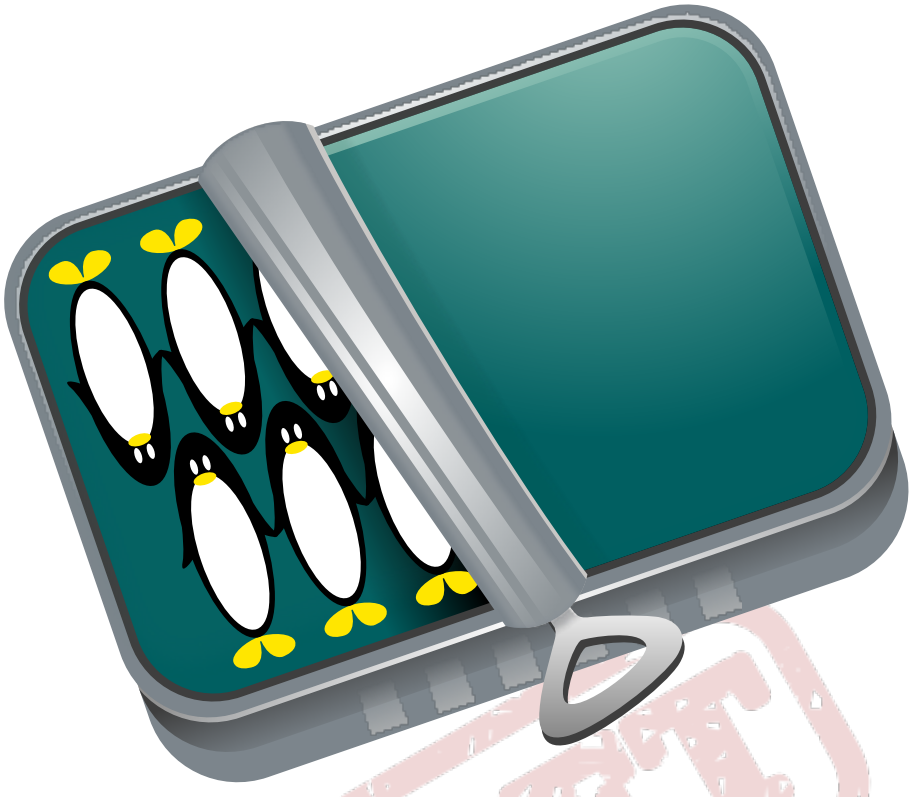
libvirt 0.7.3

Application Development Guide

A guide to application development with libvirt



Dani Coulson

Daniel Berrange

Daniel Veillard

Chris Lalancette

Laine Stump

libvirt 0.7.3 Application Development Guide
A guide to application development with libvirt
Edition 1

Author	Dani Coulson
Author	Daniel Berrange
Author	Daniel Veillard
Author	Chris Lalancette
Author	Laine Stump

This document provides a guide for application developers using libvirt.


DRAFT


DRAFT

Preface

1. Document Conventions

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

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

1.1. Typographic Conventions

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

`Mono-spaced Bold`

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

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

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

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

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

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

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

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

Proportional Bold

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

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

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

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

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

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

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

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

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

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

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

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

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

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

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

import javax.naming.InitialContext;

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

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

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

1.3. Notes and Warnings

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

Note

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

Important

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

Warning

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

2. We Need Feedback!

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


DRAFT

Introduction

Libvirt is a hypervisor-independent virtualization API that is able to interact with the virtualization capabilities of a range of operating systems.

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

1.1. Overview

Libvirt provides a common, generic and stable layer to securely manage domains on a node. As nodes may be remotely located, libvirt provides all 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.

1.2. Glossary of terms

To avoid ambiguity regarding terms and concepts used in this guide, refer to the following descriptions.

Term	Description
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 machine.

Table 1.1. Terminology


DRAFT

Architecture

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

2.1. 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.

2.1.1. 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 *Section 3.2, "URI formats"* for a full description of valid URIs.

An application is permitted to open multiple connections at the same time, even when using more than one type of hypervisor on a single machine. For example, a host may provide both KVM full machine virtualization and LXC container virtualization. A connection object may be used concurrently across multiple threads. Once a connection has been established, it is possible to obtain handles to other managed objects or create new managed objects, as discussed in *Section 2.1.2, "Guest domains"*..

2.1.2. Guest domains

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

2.1.3. Virtual networks

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

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

A virtual network is represented by 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 trace of it will disappear. A persistent virtual network has its configuration maintained in a data store on the host, in an implementation defined format. Thus when a persistent network is brought offline, it is still possible to manage its inactive config. A transient network can be turned into a persistent network on the fly by defining a configuration for it.

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

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

2.1.4. Storage pools

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

2.1.5. Storage volumes

The storage volume object provides management of an allocated block of storage within a pool, be it a disk partition, logical volume, SCSI/iSCSI LUN, or a file within a local/network file system. Once allocated, a volume can be used to provide disks to one (or more) virtual domains. A volume is represented by 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 *Section 5.7, "Volume overview"* for further information about using storage volume objects

2.1.6. Host devices

Host devices provide a view to the hardware devices available on the host machine. This covers both the physical USB or PCI devices and logical devices these provide, such as a NIC, disk, disk controller, sound card, etc. Devices can be arranged to form a tree structure allowing relationships to be identified. A host device is represented by 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.

2.2. Driver model

The libvirt library exposes a guaranteed stable API & ABI which is decoupled from any particular virtualization technology. In addition many of the APIs have associated XML schemata which are considered part of the stable ABI guarantee. Internally, there are multiple of implementations of the public ABI, each targeting a different virtualization technology. Each implementation is referred to as a driver. When obtaining a instance of the **`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`**

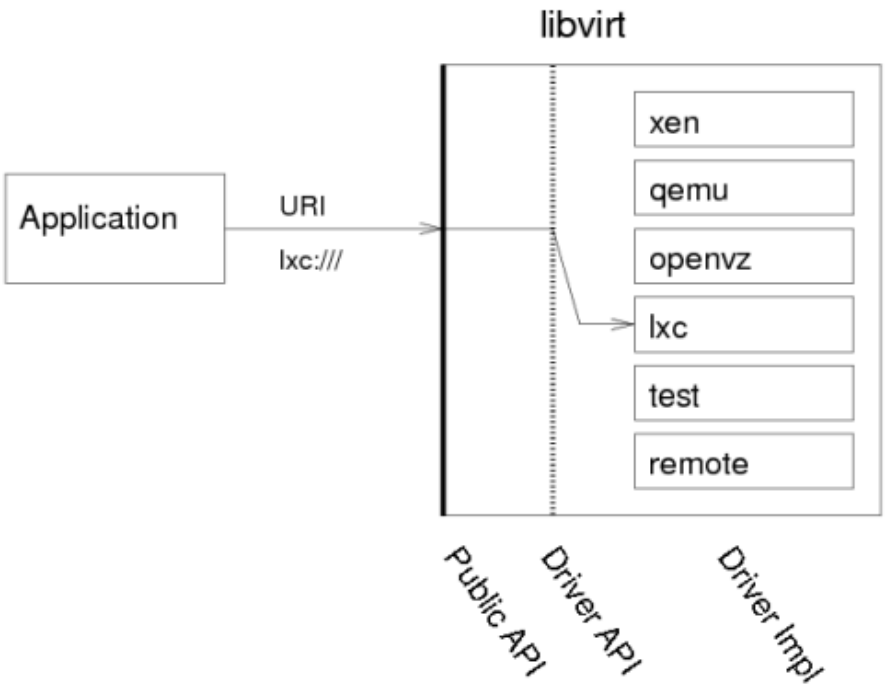


Figure 2.1. libvirt driver architecture

2.3. Remote management

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

In addition to the native RPC service included in libvirt, there are a number of alternatives for remote management that will not be discussed in this document. The **`libvirt-qpid`** project provides an agent for the QPid messaging service, exposing all libvirt managed objects and operations over the

message bus. This keeps a fairly close, near 1-to-1, mapping to the C API in libvirt. The **libvirt-CIM** project provides a CIM agent, that maps the libvirt object model onto the DMTF virtualization schema.

2.3.1. Basic usage

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

As can be seen from earlier example libvirt driver URIs, then hostname field in the URI is always left empty for local libvirt connections. To make use of the libvirt RPC driver, only two changes are required to the local URI. At least a hostname must be specified, at which point libvirt will attempt to use the direct TLS data transport. An alternative data transport can be requested by appending its name to the URI scheme. The URIs formats will be described in detail later in this document *Section 2.3.4, "Remote URIs"*

2.3.2. Data Transports

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

Transport	Description
tls	A TCP socket running the TLS protocol on the wire. This is the default data transport if none is explicitly requested, and uses a TCP connection on port 16514. At minimum it is necessary to configure the server with a x509 certificate authority and issue it a server certificate. The **libvirtd** server can, optionally, be configured to require clients to present x509 certificates as a means of authentication.
tcp	A TCP socket without the TLS protocol on the wire. This data transport should not be used on untrusted networks, unless the SASL authentication service has been enabled and configured with a plug-in that provides encryption. The TCP connection is made on port 16509.
unix	A local only data transport, allowing users to connect to a **libvirtd** daemon running as a different user account. As it is only accessible on the local machine, it is unencrypted. The standard socket names are **/var/run/libvirt/libvirt-sock** for full management capabilities and **/var/run/libvirt/libvirt-sock-ro** for a socket restricted to read only operations.

Transport	Description
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.

Table 2.1. Transports

2.3.3. Authentication schemes

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

Scheme	Description
sasl	SASL is a industry standard for pluggable authentication mechanisms. Each plug-in has a wide variety of capabilities and discussion of their merits is outside the scope of this document. For the **`tls`** data transport there is a wide choice of plug-ins, since TLS is providing data encryption for the network channel. For the **`tcp`** data transport, libvirt will refuse to use any plug-in which does not support data encryption. This effectively limits the choice to GSSAPI/ Kerberos. SASL can optionally be enabled on the UNIX domain socket data transport if strong authentication of local users is required.
polkit	PolicyKit is an authentication scheme suitable for local desktop virtualization deployments, for use only on the UNIX domain socket data transport. It enables the libvirtd daemon to validate that the client application is running within the local X desktop session. It can be configured to allow access to a logged in user automatically, or

Scheme	Description
	prompt them to enter their own password, or the superuser (root) password.
x509	Although not strictly an authentication scheme, the TLS data transport can be configured to mandate the use of client x509 certificates. The server can then whitelist the client distinguished names to control access.

Table 2.2. Schemes

2.3.4. Remote URIs

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

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

A detailed description of each component now follows

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**

Component	Description
	for the TCP data transport and **16514** for the TLS data transport.
path	The path should be the same path used for the hypervisor driver's local URIs. For Xen, this is always just **/**, while for QEMU this would be **/system**.
extraparameters	The URI query parameters provide the mean to fine tune some aspects of the remote connection, and are discussed in depth in the next section.

Table 2.3. URI components

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

Connect to a remote Xen hypervisor on host *node.example.com* using ssh tunneled data transport and ssh username *root*: **xen+ssh://root@node.example.com/**

Connect to a remote QEMU hypervisor on host *node.example.com* using TLS with x509 certificates: **qemu://node.example.com/system**

Connect to a remote Xen hypervisor on host *node.example.com* using TLS, skipping verification of the server's x509 certificate (NB: this is compromising your security): **xen://node.example.com/?no_verify=1**

Connect to the local QEMU instances over a non-standard Unix socket (the full path to the Unix socket is supplied explicitly in this case): **qemu+unix:///system?socket=/opt/libvirt/run/libvirt/libvirt-sock**

Connect to a libvirtd daemon offering unencrypted TCP/IP connections on an alternative TCP port 5000 and use the test driver with default configuration: **test+tcp://node.example.com:5000/default**

For further information on local URIs, refer to *Section 3.2, "URI formats"*

Extra parameters

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

Name	Transports	Description
name	*any transport*	The local hypervisor URI passed to the remote virConnectOpen function. This URI is normally formed by removing transport, hostname, port number, username and extra parameters from the remote URI, but in certain very complex cases it may be necessary to supply the

Name	Transports	Description
		name explicitly. Example: **`name=qemu:///system`**
`command`	ssh, ext	The external command. For ext transport this is required. For ssh the default is ssh. The PATH is searched for the command. Example: **`command=/opt/openssh/bin/ssh`**
socket	unix, ssh	The external command. For ext transport this is required. For ssh the default is **`ssh`**. The PATH is searched for the command. Example: **`socket=/opt/libvirt/run/libvirt/libvirt-sock`**
netcat	ssh	The name of the netcat command on the remote machine. The default is nc. For ssh transport, libvirt constructs an ssh command which looks like: `command -p port [-l username] hostname netcat -U socket` Where port, username, hostname can be specified as part of the remote URI, and command, netcat and socket come 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

Name	Transports	Description
		automatically (For example, when using a ssh-agent). Use this when you don't have access to a terminal - for example in graphical programs which use libvirt. Example: **no_tty=1**

Table 2.4. Extra parameters for remote URIs

2.4. Generating TLS certificates

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

1. The client checks that it is connecting to the correct server by matching the certificate the server sends with the server's hostname. This check can be disabled by adding **?no_verify=1**. Refer to *Table 2.4, "Extra parameters for remote URIs"* for details.
2. The server checks to ensure that only allowed clients are connected. This is performed using either:
   a. The client's IP address; or
   b. The client's IP address and the client's certificate.

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

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

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

2.4.1. Public Key Infrastructure setup

Placeholder

Location	Machine	Description	Required fields
/etc/pki/CA/cacert.pem	Installed on all clients and servers	CA's certificate	n/a
/etc/pki/libvirt/private/serverkey.pem	Installed on the server	Server's private key	n/a
/etc/pki/libvirt/servercert.pem	Installed on the server	Server's certificate signed by the CA	CommonName (CN) must be the hostname of the server as it is seen by clients.

Location	Machine	Description	Required fields
`/etc/pki/libvirt/private/clientkey.pem`	Installed on the client	Client's private key.	n/a
`/etc/pki/CA/cacert.pem`	Installed on the client	Client's certificate signed by the CA	Distinguished Name (DN) can be checked against an access control list (**`tls_allowed_dn_list`**).

Table 2.5. Public Key setup

Connections

In libvirt, a connection is the underpinning of every action and object in the system. Every entity that wants to interact with libvirt, be it virsh, virt-manager, or a program using the libvirt library, needs to first obtain a connection to the libvirt daemon on the host it is interested in interacting with. A connection describes not only the type of virtualization technology that the agent wants to interact with (qemu, xen, uml, etc), but also describes any authentication methods necessary to connect to that resource.

3.1. Overview

The very first thing a libvirt agent must do is call 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 flags)
```

In all three cases there is a **name** parameter which in fact refers to the URI of the hypervisor to connect to. The previous sections *Section 2.2, “Driver model”* and *Section 2.3.4, “Remote URIs”* provide full details on the various URI formats that are acceptable. If the URI is omitted 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 they want 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. The first **virConnectOpen** method 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 POSIX credentials of the application. The second **virConnectOpenReadOnly** 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 before this API has no scope for authentication callbacks, so relies on POSIX credentials. The final **virConnectOpenAuth** method 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 if desired.

A connections must be released by calling **virConnectClose** when no longer required. Connections are a reference counted object, 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 relaese the reference when no longer required. Note also, that every other object associated with a connection (**virDomainPtr**, **virNetworkPtr**, etc) will also hold a reference on the connection. So to avoid leaking a connection object, applications must ensure all associated objects are also freed.

3.2. URI formats

TBD

3.3. Capability information

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

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

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

 <guest>
   <os_type>hvm</os_type>
   <arch name='x86_64'>
     <wordsize>64</wordsize>
     <emulator>/usr/bin/qemu-system-x86_64</emulator>
```

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

</capabilities>
```

Example 3.1. Example QEMU driver capabilities

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

The /host sub-document describes the capabilities of the host.

/host/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/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, only the xen and qemu drivers support migration. The /host/migration_features/live XML node exists if the driver supports live migration; both xen and qemu support 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 (FIXME: where is this?) 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. As of this writing, only the qemu driver supports the security model, and

only the selinux model is implemented. For more information about security, please see the Security section. (FIXME: where is this?)

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.

qemu driver: always "hvm" for a fully-virtualized guest xen driver: either "xen" for a paravirtualized guest or "hvm" for a fully virtualized guest uml driver: always "uml" lxc driver: always "exe" vbox driver: always "hvm" for a fully-virtualized guest openvz driver: always "exe" one driver: always "hvm" esx driver: 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. Note that these default machine types can be overridden by the /guest/arch/domain/machine elements (described below) for guest types 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. FIXME: describe more about this

3.4. Host information

TBD

3.5. Event loop integration

The purpose of the libvirt event loop APIs is so that libvirt can be easily integrated into event driven applications, such as GUIs. These methods provide a way for the libvirt library to notify an application

that there are events that need servicing; the application can then service these events during it's normal event loop.

In order to accomplish this, there are callbacks in both directions, from the application to libvirt, and from libvirt to the application. When initially registering the event callback, the application should provide various callbacks to libvirt; these functions will be called when libvirt adds, deletes, or otherwise modifies a handle. When libvirt calls these functions, they should update internal application state regarding the handles to be monitored.

Eventually a libvirt handle will need servicing. When this happens, the application should call the callback previously provided by libvirt to service the handle in question.

3.5.1. Event Types

The event types define what types of events an event loop should monitor a particular handle for. The event types are specific to libvirt, and must be translated by the application to and from poll() events.

VIR_EVENT_HANDLE_READABLE - the handle has data to read VIR_EVENT_HANDLE_WRITABLE - writing to the handle will not block VIR_EVENT_HANDLE_ERROR - the handle has had some kind of error VIR_EVENT_HANDLE_HANGUP - the handle has hung up. This typically means the handle has been closed.

3.6. Security model

TBD

3.7. Error handling

TBD

3.8. Debugging / logging

TBD


DRAFT

Guest Domains

4.1. Domain overview

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 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 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.

```
int domainID = 6;
virDomainPtr dom;

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

Example 4.1. Fetching a domain object from an ID

```
int domainName = "someguest";
virDomainPtr dom;

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

Example 4.2. Fetching a domain object from an name

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

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

Example 4.3. Fetching a domain object from an UUID

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[]`**

4.2. 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

```
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);
```

Example 4.4. Listing active domains

In addition to the running domains, there may be some persistent inactive domain configurations stored on the host. Since an inactive domain does not have any ID identifier, the listing of inactive domains is exposed as a list of name strings. 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:

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

numDomains = virConnectNumOfDefinedDomains(conn);

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

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

Example 4.5. Listing inactive domains

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:

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

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

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

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

for (i = 0 ; i < numActiveDomains ; i++) {
allDomains[numDomains]
= virDomainLookupByID(activeDomains[i]);
numDomains++
}
```

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

Example 4.6. Fetching all domain objects

4.3. Lifecycle control

TBD

4.3.1. Provisioning

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.

4.3.1.1. 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.

4.3.1.1.1. Booting a transient guest domain

To boot a transient guest domain, simply requires a connection to libvirt and a string containing the XML document describing the required guest configuration. 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.

4.3.1.1.2. Defining and booting a persistent guest domain

Before a persistent domain can be booted, it must have its configuration defined. This again requires a connection to libvirt and a string containing the XML document describing the required guest configuration. The **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);
```

4.3.1.2. 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>
```

TIP: Be careful in the choice of initial memory allocation, since too low a value may cause mysterious crashes and installation failures. Some operating systems need as much as 600 MB of memory for initial installation, though this can often be reduced post-install.

4.3.1.2.1. CDROM/ISO image provisioning

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

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

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

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

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

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

With the configuration determined, it is now possible to provision the guest. This is an easy process, simply requiring a persistent guest to be defined, and then booted.

```
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.

4.3.1.2.2. PXE boot provisioning

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

4.3.1.2.3. Direct kernel boot provisioning

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

The one complication with direct kernel booting is that provisioning becomes a two step process. For the first step, it is necessary to configure the guest XML configuration to point to a kernel/initrd.

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

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");
```

4.3.2. Save / restore

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, &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);
```

4.3.3. Migration

TBD

4.3.4. Autostart

TBD

4.4. Monitoring performance

TBD

4.4.1. Domain performance

TBD

4.4.2. vCPU performance

TBD

4.4.3. I/O statistics

TBD

4.5. Domain configuration

TBD

4.5.1. Boot modes

TBD

4.5.2. Memory / CPU resources

TBD

4.5.3. Lifecycle controls

TBD

4.5.4. Clock sync

TBD

4.5.5. Features

TBD

4.6. Device configuration

TBD

4.6.1. Emulator

TBD

4.6.2. Disks

TBD

4.6.3. Networking

TBD

4.6.4. Filesystems

TBD

4.6.5. Mice & tablets

TBD

4.6.6. USB device passthrough

TBD

4.6.7. 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\n");
    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>
```

4.7. Live configuration change

TBD

4.7.1. Memory ballooning

TBD

4.7.2. CPU hotplug

TBD

4.7.3. Device hotplug / unplug

TBD

4.7.4. Device media change

TBD

4.8. Security model

TBD

4.9. Event notifications

TBD

4.10. Tuning

TBD

4.10.1. Scheduler parameters

TBD

4.10.2. NUMA placement

TBD

Storage Pools

This is a test paragraph

5.1. Overview

TBD

5.2. Listing pools

TBD

5.3. Pool usage

TBD

5.4. Lifecycle control

TBD

5.5. Discovering pool sources

TBD

5.6. Pool configuration

TBD

5.7. Volume overview

TBD

5.8. Listing volumes

TBD

5.9. Volume information

TBD

5.10. Creating and deleting volumes

TBD

5.11. Cloning volumes

TBD

5.12. Configuring volumes

TBD

Virtual Networks

TBD

6.1. Overview

TBD

6.2. Listing networks

TBD

6.3. Lifecycle control

TBD

6.4. Network configuration

TBD


DRAFT

Network Interfaces

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

7.1. 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.

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, but the live configuration of the interface itself will not be changed until either the interface is started (with virInterfaceCreate), or the host is rebooted.

7.2. XML Interface Description Format

The current Relax NG definition of the XML that is produced/accepted by virInterfaceDefineXML/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.

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

```
  </protocol>
</interface>
```

Example 7.1. XML definition of an ethernet interface using DHCP

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

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

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

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

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

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

7.3. Retrieving Information About Interfaces

7.3.1. Enumerating Interfaces

Once you have a connection to a host (a virConnectPtr) you can learn the number of interfaces on the host with virConnectNumOfInterfaces and virConnectNumOfDefinedInterfaces, and a list of those interfaces' names 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 4 functions return the number of interfaces found, or -1 if an error is encountered.

NB, 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 7.5. Getting a list of active ("up") 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);
```

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

7.3.2. Alternative method of enumerating interfaces

It is also possible to get a list of interfaces from the virNodeDevice API. 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 section 4.6 for more information and examples of using virNodeDevice functions for this purpose.

7.3.3. 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.

```
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 7.7. Fetching the virInterfacePtr for a given interface name

```
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);
}
```

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

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). Another important detail: doing 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.

7.3.4. Retrieving Detailed Interface Info

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. virInterfaceGetName, virInterfaceGetMACString, and virInterfaceGetXMLDesc will take care of those needs.

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

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

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

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

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.

```
const char *xml;

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

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

7.4. Managing interface configuration files

In libvirt, "defining" an interface means creating or changing the configuration, and "undefining" means deleting that configuration from the system. Newcomers may sometimes confuse these two operations with Create/Delete (which actually are used to activate and deactivate an existing interface - see section 7.5).

7.4.1. Defining an inteface configuration

The virInterfaceDefineXML function is used both for 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 section 7.5)

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

When you are finished using the returned virInterfacePtr, you must free it with virInterfaceFree (this doesn't remove the interface itself, just the internal object used by libvirt).

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

Example 7.11. Defining a new interface

7.4.2. Undefining an inteface configuration

virInterfaceUndefine completely and permanently removes the configuration for the given interface from tho host's configuration files. If you might 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.

```
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 */
--------------
```

Example 7.12. Undefining br0 interface after saving its XML data

7.5. Interface lifecycle management

In libvirt parlance, "creating" an interface means making it active, or "bringing it up", and "deleting" an interface means making it inactive, or "bringing it down". On hosts using the netcf backend for interface configuration (eg Fedora, RHEL), this is the same as calling the system shell scripts "ifup" and "ifdown" for the interface.

7.5.1. 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.

7.5.2. Activating an interface

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

```
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);
}
```

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

7.6. 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 (and virinterfaceFree should be called an extra time, when that copy of the virInterfacePtr is no longer being used).

```
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);
```

Example 7.14. Reference counting an interface object


DRAFT

Host Devices

Currently lacks docs


DRAFT

Alternative Language Bindings

TBD

9.1. Python

TBD

9.2. Perl

TBD

9.3. Java

TBD


DRAFT

Appendix A. Revision History

Revision 1 Tue Nov 17 2009 Daniel Berrange *berrange@redhat.com*

Initial draft of document


DRAFT