can be used to obtain the version of the host virtualization software in use. It takes a connection pointer and unsigned long pointer as input, and if successful fills in the unsigned long with the version. On success it returns 0 and if a failure occured returns -1. The following code demonstrates the use of **virConnectGetVersion**:

Example 3.16. Using virConnectGetVersion

```
/* example ex15.c */
/* compile with: gcc -g -Wall ex15.c -o ex15 -lvirt */
#include <stdio.h>
#include <stdlib.h>
```

```
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    unsigned long ver;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    virConnectGetVersion(conn, &ver);

    fprintf(stdout, "Version: %lu\n", ver);

    virConnectClose(conn);
    return 0;
}
```

virConnectGetLibVersion

The **virConnectGetLibVersion** API call can be used to obtain the version of the libvirt software in use on the host. It takes a connection pointer and unsigned long pointer as input, and if successful fills in the unsigned long with the libvirt version. On success it returns 0 and if a failure occured returns -1. The following code demonstrates the use of **virConnectGetLibVersion**:

Example 3.17. Using virConnectGetLibVersion

```
/* example ex16.c */
/* compile with: gcc -g -Wall ex16.c -o ex16 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    unsigned long ver;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    virConnectGetLibVersion(conn, &ver);

    fprintf(stdout, "Libvirt Version: %lu\n", ver);

    virConnectClose(conn);
    return 0;
}
```

virConnectGetURI

The **virConnectGetURI** API call can be used to obtain the URI for the current connection. While this is typically the same string that was passed into the **virConnectOpen** call, the underlying driver can sometimes canonicalize the string. This API call will return the canonical version. It takes a connection pointer as input and if successful, returns a URI string that must be freed by the caller. If an error occurred, NULL will be returned instead. The following code demonstrates the use of **virConnectGetURI**:

Example 3.18. Using virConnectGetURI

```
/* example ex17.c */
/* compile with: gcc -g -Wall ex17.c -o ex17 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    char *uri;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    uri = virConnectGetURI(conn);

    fprintf(stdout, "Canonical URI: %s\n", uri);

    free(uri);

    virConnectClose(conn);
    return 0;
}
```

virConnectIsEncrypted

The **virConnectIsEncrypted** API call can be used to find out if a given connection is encrypted. It takes a connection pointer as input, and if successful 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 **virConnectIsEncrypted**:

Example 3.19. Using virConnectIsEncrypted

```
/* example ex18.c */
/* compile with: gcc -g -Wall ex18.c -o ex18 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
```

```
    virConnectPtr conn;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    fprintf(stdout, "Connection is encrypted: %d\n", virConnectIsEncrypted(conn));

    virConnectClose(conn);
    return 0;
}
```

virConnectIsSecure

The **virConnectIsSecure** API call 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 eavsdropping (like a UNIX domain socket). It takes a connection pointer as input, and if successful 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 **virConnectIsSecure**:

Example 3.20. Using virConnectIsSecure

```
/* example ex19.c */
/* compile with: gcc -g -Wall ex19.c -o ex19 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    fprintf(stdout, "Connection is secure: %d\n", virConnectIsSecure(conn));

    virConnectClose(conn);
    return 0;
}
```

Event loop integration

The libvirt APIs use a basic request/response architecture that is generally synchronous. That is, a libvirt application calls a libvirt API (the request) which doesn't return until the action is complete (the response). However, a libvirtd server can also generate asynchronous messages and send them to the libvirt application; a typical usage of these messages is to inform the libvirt client when a domain has undergone a lifecycle change (like shutdown or restart).

The libvirt event loop APIs allow an application to register for these asynchrounous events and properly handle them.

The `virEventRegisterImpl` API registers a set of callbacks that libvirt will call when adding, updating, or removing a handle to watch.

Security model

The libvirt security model is known as svirt, and is designed to protect the host from guest domains, and guest domains from other guest domains. It works by putting each guest and guest disk into its own security domain, using whatever facilities are available to do so on the virtualization host. For the most part, svirt is invisible to application developers and need not be explored. There is a single libvirt API that provides information about the security model in use.

The **virNodeGetSecurityModel** API call can be used to find out the security model and DOI (Domain of Interpretation) in use on the virtualization host. It takes a connection pointer and a virSecurityModel pointer as input, and if successful fills in the virSecurityModel structure with the appropriate information. On success it returns 0; if an error occurred, -1 is returned. The following code demonstrates the use of **virNodeGetSecurityModel**:

Example 3.21. Using virNodeGetSecurityModel

```
/* example ex20.c */
/* compile with: gcc -g -Wall ex20.c -o ex20 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virSecurityModel secmodel;

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    virNodeGetSecurityModel(conn, &secmodel);

    fprintf(stdout, "Security Model: %s\n", secmodel.model);
    fprintf(stdout, "Security DOI:   %s\n", secmodel.doi);

    virConnectClose(conn);
    return 0;
}
```

Error handling

The libvirt error APIs are designed to give more detailed information about what caused a failure in the case that a normal libvirt API returned an error. An important thing to note about libvirt error reporting is that errors are stored per-connection *and* globally. This means that if multiple errors happen on a con-

nection without checking for the error, intermediate errors may be lost. For this reason, it is strongly recommended to always check for and collect errors immediately after the libvirt API failed. Alternatively, libvirt provides a way to set a custom error handling function that will be called synchronously at the time the error occurred.

All of the error functions deal with getting or clearing a `virError` structure. The `virError` structure looks like:

```
typedef struct _virError virError;
typedef virError *virErrorPtr;
struct _virError {
    int  code; /* The error code, a virErrorNumber */
    int  domain; /* What part of the library raised this error */
    char       *message;/* human-readable informative error message */
    virErrorLevel level;/* how consequent is the error */
    virConnectPtr conn VIR_DEPRECATED; /* connection if available, deprecated
                                          see note above */
    virDomainPtr dom VIR_DEPRECATED; /* domain if available, deprecated
                                        see note above */
    char       *str1; /* extra string information */
    char       *str2; /* extra string information */
    char       *str3; /* extra string information */
    int  int1; /* extra number information */
    int  int2; /* extra number information */
    virNetworkPtr net VIR_DEPRECATED; /* network if available, deprecated
                                         see note above */
};
```

(Note that conn, dom, and net are deprecated members and should not be used in new code, since they are not thread-safe)

The first member of the structure, "code", is the error code that was returned from the error. This is one of the members of the `virErrorNumber` enum in `virterror.h`; the full list of errors is:

```
typedef enum {
    VIR_ERR_OK = 0,
    VIR_ERR_INTERNAL_ERROR, /* internal error */
    VIR_ERR_NO_MEMORY,  /* memory allocation failure */
    VIR_ERR_NO_SUPPORT, /* no support for this function */
    VIR_ERR_UNKNOWN_HOST,/* could not resolve hostname */
    VIR_ERR_NO_CONNECT, /* can't connect to hypervisor */
    VIR_ERR_INVALID_CONN,/* invalid connection object */
    VIR_ERR_INVALID_DOMAIN,/* invalid domain object */
    VIR_ERR_INVALID_ARG,/* invalid function argument */
    VIR_ERR_OPERATION_FAILED,/* a command to hypervisor failed */
    VIR_ERR_GET_FAILED,/* a HTTP GET command to failed */
    VIR_ERR_POST_FAILED,/* a HTTP POST command to failed */
    VIR_ERR_HTTP_ERROR,/* unexpected HTTP error code */
    VIR_ERR_SEXPR_SERIAL,/* failure to serialize an S-Expr */
    VIR_ERR_NO_XEN,/* could not open Xen hypervisor control */
    VIR_ERR_XEN_CALL,/* failure doing an hypervisor call */
    VIR_ERR_OS_TYPE, /* unknown OS type */
    VIR_ERR_NO_KERNEL, /* missing kernel information */
    VIR_ERR_NO_ROOT, /* missing root device information */
    VIR_ERR_NO_SOURCE, /* missing source device information */
```

```
VIR_ERR_NO_TARGET, /* missing target device information */
VIR_ERR_NO_NAME, /* missing domain name information */
VIR_ERR_NO_OS, /* missing domain OS information */
VIR_ERR_NO_DEVICE, /* missing domain devices information */
VIR_ERR_NO_XENSTORE,/* could not open Xen Store control */
VIR_ERR_DRIVER_FULL, /* too many drivers registered */
VIR_ERR_CALL_FAILED, /* not supported by the drivers (DEPRECATED) */
VIR_ERR_XML_ERROR, /* an XML description is not well formed or broken */
VIR_ERR_DOM_EXIST,/* the domain already exist */
VIR_ERR_OPERATION_DENIED, /* operation forbidden on read-only connections */
VIR_ERR_OPEN_FAILED, /* failed to open a conf file */
VIR_ERR_READ_FAILED, /* failed to read a conf file */
VIR_ERR_PARSE_FAILED, /* failed to parse a conf file */
VIR_ERR_CONF_SYNTAX, /* failed to parse the syntax of a conf file */
VIR_ERR_WRITE_FAILED, /* failed to write a conf file */
VIR_ERR_XML_DETAIL, /* detail of an XML error */
VIR_ERR_INVALID_NETWORK, /* invalid network object */
VIR_ERR_NETWORK_EXIST, /* the network already exist */
VIR_ERR_SYSTEM_ERROR, /* general system call failure */
VIR_ERR_RPC, /* some sort of RPC error */
VIR_ERR_GNUTLS_ERROR, /* error from a GNUTLS call */
VIR_WAR_NO_NETWORK, /* failed to start network */
VIR_ERR_NO_DOMAIN, /* domain not found or unexpectedly disappeared */
VIR_ERR_NO_NETWORK, /* network not found */
VIR_ERR_INVALID_MAC, /* invalid MAC address */
VIR_ERR_AUTH_FAILED, /* authentication failed */
VIR_ERR_INVALID_STORAGE_POOL, /* invalid storage pool object */
VIR_ERR_INVALID_STORAGE_VOL, /* invalid storage vol object */
VIR_WAR_NO_STORAGE, /* failed to start storage */
VIR_ERR_NO_STORAGE_POOL, /* storage pool not found */
VIR_ERR_NO_STORAGE_VOL, /* storage pool not found */
VIR_WAR_NO_NODE, /* failed to start node driver */
VIR_ERR_INVALID_NODE_DEVICE,/* invalid node device object */
VIR_ERR_NO_NODE_DEVICE,/* node device not found */
VIR_ERR_NO_SECURITY_MODEL, /* security model not found */
VIR_ERR_OPERATION_INVALID, /* operation is not applicable at this time */
VIR_WAR_NO_INTERFACE, /* failed to start interface driver */
VIR_ERR_NO_INTERFACE, /* interface driver not running */
VIR_ERR_INVALID_INTERFACE, /* invalid interface object */
VIR_ERR_MULTIPLE_INTERFACES, /* more than one matching interface found */
VIR_WAR_NO_NWFILTER, /* failed to start nwfilter driver */
VIR_ERR_INVALID_NWFILTER, /* invalid nwfilter object */
VIR_ERR_NO_NWFILTER, /* nw filter pool not found */
VIR_ERR_BUILD_FIREWALL, /* nw filter pool not found */
VIR_WAR_NO_SECRET, /* failed to start secret storage */
VIR_ERR_INVALID_SECRET, /* invalid secret */
VIR_ERR_NO_SECRET, /* secret not found */
VIR_ERR_CONFIG_UNSUPPORTED, /* unsupported configuration construct */
VIR_ERR_OPERATION_TIMEOUT, /* timeout occurred during operation */
VIR_ERR_MIGRATE_PERSIST_FAILED, /* a migration worked, but making the
                                   VM persist on the dest host failed */
VIR_ERR_HOOK_SCRIPT_FAILED, /* a synchronous hook script failed */
VIR_ERR_INVALID_DOMAIN_SNAPSHOT, /* invalid domain snapshot */
VIR_ERR_NO_DOMAIN_SNAPSHOT, /* domain snapshot not found */
```

```
} virErrorNumber;
```

The second member of the structure, "domain", is named that for legacy reasons, but really represents which part of libvirt generated the error. This is one of the members of the `virErrorDomain` enum in `virterror.h`; the full list is:

```
typedef enum {
    VIR_FROM_NONE = 0,
    VIR_FROM_XEN, /* Error at Xen hypervisor layer */
    VIR_FROM_XEND, /* Error at connection with xend daemon */
    VIR_FROM_XENSTORE, /* Error at connection with xen store */
    VIR_FROM_SEXPR, /* Error in the S-Expression code */
    VIR_FROM_XML, /* Error in the XML code */
    VIR_FROM_DOM, /* Error when operating on a domain */
    VIR_FROM_RPC, /* Error in the XML-RPC code */
    VIR_FROM_PROXY, /* Error in the proxy code */
    VIR_FROM_CONF, /* Error in the configuration file handling */
    VIR_FROM_QEMU,      /* Error at the QEMU daemon */
    VIR_FROM_NET,       /* Error when operating on a network */
    VIR_FROM_TEST, /* Error from test driver */
    VIR_FROM_REMOTE, /* Error from remote driver */
    VIR_FROM_OPENVZ,    /* Error from OpenVZ driver */
    VIR_FROM_XENXM, /* Error at Xen XM layer */
    VIR_FROM_STATS_LINUX,/* Error in the Linux Stats code */
    VIR_FROM_LXC,       /* Error from Linux Container driver */
    VIR_FROM_STORAGE,   /* Error from storage driver */
    VIR_FROM_NETWORK,   /* Error from network config */
    VIR_FROM_DOMAIN,    /* Error from domain config */
    VIR_FROM_UML,       /* Error at the UML driver */
    VIR_FROM_NODEDEV,   /* Error from node device monitor */
    VIR_FROM_XEN_INOTIFY,/* Error from xen inotify layer */
    VIR_FROM_SECURITY,  /* Error from security framework */
    VIR_FROM_VBOX,      /* Error from VirtualBox driver */
    VIR_FROM_INTERFACE, /* Error when operating on an interface */
    VIR_FROM_ONE,       /* Error from OpenNebula driver */
    VIR_FROM_ESX,       /* Error from ESX driver */
    VIR_FROM_PHYP,      /* Error from IBM power hypervisor */
    VIR_FROM_SECRET,    /* Error from secret storage */
    VIR_FROM_CPU,       /* Error from CPU driver */
    VIR_FROM_XENAPI,    /* Error from XenAPI */
    VIR_FROM_NWFILTER,  /* Error from network filter driver */
    VIR_FROM_HOOK,      /* Error from Synchronous hooks */
    VIR_FROM_DOMAIN_SNAPSHOT, /* Error from domain snapshot */
} virErrorDomain;
```

The third member of the structure, "message", is a human-readable string describing the error.

The fourth member of the structure, "level", describes the severity of the error. This is one of the members of the virErrorLevel enum in virterror.h; the full list of levels is:

```
typedef enum {
    VIR_ERR_NONE = 0,
    VIR_ERR_WARNING = 1, /* A simple warning */
    VIR_ERR_ERROR = 2  /* An error */
} virErrorLevel;
```

The fifth member of the structure, "conn", is deprecated because it is not thread-safe.

The sixth member of the structure, "dom", is deprecated because it is not thread-safe.

The seventh member of the structure, "str1", gives extra human readable information.

The eighth member of the structure, "str2", gives extra human readable information.

The ninth member of the structure, "str3", gives extra human readable information.

The tenth member of the structure, "int1", gives extra numeric information that may be useful for further classifying the error.

The eleventh member of the structure, "int2", gives extra numeric information that may be useful for further classifying the error.

The twelfth member of the structure, "net", is deprecated because it is not thread-safe.

Example code that uses various parts of this structure will be presented in API call sub-sections.

virSetErrorFunc

By default when an error occurs, libvirt will call the `virDefaultErrorFunc` function which will print the error information to stderr. The `virSetErrorFunc` API call can be used to set a custom global error function that libvirt will call instead. It takes a void * pointer for userData and a virErrorFunc function pointer as input, and returns nothing. The custom error function should have a function signature of:

```
typedef void (*virErrorFunc) (void *userData, virErrorPtr error);
```

The following code demonstrates the use of `virSetErrorFunc`:

```
/* example ex21.c */
/* compile with: gcc -g -Wall ex21.c -o ex21 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

static void customErrorFunc(void *userdata, virErrorPtr err)
{
  fprintf(stderr, "Failure of libvirt library call:\n");
  fprintf(stderr, " Code: %d\n", err->code);
  fprintf(stderr, " Domain: %d\n", err->domain);
  fprintf(stderr, " Message: %s\n", err->message);
  fprintf(stderr, " Level: %d\n", err->level);
  fprintf(stderr, " str1: %s\n", err->str1);
  fprintf(stderr, " str2: %s\n", err->str2);
  fprintf(stderr, " str3: %s\n", err->str3);
  fprintf(stderr, " int1: %d\n", err->int1);
  fprintf(stderr, " int2: %d\n", err->int2);
}

int main(int argc, char *argv[])
{
    virConnectPtr conn;
```

```
    virSetErrorFunc(NULL, customErrorFunc);

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    if (virConnectGetVersion(conn, NULL) < 0)
        fprintf(stderr, "virConnectGetVersion failed\n");

    virConnectClose(conn);
    return 0;
}
```

virConnSetErrorFunc

The `virConnSetErrorFunc` API call can be used to set a per-connection custom error handling function. If present, this per-connection error handling function will take precendence over the global error handling function. The internal libvirt logic for deciding which error handling function to call is (in pseudo-code):

```
if (conn->handler)
    conn->handler;
else if (global_handler)
    global_handler;
else
    virDefaultErrorFunc;
```

`virConnSetErrorFunc` takes a virConnectPtr, a void * pointer for userData, and a virErrorFunc function pointer as input, and returns nothing. As with `virSetErrorFunc`, the custom connection error function should have a function signature of:

```
typedef void (*virErrorFunc) (void *userData, virErrorPtr error);
```

The following code demonstrates the use of `virConnSetErrorFunc`:

```
/* example ex22.c */
/* compile with: gcc -g -Wall ex22.c -o ex22 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

static void customConnErrorFunc(void *userdata, virErrorPtr err)
{
    fprintf(stderr, "Connection handler, failure of libvirt library call:\n");
    fprintf(stderr, " Code: %d\n", err->code);
    fprintf(stderr, " Domain: %d\n", err->domain);
    fprintf(stderr, " Message: %s\n", err->message);
    fprintf(stderr, " Level: %d\n", err->level);
    fprintf(stderr, " str1: %s\n", err->str1);
    fprintf(stderr, " str2: %s\n", err->str2);
```

```
    fprintf(stderr, " str3: %s\n", err->str3);
    fprintf(stderr, " int1: %d\n", err->int1);
    fprintf(stderr, " int2: %d\n", err->int2);
}

static void customGlobalErrorFunc(void *userdata, virErrorPtr err)
{
    fprintf(stderr, "Global handler, failure of libvirt library call:\n");
    fprintf(stderr, " Code: %d\n", err->code);
    fprintf(stderr, " Domain: %d\n", err->domain);
    fprintf(stderr, " Message: %s\n", err->message);
    fprintf(stderr, " Level: %d\n", err->level);
    fprintf(stderr, " str1: %s\n", err->str1);
    fprintf(stderr, " str2: %s\n", err->str2);
    fprintf(stderr, " str3: %s\n", err->str3);
    fprintf(stderr, " int1: %d\n", err->int1);
    fprintf(stderr, " int2: %d\n", err->int2);
}

int main(int argc, char *argv[])
{
    virConnectPtr conn1;
    virConnectPtr conn2;

    /* set a global error function for all connections */
    virSetErrorFunc(NULL, customGlobalErrorFunc);

    conn1 = virConnectOpen("qemu:///system");
    if (conn1 == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    conn2 = virConnectOpen("qemu:///system");
    if (conn2 == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        virConnectClose(conn1);
        return 2;
    }

    /* conn1 will use a different error function */
    virConnSetErrorFunc(conn1, NULL, customConnErrorFunc);

    /* this failure will use customConnErrorFunc */
    if (virConnectGetVersion(conn1, NULL) < 0)
        fprintf(stderr, "virConnectGetVersion failed\n");

    /* this failure will use customGlobalErrorFunc */
    if (virConnectGetVersion(conn2, NULL) < 0)
        fprintf(stderr, "virConnectGetVersion failed\n");

    virConnectClose(conn2);
    virConnectClose(conn1);
    return 0;
```

```
}
```

virCopyLastError

The `virCopyLastError` API call can be used to obtain a copy of the last error reported from libvirt. The error object is kept in thread local storage so separate threads can safely use this function concurrently. It takes a virErrorPtr as input, and if successful makes a deep copy of the error. If there were no errors pending, 0 is returned. If there was an error pending, the error code is returned. If an error occurred, -1 is returned. The following code demonstrates the use of `virCopyLastError`:

```
/* example ex23.c */
/* compile with: gcc -g -Wall ex23.c -o ex23 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

/* dummy error function to suppress virDefaultErrorFunc */
static void customErrorFunc(void *userdata, virErrorPtr err)
{
}

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virError err;

    virSetErrorFunc(NULL, customErrorFunc);

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    if (virConnectGetVersion(conn, NULL) < 0) {
        virCopyLastError(&err);
        fprintf(stderr, "virConnectGetVersion failed: %s\n", err.message);
        virResetError(&err);
    }

    virConnectClose(conn);
    return 0;
}
```

virGetLastError

The `virGetLastError` API call can be used to obtain a pointer to the last error reported from libvirt. The error object is kept in thread local storage so separate threads can safely use this function concurrently. Note that it does not take a copy, so error information can be lost if the current thread obtains this pointer, calls another libvirt function, and then tries to access this pointer. If that behavior is desired, use `virCopyLastError` or `virSaveLastError` instead. It takes no input, and returns a pointer to the last error object if one exists. If there was no last error, or the last error was VIR_ERR_OK, NULL is returned instead. The following code demonstrates the use of `virGetLastError`:

```
/* example ex24.c */
/* compile with: gcc -g -Wall ex24.c -o ex24 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

static void customErrorFunc(void *userdata, virErrorPtr err)
{
}

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virErrorPtr err;

    virSetErrorFunc(NULL, customErrorFunc);

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    if (virConnectGetVersion(conn, NULL) < 0) {
      /* this is a valid way to use virGetLastError */
      err = virGetLastError();
      fprintf(stderr, "virConnectGetVersion failed: %s\n", err->message);
    }

    if (virConnectGetVersion(conn, NULL) < 0) {
      /* this is an invalid way to use virGetLastError; the error message will
       * not represent the error from virConnectGetVersion
       */
      err = virGetLastError();
      virNodeGetFreeMemory(NULL);
      fprintf(stderr, "virConnectGetVersion failed: %s\n", err->message);
    }

    virConnectClose(conn);
    return 0;
}
```

virSaveLastError

The `virSaveLastError` API call can be used to allocate and obtain a copy of the last error reported from libvirt. The error object is kept in thread local storage so separate threads can safely use this function concurrently. It takes nothing as input and if successful returns a newly allocated virError object. It is the responsibility of the caller to free the error with `virFreeError`. If an error occurred, NULL is returned instead. The following code demonstrates the use of `virSaveLastError`:

```
/* example ex25.c */
/* compile with: gcc -g -Wall ex25.c -o ex25 -lvirt */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

/* dummy error function to suppress virDefaultErrorFunc */
static void customErrorFunc(void *userdata, virErrorPtr err)
{
}

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virErrorPtr err;

    virSetErrorFunc(NULL, customErrorFunc);

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    if (virConnectGetVersion(conn, NULL) < 0) {
        err = virSaveLastError();
        fprintf(stderr, "virConnectGetVersion failed: %s\n", err->message);
        virFreeError(err);
    }

    virConnectClose(conn);
    return 0;
}
```

virResetError

The `virResetError` API call can be used to clear and free any memory associated with an virError object (though it does not free the object itself). It is typically used after a program is finished using an virError object obtained through `virCopyLastError`, though it can also be used after a `virSaveLastError` in special circumstances. It takes the virErrorPtr as input, and returns nothing. The following code demonstrates the use of `virResetError`:

```
/* example ex26.c */
/* compile with: gcc -g -Wall ex26.c -o ex26 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

/* dummy error function to suppress virDefaultErrorFunc */
static void customErrorFunc(void *userdata, virErrorPtr err)
{
}

int main(int argc, char *argv[])
{
```

```
    virConnectPtr conn;
    virError err;

    virSetErrorFunc(NULL, customErrorFunc);

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }

    if (virConnectGetVersion(conn, NULL) < 0) {
        virCopyLastError(&err);
        fprintf(stderr, "virConnectGetVersion failed: %s\n", err.message);
        virResetError(&err);
    }

    virConnectClose(conn);
    return 0;
}
```

virFreeError

The `virFreeError` API call can be used to clear and free any memory associated with an virError object, including the object itself. It is typically used after a program is finished using an virError object obtained through `virSaveLastError`, though it can also be used after a `virCopyLastError` in special circumstances. It takes the virErrorPtr as input, and returns nothing. The following code demonstrates the use of `virFreeError`:

```
/* example ex27.c */
/* compile with: gcc -g -Wall ex27.c -o ex27 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

/* dummy error function to suppress virDefaultErrorFunc */
static void customErrorFunc(void *userdata, virErrorPtr err)
{
}

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virErrorPtr err;

    virSetErrorFunc(NULL, customErrorFunc);

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        return 1;
    }
```

```
    if (virConnectGetVersion(conn, NULL) < 0) {
        err = virSaveLastError();
        fprintf(stderr, "virConnectGetVersion failed: %s\n", err->message);
        virFreeError(err);
    }

    virConnectClose(conn);
    return 0;
}
```

virConnResetError

The `virConnResetError` API call can be used to clear and free any memory associated with an virError object on a particular connection (though it does not free the object itself). It is typically used after a program is finished using an virError object obtained through `virConnCopyLastError`, though it can also be used after a `virConnSaveLastError` in special circumstances. However, both `virConnCopyLastError` and `virConnSaveLastError` are deprecated, and since libvirt 0.6.0 all per-connection errors are also propagated to the global error storage. Therefore, this API should not be used in new code; `virResetError` should be used instead. The function remains for backwards compatibility.

virConnCopyLastError

The `virConnCopyLastError` API call can be used to obtain a copy of the last error reported from libvirt on a particular connection. This function has been deprecated because it is not thread-safe, and since libvirt 0.6.0 all connection errors are also propagated to the global error object. New code should use `virCopyLastError` instead. This function remains for backwards compatilibity.

virConnGetLastError

The `virConnGetLastError` API call can be used to obtain a pointer to the last error reported from libvirt on a particular connection. This function has been deprecated because it is not thread-safe, and since libvirt 0.6.0 all connection errors are also propagated to the global error object. New code should use `virGetLastError` instead. This function remains for backwards compatilibity.

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
```

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.

Example 3.22. Running virsh with environment variables

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.

Integrated example

This example demonstrates many of the concepts from the chapter together, including error checking. While still not a "real" program (which would likely be multi-threaded), it's a good example of how to write a libvirt program from end to end.

```
/* example ex28.c */
/* compile with: gcc -g -Wall ex28.c -o ex28 -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <libvirt/libvirt.h>
#include <libvirt/virterror.h>

static void customConnErrorFunc(void *userdata, virErrorPtr err)
{
    fprintf(stderr, "Connection handler, failure of libvirt library call:\n");
    fprintf(stderr, " Code: %d\n", err->code);
    fprintf(stderr, " Domain: %d\n", err->domain);
    fprintf(stderr, " Message: %s\n", err->message);
    fprintf(stderr, " Level: %d\n", err->level);
    fprintf(stderr, " str1: %s\n", err->str1);
    fprintf(stderr, " str2: %s\n", err->str2);
    fprintf(stderr, " str3: %s\n", err->str3);
    fprintf(stderr, " int1: %d\n", err->int1);
    fprintf(stderr, " int2: %d\n", err->int2);
}

static void customGlobalErrorFunc(void *userdata, virErrorPtr err)
{
    fprintf(stderr, "Global handler, failure of libvirt library call:\n");
    fprintf(stderr, " Code: %d\n", err->code);
    fprintf(stderr, " Domain: %d\n", err->domain);
    fprintf(stderr, " Message: %s\n", err->message);
    fprintf(stderr, " Level: %d\n", err->level);
    fprintf(stderr, " str1: %s\n", err->str1);
    fprintf(stderr, " str2: %s\n", err->str2);
    fprintf(stderr, " str3: %s\n", err->str3);
    fprintf(stderr, " int1: %d\n", err->int1);
    fprintf(stderr, " int2: %d\n", err->int2);
}

int main(int argc, char *argv[])
{
    virConnectPtr conn1;
    virConnectPtr conn2;
    virConnectPtr conn3;
    virConnectPtr conn4;
    char *caps;
    virError err;
    char *hostname;
    virErrorPtr err2;
    int vcpus;
    unsigned long long node_free_memory;
```

```
virNodeInfo nodeinfo;
unsigned long long *node_cells_freemem;
int numnodes;
int i;
const char *type;
unsigned long virtVersion;
unsigned long libvirtVersion;
char *uri;
int is_encrypted;
int is_secure;
virSecurityModel secmodel;

/* set a global error function for all connections */
virSetErrorFunc(NULL, customGlobalErrorFunc);

/* open a connection using the old-style virConnectOpen */
conn1 = virConnectOpen("qemu:///system");
if (conn1 == NULL) {
    fprintf(stderr, "Failed to open connection to qemu:///system\n");
    return 1;
}

/* open a read-only connection using the old-style virConnectOpenReadOnly */
conn2 = virConnectOpenReadOnly("qemu:///system");
if (conn2 == NULL) {
    fprintf(stderr, "Failed to open connection to qemu:///system\n");
    virConnectClose(conn1);
    return 2;
}

/* open a connection using the new-style virConnectOpenAuth */
conn3 = virConnectOpenAuth("qemu:///system", virConnectAuthPtrDefault, 0);
if (conn3 == NULL) {
    fprintf(stderr, "Failed to open connection to qemu:///system\n");
    virConnectClose(conn2);
    virConnectClose(conn1);
    return 3;
}

/* open a read-only connection using the new-style virConnectOpenAuth */
conn4 = virConnectOpenAuth("qemu:///system", virConnectAuthPtrDefault,
                           VIR_CONNECT_RO);
if (conn4 == NULL) {
    fprintf(stderr, "Failed to open connection to qemu:///system\n");
    virConnectClose(conn3);
    virConnectClose(conn2);
    virConnectClose(conn1);
    return 3;
}

/* conn1 will use a different error function */
virConnSetErrorFunc(conn1, NULL, customConnErrorFunc);

/* test out error handling */
```

```
/* this failure will use customConnErrorFunc */
if (virConnectGetVersion(conn1, NULL) < 0)
    fprintf(stderr, "virConnectGetVersion failed\n");
/* this failure will use customGlobalErrorFunc */
if (virConnectGetVersion(conn2, NULL) < 0)
    fprintf(stderr, "virConnectGetVersion failed\n");

/* clear out the per-connection error function; we will report errors with
 * the vir*Error() functions below*/
virConnSetErrorFunc(conn1, NULL, NULL);

/* clear out the global error function; we will report errors with
 * the vir*Error() functions below*/
virSetErrorFunc(NULL, NULL);

/* get the capabilities of conn1 */
caps = virConnectGetCapabilities(conn1);
if (caps == NULL) {
    virCopyLastError(&err);
    fprintf(stderr, "virConnectGetCapabilities failed: %s\n", err.message);
    virResetError(&err);
}
fprintf(stdout, "Capabilities of connection 1:\n%s\n", caps);
free(caps);

/* get the hostname reported from conn2 */
hostname = virConnectGetHostname(conn2);
if (hostname == NULL) {
    err2 = virSaveLastError();
    fprintf(stderr, "virConnectGetVersion failed: %s\n", err2->message);
    virFreeError(err2);
}
fprintf(stdout, "Connection 2 hostname: %s\n", hostname);
free(hostname);

/* get the maximum number of vcpus supported by conn3 */
vcpus = virConnectGetMaxVcpus(conn3, NULL);
if (vcpus < 0) {
    err2 = virSaveLastError();
    fprintf(stderr, "virConnectGetMaxVcpus failed: %s\n", err2->message);
    virFreeError(err2);
}
fprintf(stdout, "Maximum number of cpus supported on connection 3: %d\n",
        vcpus);

/* get the amount of free memory available on the node from conn4 */
node_free_memory = virNodeGetFreeMemory(conn4);
if (node_free_memory == 0) {
    err2 = virSaveLastError();
    fprintf(stderr, "virNodeGetFreeMemory failed: %s\n", err2->message);
    virFreeError(err2);
}

/* get the node information from conn1 */
```

```
if (virNodeGetInfo(conn1, &nodeinfo) < 0) {
    err2 = virSaveLastError();
    fprintf(stderr, "virNodeGetInfo failed: %s\n", err2->message);
    virFreeError(err2);
}
fprintf(stdout, "Node information from connection 1:\n");
fprintf(stdout, " Model: %s\n", nodeinfo.model);
fprintf(stdout, " Memory size: %lukb\n", nodeinfo.memory);
fprintf(stdout, " Number of CPUs: %u\n", nodeinfo.cpus);
fprintf(stdout, " MHz of CPUs: %u\n", nodeinfo.mhz);
fprintf(stdout, " Number of NUMA nodes: %u\n", nodeinfo.nodes);
fprintf(stdout, " Number of CPU sockets: %u\n", nodeinfo.sockets);
fprintf(stdout, " Number of CPU cores per socket: %u\n", nodeinfo.cores);
fprintf(stdout, " Number of CPU threads per core: %u\n", nodeinfo.threads);

/* get the amount of memory in each of the NUMA nodes from connection 1 */
/* we already know the number of nodes from virNodeGetInfo above */
node_cells_freemem = malloc(nodeinfo.nodes * sizeof(unsigned long long));
numnodes = virNodeGetCellsFreeMemory(conn1, node_cells_freemem, 0,
                                     nodeinfo.nodes);
if (numnodes < 0) {
    err2 = virSaveLastError();
    fprintf(stderr, "virNodeGetCellsFreeMemory failed: %s\n",
            err2->message);
    virFreeError(err2);
}
fprintf(stdout, "Node Cells Free Memory from connection 1:\n");
for (i = 0; i < numnodes; i++)
 fprintf(stdout, " Cell %d: %llukb free memory\n", i,
            node_cells_freemem[i]);
free(node_cells_freemem);

/* get the virtualization type from conn2 */
type = virConnectGetType(conn2);
if (type == NULL) {
    err2 = virSaveLastError();
    fprintf(stderr, "virConnectGetType failed: %s\n", err2->message);
    virFreeError(err2);
}
fprintf(stdout, "Virtualization type from connection 2: %s\n", type);

/* get the virtualization version from conn3 */
if (virConnectGetVersion(conn3, &virtVersion) < 0) {
    err2 = virSaveLastError();
    fprintf(stderr, "virConnectGetVersion failed: %s\n", err2->message);
    virFreeError(err2);
}
fprintf(stdout, "Virtualization version from connection 3: %lu\n",
        virtVersion);

/* get the libvirt version from conn4 */
if (virConnectGetLibVersion(conn4, &libvirtVersion) < 0) {
    err2 = virSaveLastError();
    fprintf(stderr, "virConnectGetLibVersion failed: %s\n", err2->message);
```

```
        virFreeError(err2);
    }
    fprintf(stdout, "Libvirt version from connection 4: %lu\n", libvirtVersion);

    /* get the URI from conn1 */
    uri = virConnectGetURI(conn1);
    if (uri == NULL) {
        err2 = virSaveLastError();
        fprintf(stderr, "virConnectGetURI failed: %s\n", err2->message);
        virFreeError(err2);
    }
    fprintf(stdout, "Canonical URI from connection 1: %s\n", uri);
    free(uri);

    /* is the connection encrypted? from conn2 */
    is_encrypted = virConnectIsEncrypted(conn2);
    if (is_encrypted < 0) {
        err2 = virSaveLastError();
        fprintf(stderr, "virConnectIsEncrypted failed: %s\n", err2->message);
        virFreeError(err2);
    }
    fprintf(stdout, "Connection 2 %s encrypted\n",
            (is_encrypted == 0) ? "is not" : "is");

    /* is the connection secure? from conn3 */
    is_secure = virConnectIsSecure(conn3);
    if (is_secure < 0) {
        err2 = virSaveLastError();
        fprintf(stderr, "virConnectIsSecure failed: %s\n", err2->message);
        virFreeError(err2);
    }
    fprintf(stdout, "Connection 3 %s secure\n",
            (is_secure == 0) ? "is not" : "is");

    /* get the security model from conn4 */
    if (virNodeGetSecurityModel(conn4, &secmodel) < 0) {
        err2 = virSaveLastError();
        fprintf(stderr, "virNodeGetSecurityModel failed: %s\n", err2->message);
        virFreeError(err2);
    }
    fprintf(stdout, "Connection 4 Security Model = %s, DOI = %s\n",
            secmodel.model, secmodel.doi);

    virConnectClose(conn4);
    virConnectClose(conn3);
    virConnectClose(conn2);
    virConnectClose(conn1);
    return 0;
}
```

Chapter 4. Guest Domains

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 APIs 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 a 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 `virDomainPtr` object. There are three, imaginatively named, methods to do lookup existing domains, `virDomainLookupByID`, `virDomainLookupByName` and `virDomainLookupByUUID`. Each of these takes a connection object as first parameter, and the domain identifier as the other. They will return NULL if no matching domain exists. The connection's error object can be queried to find specific details of the error if required.

Example 4.1. Fetching a domain object from an ID

```
int domainID = 6;
virDomainPtr dom;

dom = virDomainLookupByID(conn, domainID);
```

Example 4.2. Fetching a domain object from an name

```
char *domainName = "someguest";
virDomainPtr dom;
```

```
dom = virDomainLookupByName(conn, domainName);
```

Example 4.3. Fetching a domain object from an UUID

```
char *domainUUID = "00311636-7767-71d2-e94a-26e7b8bad250";
virDomainPtr dom;

dom = virDomainLookupByUUIDString(conn, domainUUID);
```

For convenience of this document, the UUID example used the printable format of UUID. There is an equivalent method which accepts the raw bytes `unsigned char[]`

Listing domains

The libvirt API exposes 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 API described later in this section provides a way to track all lifecycle changes avoiding this potential race condition.

The API 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 API for listing active domains, `virConnectListDomains`, requires the caller to pass in a pre-allocated `int` array which will be filled in domain IDs. The return value will be -1 upon error, or the total number of array elements filled. To determine how large to make the ID array, the application can use the API call `virConnectNumOfDomains`. Putting these two calls together, a fragment of code which prints a list running domain IDs would be

Example 4.4. Listing active domains

```
int i;
int numDomains;
int *activeDomains;

numDomains = virConnectNumOfDomains(conn);

activeDomains = malloc(sizeof(int) * numDomains);
numDomains = virConnectListDomains(conn, activeDomains, numDomains);

printf("Active domain IDs:\n");
for (i = 0 ; i < numDomains ; i++) {
    printf("  %d\n", activeDomains[i]);
}
free(activeDomains);
```

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. In a similar style to the API just discussed, the `virConnectListDefinedDomains` API requires the caller to provide a pre-allocated `char *` array which will be filled with domain name strings. The return value will be -1 upon error, or the total number of array elements filled with names. It is the caller's responsibility to free the memory associated with each returned name. As you might expect, there is also a `virConnectNumOfDefinedDomains` API to determine how

many names are known. Putting these calls together, a fragment of code which prints a list of inactive persistent domain names would be:

Example 4.5. Listing inactive domains

```
int i;
int numDomains;
char **inactiveDomains;

numDomains = virConnectNumOfDefinedDomains(conn);

inactiveDomains = malloc(sizeof(char *) * numDomains);
numDomains = virConnectListDefinedDomains(conn, inactiveDomains, numDomains);

printf("Inactive domain names:\n");
for (i = 0 ; i < numDomains ; i++) {
    printf("  %s\n", inactiveDomains[i]);
    free(inactiveDomains[i]);
}
free(inactiveDomains);
```

The APIs for listing domains do not directly return the full `virDomainPtr` objects, since this may incur undue performance penalty for applications which wish to query the list of domains on a frequent basis. Given a domain ID or name, obtaining a full `virDomainPtr` object is a straightforward matter of calling one of the `virDomainLookupBy{Name,ID}` methods. So an example which obtained a `virDomainPtr` object for every domain, both active and inactive, would be:

Example 4.6. Fetching all domain objects

```
virDomainPtr *allDomains;
int numDomains = 0;
int numActiveDomains, numInactiveDomains;
char **inactiveDomains;
int *activeDomains;
int i;

numActiveDomains = virConnectNumOfDomains(conn);
numInactiveDomains = virConnectNumOfDefinedDomains(conn);

allDomains = malloc(sizeof(virDomainPtr) *
    (numActiveDomains + numInactiveDomains));
inactiveDomains = malloc(sizeof(char *) * numInactiveDomains);
activeDomains = malloc(sizeof(int) * numActiveDomains);

numActiveDomains = virConnectListDomains(conn,
    activeDomains,
    numActiveDomains);
numInactiveDomains = virConnectListDefinedDomains(conn,
    inactiveDomains,
    numInactiveDomains);

for (i = 0 ; i < numActiveDomains ; i++) {
```

```
    allDomains[numDomains] = virDomainLookupByID(conn, activeDomains[i]);
    numDomains++;
}

for (i = 0 ; i < numInactiveDomains ; i++) {
    allDomains[numDomains] = virDomainLookupByName(conn, inactiveDomains[i]);
    free(inactiveDomains[i]);
    numDomains++;
}
free(activeDomains);
free(inactiveDomains);
```

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: the section called “Provisioning and starting”, the section called “Suspend / Resume and Save / Restore”, the section called “Migration” and the section called “Autostart”.

Figure 4.1. Guest domain lifecycle

Provisioning and starting

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

APIs for provisioning

There are up to three APIs involved in provisioning guests. The `virDomainCreateXML` command will create and immediately boot a new transient guest domain. When this guest domain shuts down, all trace of it will disappear. The `virDomainDefineXML` command will store the configuration for a persistent guest domain. The `virDomainCreate` command will boot a previously defined guest domain from its

persistent configuration. One important thing to note, is that the `virDomainDefineXML` command can be used to turn a previously booted transient guest domain, into a persistent domain. This can be useful for some provisioning scenarios that will be illustrated later.

Booting a transient guest domain

To boot a transient guest domain, simply requires a connection to libvirt and a string containing the XML document describing the required guest configuration. The following example assumes that `conn` is an instance of the `virConnectPtr` object.

```
virDomainPtr dom;
const char *xmlconfig = "<domain>........</domain>";

dom = virConnectCreateXML(conn, xmlconfig, 0);

if (!dom) {
    fprintf(stderr, "Domain creation failed");
    return;
}

fprintf(stderr, "Guest %s has booted", virDomainName(dom));
virDomainFree(dom);
```

If the domain creation attempt succeeded, then the returned `virDomainPtr` will be a handle to the guest domain. This must be released later when no longer needed by using the `virDomainFree` method. 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 `virDomainPtr` 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.

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 `virDomainPtr` object obtained from defining the guest, can then be used to boot it. The following example assumes that `conn` is an instance of the `virConnectPtr` object.

```
virDomainPtr dom;
const char *xmlconfig = "<domain>........</domain>";

dom = virConnectDefineXML(conn, xmlconfig, 0);

if (!dom) {
    fprintf(stderr, "Domain definition failed");
    return;
}

if (virDomainCreate(dom) < 0) {
    virDomainFree(dom);
    fprintf(stderr, "Cannot boot guest");
```

```
    return;
}

fprintf(stderr, "Guest %s has booted", virDomainName(dom));
virDomainFree(dom);
```

New guest 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 a 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.

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.

```
const char *xml = "<domain>....</domain>";
virDomainPtr dom;

dom = virDomainDefineXML(conn, xml);
if (!dom) {
    fprintf(stderr, "Unable to define persistent guest configuration");
    return;
}

if (virDomainCreate(dom) < 0) {
    fprintf(stderr, "Unable to boot guest configuration");
}
```

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.

PXE boot provisioning

Some newer full virtualization technologies provide a BIOS that is able to use the PXE boot protocol to boot of 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.

```
const char *xml = "<domain>....</domain>";
virDomainPtr dom;

dom = virDomainDefineXML(conn, xml);
if (!dom) {
    fprintf(stderr, "Unable to define persistent guest configuration");
    return;
}

if (virDomainCreate(dom) < 0) {
    fprintf(stderr, "Unable to boot guest configuration");
}
```

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.

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/x
</os>
```

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

```
const char *xml = "<domain>....</domain>";
virDomainPtr dom;

dom = virDomainCreateXML(conn, xml);
if (!dom) {
    fprintf(stderr, "Unable to boot transient guest configuration");
    return;
}
```

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

```
const char *xml = "<domain>....</domain>";
virDomainPtr dom;

dom = virDomainCreateXML(conn, xml);
if (!dom) {
    fprintf(stderr, "Unable to define persistent guest configuration\n");
    return;
}

if (virDomainCreate(dom) < 0) {
    fprintf(stderr, "Unable to boot persistent guest\n");
    return;
}

fprintf(stderr, "Guest provisoning complete, OS is running\n");
```

Stopping

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

Shutdown 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 successfuly shut down. The shutdown process is analagous to running a shutdown command on a physical machine.

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

Suspend / Resume and Save / Restore

Suspend and resume refers 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, contiuining execution where it left off. Suspend does not save a persistent image of the guest's memory. For this, save is used.

Save and restore refers 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 APIs 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 API 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.

```
virDomainPtr dom;
virDomainInfoPtr info;
const char *filename = "/var/lib/libvirt/save/demo-guest.img";

dom = virDomainLookupByName(conn, "demo-guest");
if (!dom) {
    fprintf(stderr, "Cannot find guest to be saved");
    return;
}

if (virDomainGetInfo(dom, &amp;info) < 0) {
    fprintf(stderr, "Cannot check guest state");
    return;
}

if (info.state == VIR_DOMAIN_SHUTOFF) {
    fprintf(stderr, "Not saving guest that isn't running");
    return;
}

if (virDomainSave(dom, filename) < 0) {
    fprintf(stderr, "Unable to save guest to %s", filename);
}

fprintf(stderr, "Guest state saved to %s", filename);
```

Some period of time later, the saved state file can then be used to restart the guest where it left of, using the virDomainRestore API. 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

```
virDomainPtr dom;
int id;
const char *filename = "/var/lib/libvirt/save/demo-guest.img";

if ((id = virDomainRestore(conn, filename)) < 0) {
    fprintf(stderr, "Unable to restore guest from %s", filename);
}

dom = virDomainLookupByID(conn, id);
if (!dom) {
    fprintf(stderr, "Cannot find guest that was restored");
    return;
}

fprintf(stderr, "Guest state restored from %s", filename);
```

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 APIs for migration. The `virDomainMigrate` command takes an established hypervisor connection, and instructs the domain to migrate to this connection. The `virMigrateToUri` command takes a URI specifying a hypervisor connection, opens the connection, then instructions the domain to migrate to this connection. Both these commands 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.

TODO: Add 2 cold examples, 1 live example.

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.

TODO: code example.

Domain configuration

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

Boot modes

TBD

Memory / CPU resources

TBD. Maps to the basic resources section.

Lifecycle controls

TBD

Clock sync

TBD

Features

TBD

Monitoring performance

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

Domain performance

TBD

vCPU performance

TBD

I/O statistics

TBD

Device configuration

TBD

Emulator

TBD

Disks

TBD

Networking

TBD

Filesystems

TBD

Mice & tablets

TBD

USB device passthrough

TBD

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. The section documenting node device APIs covers this topic in detail, but as a quick demonstration the following code checks whether a PCI device (represented by a virNodeDevicePtr object instance) can be reset and is thus assignable to a guest

```
virNodeDevicePtr dev = ....get virNodeDevicePtr for the PCI device...

if (virNodeDeviceDettach(dev) < 0) {
    fprintf(stderr, "Device cannot be dettached from the host OS drivers\n");
    return;
}

if (virNodeDeviceReset(dev) < 0) {
    fprintf(stderr, "Device cannot be safely reset without affecting other devices
    return;
}
```

```
fprintf(stderr, "Device is suitable for passthrough to a guest\n");
```

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>
```

Live configuration change

TBD

Memory ballooning

TBD

CPU hotplug

TBD

Device hotplug / unplug

TBD

Device media change

TBD

Block Device Jobs

Libvirt provides a generic Block Job API 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. `virDomainBlockPull`). 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. `virDomainBlockPull()` 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 `virDomainGetBlockJobInfo()`. 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 `virDomainBlockJobSetSpeed()` to limit the amount of bandwidth that a block job may consume. Bandwidth is specified in units of MB/sec.

When a block job operation completes, the final status is reported using an asynchronous event. To receive this event, register a `virConnectDomainEventBlockJobCallback` function which will receive the disk, event type, and status as parameters.

```
/* example blockpull-example.c */
/* compile with: gcc -g -Wall blockpull-example.c -o blockpull-example -lvirt */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libvirt/libvirt.h>

int do_cmd(const char *cmdline)
{
    int status = system(cmdline);
    if (status < 0)
        return -1;
    else
        return WEXITSTATUS(status);
}

virDomainPtr make_domain(virConnectPtr conn)
{
    virDomainPtr dom;
    char 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>";
```

```
    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 = virDomainCreateXML(conn, domxml, 0);
    return dom;
}

int main(int argc, char *argv[])
{
    virConnectPtr conn;
    virDomainPtr dom = NULL;
    char disk[] = "/var/lib/libvirt/images/example.qed";

    conn = virConnectOpen("qemu:///system");
    if (conn == NULL) {
        fprintf(stderr, "Failed to open connection to qemu:///system\n");
        goto error;
    }

    dom = make_domain(conn);
    if (dom == NULL) {
        fprintf(stderr, "Failed to create domain\n");
        goto error;
    }

    if ((virDomainBlockPull(dom, disk, 0, 0)) < 0) {
        fprintf(stderr, "Failed to start block pull");
        goto error;
    }

    while (1) {
        virDomainBlockJobInfo info;
        int ret = virDomainGetBlockJobInfo(dom, disk, &info, 0);

        if (ret == 1) {
            printf("BlockPull progress: %0.0f %%\n",
                (float)(100 * info.cur / info.end));
        } else if (ret == 0) {
            printf("BlockPull complete\n");
            break;
        } else {
            fprintf(stderr, "Failed to query block jobs\n");
            break;
        }
        usleep(100000);
    }

error:
    unlink("/var/lib/libvirt/images/backing.qed");
    unlink("/var/lib/libvirt/images/example.qed");
    if (dom != NULL) {
        virDomainDestroy(dom);
        virDomainFree(dom);
```

```
    }
    if (conn != NULL)
        virConnectClose(conn);
    return 0;
}
```

Security model

TBD

Event notifications

TBD

Tuning

TBD

Scheduler parameters

TBD

NUMA placement

TBD

Chapter 5. Storage Pools

This is a test paragraph

Overview

TBD

Listing pools

TBD

Pool usage

TBD

Lifecycle control

TBD

Discovering pool sources

TBD

Pool configuration

TBD

Volume overview

TBD

Listing volumes

TBD

Volume information

TBD

Creating and deleting volumes

TBD

Cloning volumes

TBD

Configuring volumes

TBD

Chapter 6. Virtual Networks

TBD

Overview

TBD

Listing networks

TBD

Lifecycle control

TBD

Network configuration

TBD

Chapter 7. Network Interfaces

This section covers the management of physical network interfaces using the libvirt API.

Overview

The configuration of network interfaces on physical hosts can be examined and modified with functions in the `virInterface` API. This is useful for setting up the host to share one physical interface bewteen 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, this API can also be used to configure bridges, bonded interfaces, and vlan interfaces.

The `virInterface` API 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` API described in Chapter 6, *Virtual Networks*.

Each host interface is represented in the API by a `virInterfacePtr` - a pointer to an "interface object" - and each of these has a single unique identifier:

`name`: 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:

`mac`: 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. However, there are API functions to start and stop interfaces, and those actions cause the new configuration to be applied to the interface immediately.

When a new interface is defined (with `virInterfaceDefineXML`), or the configuration of an existing interface is changed (again, with `virInterfaceDefineXML`), this configuration will be stored on the host. The live configuration of the interface itself will not be changed until either the interface is started (with `virInterfaceCreate`), or the host is rebooted.

XML Interface Description Format

The current Relax NG definition of the XML that is produced and accepted by `virInterfaceDefineXML` and `virInterfaceGetXMLDesc` 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.

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'>
    <dhcp/>
  </protocol>
</interface>
```

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

```
<interface type='ethernet' name='eth0'>
  <start mode='onboot'/>
  <mac address='aa:bb:cc:dd:ee:ff'/>
  <protocol family='ipv4'>
    <ip address="192.168.0.5" prefix="24"/>
    <route gateway="192.168.0.1"/>
  </protocol>
</interface>
```

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

```
<interface type="bridge" name="br0">
  <start mode="onboot"/>
  <mtu size="1500"/>
  <protocol family="ipv4">
    <dhcp/>
  </protocol>
  <bridge stp="off" delay="0.01">
    <interface type="ethernet" name="eth0">
      <mac address="ab:bb:cc:dd:ee:ff"/>
    </interface>
    <interface type="ethernet" name="eth1"/>
  </bridge>
</interface>
```

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

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

Retrieving Information About Interfaces

Enumerating Interfaces

Once you have a connection to a host, represented by a `virConnectPtr`, you can determine the number of interfaces on the host with `virConnectNumOfInterfaces` and `virConnectNumOfDe-`

finedInterfaces. A list of those interfaces' names can be obtained with virConnectListInterfaces and virConnectListDefinedInterfaces ("defined" interfaces are those that have been defined, but are currently inactive). Each of these functions takes a connection object as its first argument; the list functions also take an argument pointing to a char* array for the result, and another giving the maximum number of entries to put in that array. All four functions return the number of interfaces found, or -1 if an error is encountered.

Example 7.5. Getting a list of active ("up") interfaces on a host

Note: error handling omitted for clarity

```
int numIfaces, i;
char *ifaceNames;

numIfaces = virConnectNumOfInterfaces(conn);
ifaceNames = malloc(numIfaces * sizeof(char*));
numIfaces = virConnectListInterfaces(conn, names, ct);

printf("Active host interfaces:\n");
for (i = 0; i < numIfaces; i++) {
    printf(" %s\n", ifaceNames[i]);
    free(ifaceNames[i]);
}
free(ifaceNames);
```

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

```
int numIfaces, i;
char *ifaceNames;

numIfaces = virConnectNumOfDefinedInterfaces(conn);
ifaceNames = malloc(numIfaces * sizeof(char*));
numIfaces = virConnectListDefinedInterfaces(conn, names, ct);

printf("Inactive host interfaces:\n");
for (i = 0; i < numIfaces; i++) {
    printf(" %s\n", ifaceNames[i]);
    free(ifaceNames[i]);
}
free(ifaceNames);
```

Alternative method of enumerating interfaces

It is also possible to get a list of interfaces from the virNodeDevice function. Calling virNodeListDevices with the *cap* argument (capabilities) set to net. This will return a list of device names (each starting with "net_"), and those device names can, in turn, be sent through virNodeDeviceLookupByName, then virNodeDeviceGetXMLDesc to get an XML string containing the interfaces' names, MAC addresses, and 802.11 vs. 802.03 status (wired vs wireless). See the section called “Device configuration” for more information and examples of using virNodeDevice functions for this purpose.

Obtaining a virInterfacePtr for an Interface

Many operations require that you have a `virInterfacePtr`, but you may only have the name or MAC address of the interface. You can use `virInterfaceLookupByName` and `virInterfaceLookupByMACString` to get the `virInterfacePtr` in these cases.

Example 7.7. Fetching the virInterfacePtr for a given interface name

```
virInterfacePtr iface;
const char *name = "eth0";

iface = virInterfaceLookupByName(name);

if (iface) {

    /* use the virInterfacePtr ... */

    virInterfaceFree(iface);
} else {
    printf("Interface '%s' not found.\n", name);
}
```

Example 7.8. Fetching the virInterfacePtr for a given interface MAC Address

```
virInterfacePtr iface;
const char *mac = "00:01:02:03:04:05";

iface = virInterfaceLookupByMACString(mac);

if (iface) {

    /* use the virInterfacePtr ... */

    virInterfaceFree(iface);
} else {
    printf("No interface found with MAC address '%s'.\n", mac);
}
```

Note that, as shown in the examples, after you are finished using the `virInterfacePtr`, you must call `virInterfaceFree` to free up its resources, even if you undefined or destroyed the interface in the meantime. Also note that performing a lookup for a MAC address that has multiple matches will result in a NULL return and a VIR_ERR_MULTIPLE_INTERFACES error being raised. This limitation will be addressed in the near future with a new API function.

Retrieving Detailed Interface Information

You may also find yourself with a `virInterfacePtr`, and need the name or MAC address of the interface, or want to examine the full interface configuration. The `virInterfaceGetName`, `virInterfaceGetMACString`, and `virInterfaceGetXMLDesc` functions provide this capability.

Example 7.9. Fetching the name and mac address from an interface object

```
const char *name;
const char *mac;

name = virInterfaceGetName(iface);
mac = virInterfaceGetMACString(iface);

printf("Interface %s has MAC address %s", name, mac);
```

Note that the strings returned by `virInterfaceGetName` and `virInterfaceGetMACString` do not need to be freed by the application; their lifetime will be the same as the interface object.

The string returned by `virInterfaceGetXMLDesc`, on the other hand, is created especially for the caller, and the caller must free it when finished. `virInterfacegetXMLDesc` also has a flags argument, intended for future expansion. For forward compatibility, you should always set it to 0. The returned string is UTF-8 encoded. The same string may later be given to `virInterfaceDefineXML` to recreate the interface configuration.

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

```
const char *xml;

name = virInterfaceGetXMLDesc(iface, 0);
printf("Interface configuration:\n%s\n", xml);
free(xml);
```

Managing interface configuration files

In libvirt, "defining" an interface means creating or changing the configuration, and "undefining" means deleting that configuration from the system. Newcomers may sometimes confuse these two operations with Create/Delete (which actually are used to activate and deactivate an existing interface - see the section called “Interface lifecycle management”).

Defining an interface configuration

The `virInterfaceDefineXML` function 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 `virInterfaceCreate`, described in the section called “Interface lifecycle management”)

If the interface is successfully added/modified in the host's configuration, `virInterfaceDefineXML` returns a `virInterfacePtr`. This can be used as a handle to perform further actions on the new interface, for example making it active with `virInterfaceCreate`.

When you are finished using the returned `virInterfacePtr`, you must free it with `virInterfaceFree`. This does not remove the interface itself, just the internal object used by libvirt.

Example 7.11. Defining a new interface

```
/* xml is a char* containing the description, per section 7.2 */
virInterfacePtr iface;

iface = virInterfaceDefineXML(xml, 0);
if (!iface) {
   fprintf(stderr, "Failed to define interface.\n");
   /* other error handling */
   goto cleanup;
}
if (virInterfaceCreate(iface) != 0) {
   fprintf(stderr, "Failed to create (activate) interface\n");
   /* other error handling */
   goto cleanup;
}
virInterfaceFree(iface);

cleanup:
   /* ... */
```

Undefining an interface configuration

The `virInterfaceUndefine` function 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 call `virInterfacegetXMLDesc` and save the string prior to the undefine.

`virInterfaceUndefine` does not free the `virInterfacePtr` itself, it only removes the configuration from the host. You must still free the `virInterfacePtr` with `virInterfaceFree`.

Example 7.12. Undefining br0 interface after saving its XML data

```
virInterfacePtr iface;
char *xml = NULL;;

iface = virInterfaceLookupByName("br0");
if (!iface) {
    printf ("Interface br0 not found.\n");
} else {
    xml = virInterfaceGetXMLDesc(iface, 0);
    virInterfaceUndefine(iface);
    virInterfaceFree(iface);
}
/* you must also free the buffer at xml when you're finished with it */
--------------
```

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.

Activating an interface

`virInterfaceCreate` 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 function.

Deactivating an interface

`virInterfaceDestroy` makes the given interface inactive ("down"). On success, it returns 0. If there is any problem making the interface active, -1 is returned.

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

```
virInterfacePtr iface;

iface = virInterfaceLookupByName("eth2");
if (!iface) {
    printf("Interface eth2 not found.\n");
} else {
    if (virInterfaceDestroy(iface) != 0) {
        fprintf(stderr, "failed to destroy (deactivate) interface eth2.\n");
    } else
        /* do whatever you wanted to do with interface down */
        if (virInterfaceCreate(iface) != 0) {
        fprintf(stderr, "failed to create (activate) interface eth2.\n");
        }
    }
    free(iface);
}
```

Interface object memory management

Any time an application calls a function that returns a `virInterfacePtr`, it is implied that a reference counter has been incremented for that particular interface object. To decrement the reference counter (eventually resulting in the interface object's resources being freed), call `virInterfaceFree`. This reference counting assures that the interface object will not be freed while an application is still using it.

For cases where an application makes a copy of a `virInterfacePtr` and stores it away somewhere which may require a lifetime longer than that of the original `virInterfacePtr`, `virInterfaceRef` should be called to manually increment the reference count. `virInterfaceFree` should then be called an extra time, when that copy of the `virInterfacePtr` is no longer being used.

Example 7.14. Reference counting an interface object

```
virInterfacePtr iface;

iface = virInterfaceLookupByName("eth0");

mydata.iface = iface;
virInterfaceRef(mydata.iface);
```

```
/* now we're done with iface */
virInterfaceFree(iface);

...

/* now we're done with mydata.iface */
virInterfaceFree(mydata.iface);
```

Chapter 8. Host Devices

Currently lacks docs

Chapter 9. Alternative Language Bindings

TBD

Python

TBD

Perl

TBD

Java

TBD

Appendix A. Revision History

Revision History

Revision 1-4	Mon Aug 23 2010	DavidJorm<djorm@redhat.com>

Updated content for the connections chapter

Revision 1-3	Mon Aug 16 2010	DavidJorm<djorm@redhat.com>

Several new chapters of content, extensive editing

Revision 1-2	Tue Jan 19 2010	DavidJorm<djorm@redhat.com>

First draft published to libvirt.org

Revision 1-1	Tue Nov 17 2009	DanielBerrange<berrange@redhat.com>

Initial draft of document