

XenServer Software Development Kit Guide

4.1.0

Published March 2008

1.0 Edition

CITRIX®

XenServer Software Development Kit Guide: Release 4.1.0

Published March 2008
Copyright © 2008 Citrix Systems, Inc.

Xen®, Citrix™, Enterprise Edition™, XenServer™, Express Edition™, XenCenter™ and logos are either registered trademarks or trademarks of Citrix Systems, Inc. in the United States and/or other countries. Other company or product names are for informational purposes only and may be trademarks of their respective owners.

This product contains an embodiment of the following patent pending intellectual property of Citrix Systems, Inc.:

1. United States Non-Provisional Utility Patent Application Serial Number 11/487,945, filed on July 17, 2006, and entitled “Using Writeable Page Tables for Memory Address Translation in a Hypervisor Environment”.

2. United States Non-Provisional Utility Patent Application Serial Number 11/879,338, filed on July 17, 2007, and entitled “Tracking Current Time on Multiprocessor Hosts and Virtual Machines”.

Table of Contents

1. Introduction 1
2. Getting Started 2
    2.1. System Requirements and Preparation 2
    2.2. Downloading 2
    2.3. Installation 2
    2.4. What's new 2
    2.5. Content Map 2
    2.6. Building Samples for the Linux Platform 3
    2.7. Building Samples for the Windows Platform 3
    2.8. Running the CLI 3
        2.8.1. Tab Completion 3
    2.9. Accessing SDK reference 3
3. Overview of the XenServer API 5
    3.1. Getting Started with the API 5
        3.1.1. Authentication: acquiring a session reference 5
        3.1.2. Acquiring a list of templates to base a new VM installation on 5
        3.1.3. Installing the VM based on a template 6
        3.1.4. Taking the VM through a start/suspend/resume/stop cycle 6
        3.1.5. Logging out 6
        3.1.6. "Install and start example": summary 7
    3.2. Object Model Overview 7
    3.3. Working with VIFs and VBDs 9
        3.3.1. Creating disks and attaching them to VMs 9
        3.3.2. Creating and attaching Network Devices to VMs 11
        3.3.3. Host configuration for networking and storage 11
    3.4. Exporting and Importing VMs 12
    3.5. Where to look next 13
4. Using the API 14
    4.1. Anatomy of a typical application 14
        4.1.1. Choosing a low-level transport 14
        4.1.2. Authentication and session handling 14
        4.1.3. Finding references to useful objects 15
        4.1.4. Invoking synchronous operations on objects 15
        4.1.5. Using Tasks to manage asynchronous operations 16
        4.1.6. Subscribing to and listening for events 16
    4.2. Language bindings 17
        4.2.1. C 17
        4.2.2. C# 18
        4.2.3. Python 18
        4.2.4. Command Line Interface (CLI) 19
    4.3. Complete application examples 19
        4.3.1. Simultaneously migrating VMs using XenMotion 19
        4.3.2. Cloning a VM via the XE CLI 22
5. XenServer API extensions 24
    5.1. VM console forwarding 24
        5.1.1. Retrieving VNC consoles via the API 24
        5.1.2. Disabling VNC forwarding for Linux VM 25
    5.2. Paravirtual Linux installation 26
        5.2.1. Red Hat Enterprise Linux 4.1/4.4 26
        5.2.2. Red Hat Enterprise Linux 4.5/5.0 26
        5.2.3. SUSE Enterprise Linux 10 SP1 27

5.2.4. CentOS 4.5/5.0 27
5.3. Adding Xenstore entries to VMs 27
5.4. Security enhancements 27
5.5. Advanced settings for network interfaces 28
5.5.1. ethtool settings 28
5.5.2. Miscellaneous settings 29
5.6. Internationalization for SR names 29
5.7. Hiding objects from XenCenter 30
Index 31

List of Figures

3.1. Graphical overview of API classes for managing VMs, Hosts, Storage and Networking 9
3.2. A VM object with 2 associated VDIs 11

Chapter 1. Introduction

Welcome to the developer's guide for XenServer. Here you will find the information you need in order to understand and use the Software Development Kit (SDK) that XenServer provides. This information will provide you with some of the architectural background and thinking that underpins the APIs, the tools that have been provided, and how to quickly get off the ground.

Chapter 2. Getting Started

2.1. System Requirements and Preparation

The XenServer SDK is packaged as a Linux VM that must be imported into a XenServer Host. This document refers to the SDK virtual machine interchangeably as an SDK and an SDK VM. The first step towards working with the SDK is to install XenServer. A free version, Express Edition, is available to download at http://www.xensource.com/. Please refer to the *XenServer Installation Guide* for detailed instructions on how to set up your development host. When the installation is complete, please note the *host IP address* and the *host password.*.

Once you have installed your XenServer Host, install XenCenter on a Windows PC. Launch the application and connect to your new XenServer Host using its IP address and the password.

2.2. Downloading

The SDK is available for download as a ZIP file at http://www.xensource.com/ of around 250MB.

2.3. Installation

Ensure that your XenServer Host is up and running. In XenCenter, right-click on the XenServer Host and select Import VM from the context menu, or select Import... from the VM menu, then double-click on the `sdk` directory and click the Import button. When the import has completed, the SDK VM is installed and ready to go. Click on the Overview tab of the SDK VM and inspect the networking configuration. If the SDK is to be accessed remotely, then ensure that it has an interface connected to the correct network. Note that the SDK VM will attempt to acquire an IP address via DHCP each time it boots.

2.4. What's new

Starting with version 4.0, we now provide a rich management infrastructure consisting of a comprehensive object model together with an application program interface (API) to install, monitor and manage various aspects of virtual machine infrastructure.

This is the first version of a supported Software Development Kit that Citrix has published. Prior to this version, the primary means of integrating Citrix products into the existing IT infrastructure was through command line interface tools.

The XenServer 4.1.0 SDK provides the API with C and Python language bindings, and C# language binding compatible with .NET 2.0. The SDK also provides a new and improved CLI that provides a comprehensive set of commands to manage your XenServer Hosts. The CLI is available for both Linux and Windows platforms.

2.5. Content Map

The following is an overview of the contents of the /SDK directory. Where necessary, subdirectories have their own individual README files.

Directory	Description
/SDK/	Contains README.txt, a brief text overview
/SDK/docs/pdf/	Contains api.pdf , the PDF version of the reference for the API

Directory	Description
/SDK/docs/html/	Contains index.html, the HTML version of the reference for the API
/SDK/windows-cli	a Windows version of the CLI xe.exe
/SDK/client-examples/c	C examples and a Makefile to build them
/SDK/client-examples/c/src	C source for the language bindings
/SDK/client-examples/csharp/XenSdk.net	Microsoft Visual Studio 2005 solution which includes the C# language bindings (which compile to a .dll) and several example projects
/SDK/client-examples/bash-cli	Simple bash scripts which use the xe CLI
/SDK/client-examples/python	Several example python programs

2.6. Building Samples for the Linux Platform

The SDK VM comes complete with the tools necessary to build the C samples. Looking at the content map, the directory `/SDK/client-examples/c` has a Makefile at the top level which builds the language bindings, as well as the C language samples that are included as part of the SDK VM.

2.7. Building Samples for the Windows Platform

The C# examples in `/SDK/client-examples/csharp/XenSdk.net` must first be copied to a Windows machine with Visual Studio and .NET 2.0 installed. The C# bindings and the samples directories each have the appropriate solution (`.sln`) files generated by Microsoft Visual Studio 2005. Launching them via the Windows Explorer and rebuilding at the top level will build the language bindings as well as the applications.

The IP address of the host is passed in as a parameter to each of the sample applications. This needs to be set/changed to ensure that the applications work against the right XenServer Host.

2.8. Running the CLI

The CLI for Linux is called **xe**, and for Windows is called **xe.exe**. The Windows version is under `/SDK/windows-cli`. This needs to be copied onto a PC running Windows XP or higher, and must have .Net 2.0 installed. When running in the SDK VM, the Linux CLI is already installed and in the default path. Typing **xe** in the SDK VM console launches the CLI.

2.8.1. Tab Completion

The CLI has comprehensive tab completion that allows discovery of the commands and parameters. On launching xe, hitting tab twice shows all the commands that the CLI has.

2.9. Accessing SDK reference

The SDK VM has a built-in web server that allows access to the samples and the complete reference documentation.

Procedure 2.1. To access this information

1. From the command prompt type **ifconfig** and hit ENTER.
2. Note down the IP address for the *`eth0`* interface for this VM. If there is no eth0 interface, please add a virtual NIC.

3. From any other machine, fire up a web browser and type http://<SDK IP address>/

The full URL you need is also displayed in the “Message of the Day” in the SDK VM console after it has completed booting.

Chapter 3. Overview of the XenServer API

In this chapter we introduce the XenServer API (hereafter referred to as just "the API") and its associated object model. The API has the following key features:

- **Management of all aspects of XenServer Host:** Through the API one can manage VMs, storage, networking, host configuration and pools. Performance and status metrics can also be queried via the API.
- **Persistent Object Model:** The results of all side-effecting operations (e.g. object creation, deletion and parameter modifications) are persisted in a server-side database that is managed by the XenServer installation.
- **An event mechanism:** Through the API, clients can register to be notified when persistent (server-side) objects are modified. This enables applications to keep track of datamodel modifications performed by concurrently executing clients.
- **Synchronous and asynchronous invocation:** All API calls can be invoked synchronously (i.e. block until completion); any API call that may be long-running can also be invoked *asynchronously*. Asynchronous calls return immediately with a reference to a *task* object. This task object can be queried (through the API) for progress and status information. When an asynchronously invoked operation completes, the result (or error code) is available via the task object.
- **Remotable and Cross-Platform:** The client issuing the API calls does not have to be resident on the host being managed; nor does it have to be connected to the host via ssh in order to execute the API. API calls make use of the XML-RPC protocol to transmit requests and responses over the network.
- **Secure and Authenticated Access:** The XML-RPC API server executing on the host accepts secure socket connections. This allows a client to execute the APIs over the https protocol. Further, all the API calls execute in the context of a login session generated through username and password validation at the server. This provides secure and authenticated access to the XenServer installation.

3.1. Getting Started with the API

We will start our tour of the API by describing the calls required to create a new VM on a XenServer installation, and take it through a start/suspend/resume/stop cycle. This is done without reference to code in any specific language; at this stage we just describe the informal sequence of RPC invocations that accomplish our "install and start" task.

3.1.1. Authentication: acquiring a session reference

The first step is to call **Session.login(username, password)**. The API is session based, so before you can make other calls you need to authenticate with the server. Assuming the username and password are authenticated correctly, the result of this call is a *session reference*. Subsequent API calls take the session reference as a parameter. In this way we ensure that only API users who are suitably authorized can perform operations on a XenServer installation.

3.1.2. Acquiring a list of templates to base a new VM installation on

The next step is to query the list of "templates" on the host. Templates are specially-marked VM objects that specify suitable default parameters for a variety of supported guest types. (If you want to see a quick

enumeration of the templates on a XenServer installation for yourself then you can execute the "**xe template-list**" CLI command.) To get a list of templates via the API, we need to find the VM objects on the server that have their "**is_a_template**" field set to true. One way to do this by calling **VM.get_all_records(session)** where the session parameter is the reference we acquired from our **Session.login_with_password** call earlier. This call queries the server, returning a snapshot (taken at the time of the call) containing all the VM object references and their field values.

(Remember that at this stage we are not concerned about the particular mechanisms by which the returned object references and field values can be manipulated in any particular client language: that detail is dealt with by our language-specific API bindings and described concretely in the following chapter. For now it suffices just to assume the existence of an abstract mechanism for reading and manipulating objects and field values returned by API calls.)

Now that we have a snapshot of all the VM objects' field values in the memory of our client application we can simply iterate through them and find the ones that have their "**is_a_template**" set to true. At this stage let's assume that our example application further iterates through the template objects and remembers the reference corresponding to the one that has its "**name_label**" set to "Debian Etch 4.0" (one of the default Linux templates supplied with XenServer).

3.1.3. Installing the VM based on a template

Continuing through our example, we must now install a new VM based on the template we selected. The installation process requires 2 API calls:

- First we must now invoke the API call **VM.clone(session, t_ref, "my first VM")**. This tells the server to clone the VM object referenced by **t_ref** in order to make a new VM object. The return value of this call is the VM reference corresponding to the newly-created VM. Let's call this **new_vm_ref**.
- At this stage the object referred to by **new_vm_ref** is still a template (just like the VM object referred to by **t_ref**, from which it was cloned). To make **new_vm_ref** into a VM object we need to call **VM.provision(session, new_vm_ref)**. When this call returns the **new_vm_ref** object will have had its **is_a_template** field set to false, indicating that **new_vm_ref** now refers to a regular VM ready for starting.

Note that the provision operation may take a few minutes, as it is as during this call that the template's disk images are created. In the case of the Debian template, the newly created disks are actually populated with a Debian root filesystem at this stage too.

3.1.4. Taking the VM through a start/suspend/resume/stop cycle

Now we have an object reference representing our newly-installed VM, it is trivial to take it through a few lifecycle operations:

- To start our VM we can just call **VM.start(session, new_vm_ref)**
- After it's running, we can suspend it by calling **VM.suspend(session, new_vm_ref)**;
- and then resume it by calling **VM.resume(session, new_vm_ref)**.
- We can call **VM.shutdown(session, new_vm_ref)** to shutdown the VM cleanly.

3.1.5. Logging out

Once an application is finished interacting with a XenServer Host it is good practice to call **Session.logout(session)**. This invalidates the session reference (so it can not be used in subsequent API calls) and simultaneously deallocates server-side memory used to store the session object.

Although inactive sessions will timeout eventually, the server has a hardcoded limit of 200 concurrent sessions. Once this limit has been reached fresh logins will evict the oldest session objects, causing their associated session references to become invalid. So if you want your applications to play nice with others accessing the server concurrently, then the best policy is to create a single session at start-of-day, use this throughout the applications (note that sessions can be used across multiple separate client-server *network connections*) and then explicitly logout when possible.

3.1.6. "Install and start example": summary

We have seen how the API can be used to install a VM from a XenServer template and perform a number of lifecycle operations on it. You will note that the number of calls we had to make in order to affect these operations was small:

- One call to acquire a session: **Session.login_with_password(...)**
- One call to query the VM (and template) objects present on the XenServer installation: **VM.get_all_records(...)**. Recall that we used the information returned from this call to select a suitable template to install from.
- Two calls to install a VM from our chosen template: **VM.clone(...)**, followed by **VM.provision(...)**.
- One call to start the resultant VM: **VM.start(...)** (and similarly other single calls to suspend, resume and shutdown accordingly)
- And then one call to logout **Session.logout(...)**

The take-home message here is that, although the API as a whole is complex and fully featured, common tasks (such as creating and performing lifecycle operations on VMs) are very straightforward to perform, requiring only a small number of simple API calls. Keep this in mind while you study the next section which may, on first reading, appear a little daunting!

3.2. Object Model Overview

This section gives a high-level overview of the object model of the API. A more detailed description of the parameters and methods of each class outlined here can be found in the *XenEnterprise Management API* document. Python, C and C# sample programs that demonstrate how the API can be used practice to accomplish a variety of tasks are available in the SDK VM and described in the following Chapter.

We start by giving a brief outline of some of the core classes that make up the API. (Don't worry if these definitions seem somewhat abstract in their initial presentation; the textual description in subsequent sections, and the code-sample walk through in the next Chapter will help make these concepts concrete.)

VM	A VM object represents a particular virtual machine instance on a XenServer Host or Resource Pool. Example methods include "start", "suspend", "pool_migrate"; example fields include "power_state", "memory_static_max", "name_label". (In the previous section we saw how the VM class is used to represent both templates and regular VMs)
Host	A host object represents a physical host in a XenServer pool. Example methods include "reboot" and "shutdown". Example fields include "software_version", "hostname" and [IP] "address".
VDI	A VDI object represents a *Virtual Disk Image*. Virtual Disk Images can be attached to VMs, in which case a block device appears inside the VM through which the bits encapsulated by the Virtual Disk Image can be read and written. Example methods of the VDI class include "resize" and "clone". Example

	fields include "virtual_size" and "sharable". (When we called **VM.provision** on the VM template in our previous example, some VDI objects were automatically created to represent the newly created disks, and attached to the VM object.)
SR	An SR (*Storage Repository*) aggregates a collection of VDIs and encapsulates the properties of physical storage on which the VDIs' bits reside. Example fields include "type" (which determines the storage-specific driver a XenServer installation uses to read/write the SR's VDIs) and "physical_utilisation"; example methods include "scan" (which invokes the storage-specific driver to acquire a list of the VDIs contained with the SR and the properties of these VDIs) and "create" (which initializes a block of physical storage so it is ready to store VDIs).
Network	A network object represents a layer-2 network that exists in the environment in which the XenServer Host instance lives. Since XenServer does not manage networks directly this is a lightweight class that serves merely to model physical and virtual network topology. VM and Host objects that are *attached* to a particular Network object (by virtue of VIF and PIF instances -- see below) can send network packets to each other.

At this point, readers who are finding this enumeration of classes rather terse may wish to skip to the code walk-throughs of the next chapter: there are plenty of useful applications that can be written using only a subset of the classes already described! For those who wish to continue this description of classes in the abstract, read on.

On top of the classes listed above, there are 4 more that act as *connectors*, specifying relationships between VMs and Hosts, and Storage and Networks. The first 2 of these classes that we will consider, *VBD* and *VIF*, determine how VMs are attached to virtual disks and network objects respectively:

VBD	A VBD (*Virtual Block Device*) object represents an attachment between a VM and a VDI. When a VM is booted its VBD objects are queried to determine which disk images (i.e. VDIs) should be attached. Example methods of the VBD class include "plug" (which *hot plugs* a disk device into a running VM, making the specified VDI accessible therein) and "unplug" (which *hot unplugs* a disk device from a running guest); example fields include "device" (which determines the device name inside the guest under which the specified VDI will be made accessible).
VIF	A VIF (*Virtual network InterFace*) object represents an attachment between a VM and a Network object. When a VM is booted its VIF objects are queried to determine which network devices should be create. Example methods of the VIF class include "plug" (which *hot plugs* a network device into a running VM) and "unplug" (which *hot unplugs* a network device from a running guest).

The second set of "connector classes" that we will consider determine how Hosts are attached to Networks and Storage.

PIF	A PIF (*Physical InterFace*) object represents an attachment between a Host and a Network object. If a host is connected to a Network (via a PIF) then packets from the specified host can be transmitted/received by the corresponding host. Example fields of the PIF class include "device" (which specifies the device name to which the PIF corresponds -- e.g. *eth0*) and "MAC" (which specifies the MAC address of the underlying NIC that a PIF represents). Note that PIFs abstract both physical interfaces and VLANs (the latter distinguished by the existence of a positive integer in the "VLAN" field).

PBD	A PBD (*Physical Block Device*) object represents an attachment between a Host and a SR (Storage Repository) object. Fields include "currently-attached" (which specifies whether the chunk of storage represented by the specified SR object) is currently available to the host; and "device_config" (which specifies storage-driver specific parameters that determines how the low-level storage devices are configured on the specified host -- e.g. in the case of an SR rendered on an NFS filer, device_config may specify the host-name of the filer and the path on the filer in which the SR files live.)

Figure 3.1. Graphical overview of API classes for managing VMs, Hosts, Storage and Networking

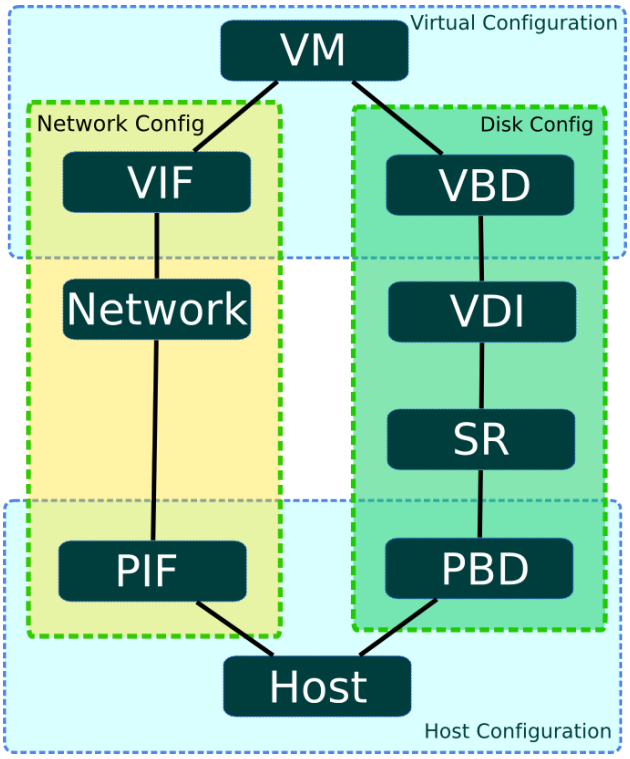


Figure 3.1, “Graphical overview of API classes for managing VMs, Hosts, Storage and Networking” presents a graphical overview of the API classes involved in managing VMs, Hosts, Storage and Networking. From this diagram, the symmetry between storage and network configuration, and also the symmetry between virtual machine and host configuration is plain to see.

3.3. Working with VIFs and VBDs

In this section we walk through a few more complex scenarios, describing informally how various tasks involving virtual storage and network devices can be accomplished using the API.

3.3.1. Creating disks and attaching them to VMs

Let's start by considering how to make a new blank disk image and attach it to a running VM. We will assume that we already have ourselves a running VM, and we know its corresponding API object reference (e.g. we may have created this VM using the procedure described in the previous section, and had the server return its reference to us.) We will also assume that we have authenticated with the XenServer installation

and have a corresponding **session reference**. Indeed in the rest of this chapter, for the sake of brevity, we will stop mentioning sessions altogether.

3.3.1.1. Creating a new blank disk image

The first step is to instantiate the disk image on physical storage. We do this via a call to **VDI.create(...)**. The **VDI.create** call takes a number of parameters, including:

- **name_label** and **name_description**: a human-readable name/description for the disk (e.g. for convenient display in the UI etc.). These fields can be left blank if desired.
- **SR**: the object reference of the Storage Repository representing the physical storage in which the VDI's bits will be placed.
- **read_only**: setting this field to true indicates that the VDI can *only* be attached to VMs in a read-only fashion. (Attempting to attach a VDI with its **read_only** field set to true in a read/write fashion results in error.)

Invoking the **VDI.create** call causes the XenServer installation to create a blank disk image on physical storage, create an associated VDI object (the datamodel instance that refers to the disk image on physical storage) and return a reference to this newly created VDI object.

The way in which the disk image is represented on physical storage depends on the type of the SR in which the created VDI resides. For example, if the SR is of type "lvm" then the new disk image will be rendered as an LVM volume; if the SR is of type "nfs" then the new disk image will be a sparse VHD file created on an NFS filer. (You can query the SR type through the API using the **SR.get_type(..)** call.)

3.3.1.2. Attaching the disk image to a VM

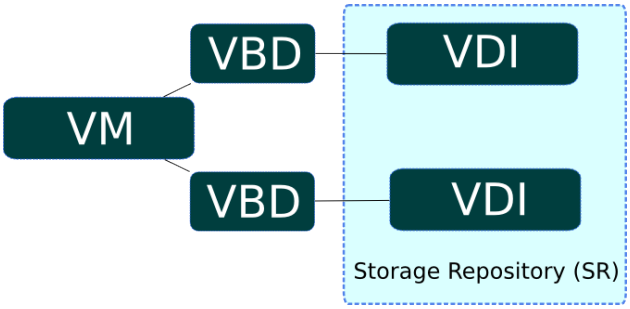
So far we have a running VM (that we assumed the existence of at the start of this example) and a fresh VDI that we just created. Right now, these are both independent objects that exist on the XenServer Host, but there is nothing linking them together. So our next step is to create such a link, associating the VDI with our VM.

The attachment is formed by creating a new "connector" object called a VBD (*Virtual Block Device*). To create our VBD we invoke the **VBD.create(...)** call. The **VBD.create(..)** call takes a number of parameters including:

- **VM**: the object reference of the VM to which the VDI is to be attached
- **VDI**: the object reference of the VDI that is to be attached
- **mode**: specifies whether the VDI is to be attached in a read-only or a read-write fashion
- **userdevice**: specifies the block device inside the guest through which applications running inside the VM will be able to read/write the VDI's bits.
- **type**: specifies whether the VDI should be presented inside the VM as a regular disk or as a CD. (Note that this particular field has more meaning for Windows VMs than it does for Linux VMs, but will not explore this level of detail in this chapter.)

Invoking **VBD.create** makes a VBD object on the XenServer installation and returns its object reference. However, this call in itself does not have any side-effects on the running VM (i.e. if you go and look inside the running VM you will see that the block device has not been created). The fact that the VBD object exists but that the block device in the guest is not active, is reflected by the fact that the VBD object's **currently_attached** field is set to false.

Figure 3.2. A VM object with 2 associated VDIs



For expository purposes, Figure 3.2, “A VM object with 2 associated VDIs” presents a graphical example that shows the relationship between VMs, VBDs, VDIs and SRs. In this instance a VM object has 2 attached VDIs: there are 2 VBD objects that form the connections between the VM object and its VDIs; and the VDIs reside within the same SR.

3.3.1.3. Hotplugging the VBD

If we rebooted the VM at this stage then, after rebooting, the block device corresponding to the VBD would appear: on boot, XenServer queries all VBDs of a VM and actively attaches each of the corresponding VDIs.

Rebooting the VM is all very well, but recall that we wanted to attach a newly created blank disk to a *running* VM. This can be achieved by invoking the **plug** method on the newly created VBD object. When the **plug** call returns successfully, the block device to which the VBD relates will have appeared inside the running VM -- i.e. from the perspective of the running VM, the guest operating system is lead to believe that a new disk device has just been *hot plugged*. Mirroring this fact in the managed world of the API, the **currently_attached** field of the VBD is set to true.

Unsurprisingly, the VBD **plug** method has a dual called "**unplug**". Invoking the **unplug** method on a VBD object causes the associated block device to be *hot unplugged* from a running VM, setting the **currently_attached** field of the VBD object to false accordingly.

3.3.2. Creating and attaching Network Devices to VMs

The API calls involved in configuring virtual network interfaces in VMs are similar in many respects to the calls involved in configuring virtual disk devices. For this reason we will not run through a full example of how one can create network interfaces using the API object-model; instead we will use this section just to outline briefly the symmetry between virtual *networking device* and virtual *storage device* configuration.

The networking analogue of the VBD class is the VIF class. Just as a VBD is the API representation of a block device inside a VM, a VIF (*Virtual Network Device*) is the API representation of a network device inside a VM. Whereas VBDs associate VM objects with VDI objects, VIFs associate VM objects with Network objects. Just like VBDs, VIFs have a **currently_attached** field that determines whether or not the network device (inside the guest) associated with the VIF is currently active or not. And as we saw with VBDs, at VM boot-time the VIFs of the VM are queried and a corresponding network device for each created inside the booting VM. Similarly, VIFs also have **plug** and **unplug** methods for hot plugging/unplugging network devices in/out of running VMs.

3.3.3. Host configuration for networking and storage

We have seen that the VBD and VIF classes are used to manage configuration of block devices and network devices (respectively) inside VMs. To manage host configuration of storage and networking there are two analogous classes: PBD (*Physical Block Device*) and PIF (*Physical [network] InterFace*).

3.3.3.1. Host storage configuration: PBDs

Let us start by considering the PBD class. A **PBD_create(...)** call takes a number of parameters including:

Parameter	Description
host	physical machine on which the PBD is available
SR	the Storage Repository that the PBD connects to
device_config	a string-to-string map that is provided to the host's SR-backend-driver, containing the low-level parameters required to configure the physical storage device(s) on which the SR is to be realized. The specific contents of the `device_config` field depend on the type of the SR to which the PBD is connected. (Executing **xe sm-list** will show a list of possible SR types; the *configuration* field in this enumeration specifies the `device_config` parameters that each SR type expects.)

For example, imagine we have an SR object *s* of type "nfs" (representing a directory on an NFS filer within which VDIs are stored as VHD files); and let's say that we want a host, *h*, to be able to access *s*. In this case we invoke **PBD.create(...)** specifying host *h*, SR *s*, and a value for the *device_config* parameter that is the following map:

("server", "my_nfs_server.example.com"), ("serverpath", "/scratch/mysrs/sr1")

This tells the XenServer Host that SR *s* is accessible on host *h*, and further that to access SR *s*, the host needs to mount the directory "/scratch/mysrs/sr1" on the NFS server named "my_nfs_server.example.com."

Like VBD objects, PBD objects also have a field called **currently_attached**. Storage repositories can be attached and detached from a given host by invoking **PBD.plug** and **PBD.unplug** methods respectively.

3.3.3.2. Host networking configuration: PIFs

Host network configuration is specified by virtue of PIF objects. If a PIF object connects a network object, *n*, to a host object *h*, then the network corresponding to *n* is bridged onto a physical interface (or a physical interface plus a VLAN tag) specified by the fields of the PIF object.

For example, imagine a PIF object exists connecting host *h* to a network *n*, and that **device** field of the PIF object is set to "eth0." This means that all packets on network *n* are bridged to the NIC in the host corresponding to host network device "eth0."

3.4. Exporting and Importing VMs

VMs can be exported to a file and later imported to any XenServer Host. The export protocol is a simple HTTP(S) GET, which should be performed on the master. Authorization is either standard HTTP basic authentication, or if a session has already been obtained, this can be used. The VM to export is specified either by UUID or by reference. To keep track of the export, a task can be created and passed in via its reference. The request might result in a redirect if the VM's disks are only accessible on a slave.

The followwing arguments are passed on the command line:

Argument	Description
session_id	the reference of the session being used to authenticate; required only when not using HTTP basic authentication
task_id	the reference of the task object with which to keep track of the operation
ref	the reference of the VM; required only if not using the UUID

Argument	Description
uuid	the uuid of the VM; required only if not using the reference

For example:

curl http://root:foo@xenserver/export&uuid=[VM UUID]&task_id= [task ID] -o export

To export just the metadata, use the URI http://server/export_metadata

The import protocol is similar, using HTTP(S) PUT. The [session_id] and [task_id] arguments are as for the export. The [ref] and [uuid] are not used; a new reference and uuid will be generated for the VM. There are some additional parameters:

Argument	Description
restore	if this parameter is *true*, the import is treated as replacing the original VM - the implication of this currently is that the MAC addresses on the VIFs are exactly as the export was, which will lead to conflicts if the original VM is still being run.
force	if this parameter is *true*, any checksum failures will be ignored (the default is to destroy the VM if a checksum error is detected)
sr_id	The reference of an SR into which the VM should be imported. The default behaviour is to import into the *Pool.default_SR*.

To import just the metadata, use the URI http://server/import_metadata

3.5. Where to look next

In this chapter we have presented a brief high-level overview of the API and its object-model. The aim here is not to present the detailed semantics of the API, but just to provide enough background for you to start reading the code samples of the next chapter and to find your way around the more detailed *XenEnterprise Management API* reference document.

There are a number of places you can find more information:

- The *Administrators Guide* contains an overview of the **xe** CLI. Since a good deal of **xe** commands are a thin veneer over the API, playing with **xe** is a good way to start finding your way around the API object model described in this chapter.
- The code samples in the next chapter provide some concrete instances of API coding in a variety of client languages.
- The *XenEnterprise Management API* reference document provides a more detailed description of the API semantics as well as describing the the format of XML/RPC messages on the wire. and
- There are a few scripts that use the API in the XenServer Host dom0 itself. For example, "/opt/xen-source/libexec/shutdown" is a python program that cleanly shuts VMs down. This script is invoked when the host itself is shutdown.

Chapter 4. Using the API

This chapter describes how to use the XenServer Management API from real programs to manage XenServer Hosts and VMs. The chapter begins with a walk-through of a typical client application and demonstrates how the API can be used to perform common tasks. Example code fragments are given in python syntax but equivalent code in C and C# would look very similar. The language bindings themselves are discussed afterwards and the chapter finishes with walk-throughs of two complete examples included in the SDK.

4.1. Anatomy of a typical application

This section describes the structure of a typical application using the XenServer Management API. Most client applications begin by connecting to a XenServer Host and authenticating (e.g. with a username and password). Assuming the authentication succeeds, the server will create a "session" object and return a reference to the client. This reference will be passed as an argument to all future API calls. Once authenticated, the client may search for references to other useful objects (e.g. XenServer Hosts, VMs, etc.) and invoke operations on them. Operations may be invoked either synchronously or asynchronously; special task objects represent the state and progress of asynchronous operations. These application elements are all described in detail in the following sections.

4.1.1. Choosing a low-level transport

API calls can be issued over two transports:

- SSL-encrypted TCP on port 443 (https) over an IP network
- plaintext over a local Unix domain socket: `/var/xapi/xapi`

The SSL-encrypted TCP transport is used for all off-host traffic while the Unix domain socket can be used from services running directly on the XenServer Host itself. In the SSL-encrypted TCP transport, all API calls should be directed at the Resource Pool master; failure to do so will result in the error `HOST_IS_SLAVE`, which includes the IP address of the master as an error parameter.

Note

As a special-case, all messages sent through the Unix domain socket are transparently forwarded to the correct node.

4.1.2. Authentication and session handling

The vast majority of API calls take a session reference as their first parameter; failure to supply a valid reference will result in a `SESSION_INVALID` error being returned. A session reference is acquired by supplying a username and password to the **login_with_password** function.

Note

As a special-case, if this call is executed over the local Unix domain socket then the username and password are ignored and the call always succeeds.

Every session has an associated "last active" timestamp which is updated on every API call. The server software currently has a built-in limit of 200 active sessions and will remove those with the oldest "last active" field if this limit is exceeded. In addition all sessions whose "last active" field is older than 24 hours are also removed. Therefore it is important to:

- Remember to log out of active sessions to avoid leaking them; and
- Be prepared to log in again to the server if a *SESSION_INVALID* error is caught.

In the following fragment a connection via the Unix domain socket is established and a session created:

```
import XenAPI

session = XenAPI.xapi_local()
try:
    session.xenapi.login_with_password("root", "")
    ...
finally:
    session.xenapi.session.logout()
```

4.1.3. Finding references to useful objects

Once an application has authenticated the next step is to acquire references to objects in order to query their state or invoke operations on them. All objects have a set of "implicit" messages which include the following:

- **get_by_name_label**: return a list of all objects of a particular class with a particular label;
- **get_by_uuid**: return a single object named by its UUID;
- **get_all**: return a set of references to all objects of a particular class; and
- **get_all_records**: return a map of reference to records for each object of a particular class.

For example, to list all hosts:

```
hosts = session.xenapi.host.get_all()
```

To find all VMs with the name "my first VM":

```
vms = session.xenapi.VM.get_by_name_label('my first VM')
```

Note

Object *name_label* fields are not guaranteed to be unique and so the **get_by_name_label** API call returns a set of references rather than a single reference.

In addition to the methods of finding objects described above, most objects also contain references to other objects within fields. For example it is possible to find the set of VMs running on a particular host by calling:

```
vms = session.xenapi.host.get_resident_VMs(host)
```

4.1.4. Invoking synchronous operations on objects

Once object references have been acquired, operations may be invoked on them. For example to start a VM:

```
session.xenapi.VM.start(vm, False, False)
```

All API calls are by default synchronous and will not return until the operation has completed or failed. For example in the case of **VM.start** the call does not return until the VM has started booting.

Note

When the **VM.start** call returns the VM will be booting. To determine when the booting has finished, wait for the in-guest agent to report internal statistics through the **VM_guest_metrics** object

4.1.5. Using Tasks to manage asynchronous operations

To simplify managing operations which take quite a long time (e.g. **VM.clone** and **VM.copy**) functions are available in two forms: synchronous (the default) and asynchronous. Each asynchronous function returns a reference to a task object which contains information about the in-progress operation including:

- whether it is pending; has succeeded or failed;
- progress in the range 0-1; and
- the result or error code returned by the operation.

An application which wanted to track the progress of a **VM.clone** operation and display a progress bar would have code like the following:

```
vm = session.xenapi.VM.get_by_name_label('my vm')
task = session.xenapi.Async.VM.clone(vm)
while session.xenapi.Task.get_status(task) == "pending":
        progress = session.xenapi.Task.get_progress(task)
        update_progress_bar(progress)
        time.sleep(1)
session.xenapi.Task.destroy(task)
```

Note

Note that a well-behaved client should remember to delete tasks created by asynchronous operations when it has finished reading the result or error. If the number of tasks exceeds a built-in threshold then the server will delete the oldest of the completed tasks.

4.1.6. Subscribing to and listening for events

With the exception of the task and metrics classes, whenever an object is modified the server generates an event. Clients can subscribe to this event stream on a per-class basis and receive updates rather than resorting to frequent polling. Events come in three types:

- *add*: generated when an object has been created;
- *del*: generated immediately before an object is destroyed; and
- *mod*: generated when an object's field has changed.

Events also contain a monotonically increasing ID, the name of the class of object and a snapshot of the object state equivalent to the result of a **get_record()**.

Clients register for events by calling **event.register()** with a list of class names or the special string "*". Clients receive events by executing **event.next()** which blocks until events are available and returns the new events.

Note

Since the queue of generated events on the server is of finite length a very slow client might fail to read the events fast enough; if this happens an *EVENTS_LOST* error is returned. Clients should be prepared to handle this by re-registering for events and checking that the condition they are waiting for hasn't become true while they were unregistered.

The following python code fragment demonstrates how to print a summary of every event generated by a system: (similar code exists in */SDK/client-examples/python/watch-all-events.py*)

```
fmt = "%8s  %20s  %5s  %s"
session.xenapi.event.register(["*"])
while True:
    try:
        for event in session.xenapi.event.next():
            name = "(unknown)"
            if "snapshot" in event.keys():
                snapshot = event["snapshot"]
                if "name_label" in snapshot.keys():
                    name = snapshot["name_label"]
            print fmt % (event['id'], event['class'], event['operation'], name)
    except XenAPI.Failure, e:
        if e.details == [ "EVENTS_LOST" ]:
            print "Caught EVENTS_LOST; should reregister"
```

4.2. Language bindings

Although it is possible to write applications which use the XenServer Management API directly through raw XML-RPC calls, the task of developing third-party applications is greatly simplified through the use of a *language binding* which exposes the individual API calls as first-class functions in the target language. The SDK includes language bindings and example code for the C, C# and python programming languages and for both Linux and Windows clients.

4.2.1. C

The SDK includes the source to the C language binding in the directory */SDK/client-examples/c* together with a Makefile which compiles the binding into a library. Every API object is associated with a header file which contains declarations for all that object's API functions; for example the type definitions and functions required to invoke VM operations are all contained with *xen_vm.h*.

C binding dependencies

Platform supported:	Linux
Library:	The language binding is generated as a "libxen.a" that is linked by C programs.

Dependencies	• XML-RPC library (libxml2.so on GNU Linux) • Curl library (libcurl2.so)

One simple example is included within the SDK called **test_vm_ops**. The example demonstrates how to query the capabilities of a host, create a VM, attach a fresh blank disk image to the VM and then perform various powercycle operations.

4.2.2. C#

The C# bindings are contained within the directory */SDK/client-examples/csharp/XenSdk.net* and include project files suitable for building under Microsoft Visual Studio. Every API object is associated with one C# file; for example the functions implementing the VM operations are contained within the file *VM.cs*.

C# binding dependencies

Platform supported:	Windows with .NET version 2.0
Library:	The language binding is generated as a Dynamic Link Library Xenapi.dll that is linked by C# programs
Dependencies	CookComputing.XMLRpcV2.dll is needed for the xenapi.dll to be able to communicate with the xml-rpc server.

Two simple examples are included with the C# bindings in the directory */SDK/client-examples/csharp/XenSdk.net*:

- **VM-Lifecycle**: logs into a XenServer Host, lists all the VM records, filters out the templates, clones a VM from one template, configures the name and description of the VM before finally power-cycling the VM; and
- **Monitor**: logs into a XenServer Host, queries properties of a host, lists all Storage Repositories, lists all VMs and prints various attributes.

4.2.3. Python

The python bindings are contained within a single file: */SDK/client-examples/python/XenAPI.py*.

Python binding dependencies

Platform supported:	Linux
Library:	XenAPI.py
Dependencies	None

The SDK includes 7 python examples:

- **fixpbds.py**: reconfigures the settings used to access shared storage;
- **install.py**: installs a Debian VM, connects it to a network, starts it up and waits for it to report its IP address;
- **license.py**: uploads a fresh license to a XenServer Host;

- **permute.py**: selects a set of VMs and uses XenMotion to move them simultaneously between hosts;
- **powercycle.py**: selects a set of VMs and powercycles them;
- **shell.py**: a simple interactive shell for testing;
- **vm_start_async.py**: demonstrates how to invoke operations asynchronously; and
- **watch-all-events.py**: registers for all events and prints details when they occur.

4.2.4. Command Line Interface (CLI)

Rather than using raw XML-RPC or one of the supplied language bindings, third-party software developers may instead integrate with XenServer Hosts by using the XE CLI **xe.exe**.

CLI dependencies

Platform supported:	Linux and Windows
Library:	None
Binary:	xe[.exe]
Dependencies	None

The CLI allows almost every API call to be directly invoked from a script or other program, silently taking care of the required session management. The XE CLI syntax and capabilities are described in detail in Chapter 5 of the XenServer Administrator's Guide. The SDK contains 3 example **bash** shell scripts which demonstrate CLI usage. These are:

- **install-debian**: installs a Debian Etch 4.0 VM, adds a network interface, starts it booting and waits for the IP address to be reported;
- **clone-vms**: shuts down a VM if it is running, clones it and starts it up again; and
- **suspend-resume**: suspends a running VM and then resumes it.

> **Note**
>
> When running the CLI from a XenServer Host console, tab-completion of both command names and arguments is available.

4.3. Complete application examples

This section describes two complete examples of real programs using the API. The application source code is contained within the SDK.

4.3.1. Simultaneously migrating VMs using XenMotion

This python example (contained in `/SDK/client-examples/python/permute.py`) demonstrates how to use XenMotion to move VMs simultaneously between hosts in a Resource Pool. The example makes use of asynchronous API calls and shows how to wait for a set of tasks to complete.

The program begins with some standard boilerplate and imports the API bindings module

```
import sys, time

import XenAPI
```

Next the commandline arguments containing a server URL, username, password and a number of iterations are parsed. The username and password are used to establish a session which is passed to the function *`main`*, which is called multiple times in a loop. Note the use of *`try: finally:`* to make sure the program logs out of its session at the end.

```
if __name__ == "__main__":
    if len(sys.argv) <> 5:
        print "Usage:"
        print sys.argv[0], " <url> <username> <password> <iterations>"
        sys.exit(1)
    url = sys.argv[1]
    username = sys.argv[2]
    password = sys.argv[3]
    iterations = int(sys.argv[4])
    # First acquire a valid session by logging in:
    session = XenAPI.Session(url)
    session.xenapi.login_with_password(username, password)
    try:
        for i in range(iterations):
            main(session, i)
    finally:
        session.xenapi.session.logout()
```

The *`main`* function examines each running VM in the system, taking care to filter out *control domains* (which are part of the system and not controllable by the user). A list of running VMs and their current hosts is constructed.

```
def main(session, iteration):
    # Find a non-template VM object
    all = session.xenapi.VM.get_all()
    vms = []
    hosts = []
    for vm in all:
        record = session.xenapi.VM.get_record(vm)
        if not(record["is_a_template"]) and \
           not(record["is_control_domain"]) and \
           record["power_state"] == "Running":
            vms.append(vm)
            hosts.append(record["resident_on"])
    print "%d: Found %d suitable running VMs" % (iteration, len(vms))
```

Next the list of hosts is rotated:

```
# use a rotation as a permutation
hosts = [hosts[-1]] + hosts[:(len(hosts)-1)]
```

Each VM is then moved via XenMotion to the new host under this rotation (i.e. a VM running on host at position 2 in the list will be moved to the host at position 1 in the list etc.) In order to execute each of the movements in parallel, the asynchronous version of the **VM.pool_migrate** is used and a list of task references constructed. Note the *live* flag passed to the **VM.pool_migrate**; this causes the VMs to be moved while they are still running.

```
tasks = []
for i in range(0, len(vms)):
    vm = vms[i]
    host = hosts[i]
    task = session.xenapi.Async.VM.pool_migrate(vm, host, { "live": "true" })
    tasks.append(task)
```

The list of tasks is then polled for completion:

```
finished = False
records = {}
while not(finished):
    finished = True
    for task in tasks:
        record = session.xenapi.task.get_record(task)
        records[task] = record
        if record["status"] == "pending":
            finished = False
    time.sleep(1)
```

Once all tasks have left the *pending* state (i.e. they have successfully completed, failed or been cancelled) the tasks are polled once more to see if they all succeeded:

```
allok = True
for task in tasks:
    record = records[task]
    if record["status"] <> "success":
        allok = False
```

If any one of the tasks failed then details are printed, an exception is raised and the task objects left around for further inspection. If all tasks succeeded then the task objects are destroyed and the function returns.

```
    if not(allok):
        print "One of the tasks didn't succeed at", \
            time.strftime("%F:%HT%M:%SZ", time.gmtime())
        idx = 0
        for task in tasks:
            record = records[task]
            vm_name = session.xenapi.VM.get_name_label(vms[idx])
            host_name = session.xenapi.host.get_name_label(hosts[idx])
            print "%s : %12s %s -> %s [ status: %s; result = %s; error = %s ]" % \
                  (record["uuid"], record["name_label"], vm_name, host_name,      \
                   record["status"], record["result"], repr(record["error_info"]))
            idx = idx + 1
        raise "Task failed"
    else:
        for task in tasks:
            session.xenapi.task.destroy(task)
```

4.3.2. Cloning a VM via the XE CLI

This example (contained in */SDK/client-examples/bash-cli/clone-vms*) is a **bash** script which uses the XE CLI to clone a VM taking care to shut it down first if it is powered on.

The example begins with some boilerplate which first checks if the environment variable *XE* has been set: if it has it assumes that it points to the full path of the CLI, else it is assumed that the XE CLI is on the current path. Next the script prompts the user for a server name, username and password:

```
# Allow the path to the 'xe' binary to be overridden by the XE environment varia
ble
if [ -z "${XE}" ]; then
  XE=xe
fi

if [ ! -e "${HOME}/.xe" ]; then
  read -p "Server name: " SERVER
  read -p "Username: " USERNAME
  read -p "Password: " PASSWORD
  XE="${XE} -s ${SERVER} -u ${USERNAME} -pw ${PASSWORD}"
fi
```

Next the script checks its commandline arguments. It requires exactly one: the UUID of the VM which is to be cloned:

```
# Check if there's a VM by the uuid specified
${XE} vm-list params=uuid | grep -q " ${vmuuid}$"
if [ $? -ne 0 ]; then
        echo "error: no vm uuid \"${vmuuid}\" found"
        exit 2
fi
```

The script then checks the power state of the VM and if it is running, it attempts a clean shutdown. The event system is used to wait for the VM to enter state "Halted".

Note

The XE CLI supports a command-line argument *--minimal* which causes it to print its output without excess whitespace or formatting, ideal for use from scripts. If multiple values are returned they are comma-separated.

```
# Check the power state of the vm
name=$(${XE} vm-list uuid=${vmuuid} params=name-label --minimal)
state=$(${XE} vm-list uuid=${vmuuid} params=power-state --minimal)
wasrunning=0

# If the VM state is running, we shutdown the vm first
if [ "${state}" = "running" ]; then
        ${XE} vm-shutdown uuid=${vmuuid}
        ${XE} event-wait class=vm power-state=halted uuid=${vmuuid}
        wasrunning=1
fi
```

The VM is then cloned and the new VM has its *name_label* set to *cloned_vm*.

```
# Clone the VM
newuuid=$(${XE} vm-clone uuid=${vmuuid} new-name-label=cloned_vm)
```

Finally, if the original VM had been running and was shutdown, both it and the new VM are started.

```
# If the VM state was running before cloning, we start it again
# along with the new VM.
if [ "$wasrunning" -eq 1 ]; then
        ${XE} vm-start uuid=${vmuuid}
        ${XE} vm-start uuid=${newuuid}
fi
```

Chapter 5. XenServer API extensions

The XenAPI is a general and comprehensive interface to managing the life-cycles of Virtual Machines, and offers a lot of flexibility in the way that XenAPI providers may implement specific functionality (e.g. storage provisioning, or console handling). XenServer has several extensions which provide useful functionality used in our own XenCenter interface. The workings of these mechanisms are described in this chapter.

Extensions to the XenAPI are often provided by specifying `other_config` map keys to various objects. The use of this parameter indicates that the functionality is supported for that particular release of XenServer, but *not* as a long-term feature. We are constantly evaluating promoting functionality into the API, but this requires the nature of the interface to be well-understood. Developer feedback as to how you are using some of these extensions is always welcome to help us make these decisions.

5.1. VM console forwarding

Most XenAPI graphical interfaces will want to gain access to the VM consoles, in order to render them to the user as if they were physical machines. There are several types of consoles available, depending on the type of guest or if the physical host console is being accessed:

Console access

Operating System	Text	Graphical	Optimized graphical
Windows	No	VNC, via API call	RDP, directly from guest
Linux	Yes, through VNC and an API call	No	VNC, directly from guest
Physical Host	Yes, through VNC and an API call	No	No

Hardware-assisted VMs, such as Windows, directly provide a graphical console via VNC. There is no text-based console, and guest networking is not necessary to use the graphical console. Once guest networking has been established, it is more efficient to setup Remote Desktop Access and use an RDP client to connect directly (this must be done outside of the XenAPI).

Paravirtual VMs, such as Linux guests, provide a native text console directly. XenServer provides a utility (called **vncterm**) to convert this text-based console into a graphical VNC representation. Guest networking is not necessary for this console to function. As with Windows above, Linux distributions often configure VNC within the guest, and directly connect to it via a guest network interface.

The physical host console is only available as a `vt100` console, which is exposed through the XenAPI as a VNC console by using **vncterm** in the control domain.

RFB (Remote Framebuffer) is the protocol which underlies VNC, specified in The RFB Protocol. Third-party developers are expected to provide their own VNC viewers, and many freely available implementations can be adapted for this purpose. RFB 3.3 is the minimum version which viewers must support.

5.1.1. Retrieving VNC consoles via the API

VNC consoles are retrieved via a special URL passed through to the host agent. The sequence of API calls is as follows:

1. Client to Master/443: XML-RPC: **Session.login_with_password(...)**.

2. Master/443 to Client: Returns a session reference to be used with subsequent calls.

3. Client to Master/443: XML-RPC: **VM.get_by_name_label(...)**.

4. Master/443 to Client: Returns a reference to a particular VM (or the "control domain" if you want to retrieve the physical host console).

5. Client to Master/443: XML-RPC: **VM.get_consoles(...)**.

6. Master/443 to Client: Returns a list of console objects associated with the VM.

7. Client to Master/443: XML-RPC: **VM.get_location(...)**.

8. Returns a URI describing where the requested console is located. The URIs are of the form: https://192.168.0.1/console?ref=OpaqueRef:c038533a-af99-a0ff-9095-c1159f2dc6a0.

9. Client to 192.168.0.1: HTTP CONNECT "/console?ref=(...)"

The final HTTP CONNECT is slightly non-standard since the HTTP/1.1 RFC specifies that it should only be a host and a port, rather than a URL. Once the HTTP connect is complete, the connection can subsequently directly be used as a VNC server without any further HTTP protocol action.

This scheme requires direct access from the client to the control domain's IP, and will not work correctly if there are Network Address Translation (NAT) devices blocking such connectivity. You can use the CLI to retrieve the console URI from the client and perform a connectivity check.

Procedure 5.1. To retrieve a console URI via the CLI

1. Retrieve the VM UUID via:

```
xe vm-list params=uuid --minimal name-label=name
```

2. Retrieve the console information via:

```
xe console-list vm-uuid=uuid
uuid ( RO): 714f388b-31ed-67cb-617b-0276e35155ef
vm-uuid ( RO): 8acb7723-a5f0-5fc5-cd53-9f1e3a7d3069
vm-name-label ( RO): etch
protocol ( RO): RFB
location ( RO): https://192.168.0.1/console?ref=(...)
```

Use command-line utilities like **ping** to test connectivity to the IP address provided in the *location* field.

5.1.2. Disabling VNC forwarding for Linux VM

When creating and destroying Linux VMs, the host agent automatically manages the **vncterm** processes which convert the text console into VNC. Advanced users who wish to directly access the text console can disable VNC forwarding for that VM. The text console can then only be accessed directly from the control domain directly, and graphical interfaces such as XenCenter will not be able to render a console for that VM.

Procedure 5.2. To disable a Linux VNC console using the CLI:

1. Before starting the guest, set the following parameter on the VM record:

```
xe vm-param-set uuid=uuid disable_pv_vnc=1
```

2. Start the VM.

3. Use the CLI to retrieve the underlying domain ID of the VM with:

```
xe vm-list params=dom-id uuid=uuid --minimal
```

4. On the host console, connect to the text console directly by:

```
/usr/lib/xen/bin/xenconsole domid
```

This configuration is an advanced procedure, and we do not recommend that the text console is directly used for heavy I/O operations. Instead, connect to the guest via SSH or some other network-based connection mechanism.

5.2. Paravirtual Linux installation

The installation of paravirtual Linux guests is complicated by the fact that a Xen-aware kernel must be booted, rather than simply installing the guest via hardware-assistance. This does have the benefit of providing near-native installation speed due to the lack of emulation overhead. XenServer supports the installation of several different Linux distributions, and abstracts this process as much as possible.

To this end, a special bootloader known as **eliloader** is present in the control domain which reads various *other_config* keys in the VM record at start time and performs distribution-specific installation behaviour.

- *install-repository*: Required. Path to a repository; 'http', 'https', 'ftp', or 'nfs'. Should be specified as would be used by the target installer, but not including prefixes, e.g. method=.
- *install-vnc*: Default: false. Use VNC where available during the installation.
- *install-vncpasswd*: Default: empty. The VNC password to use, when providng one is possible via the command-line of the target distribution.
- *install-round*: Default: 1. The current bootloader round. Not to be edited by the user (see below)

5.2.1. Red Hat Enterprise Linux 4.1/4.4

eliloader is used for two rounds of booting. In the first round, it returns the installer `initrd` and kernel from `/opt/xensource/packages/files/guest-installer`. Then, on the second boot, it removes the additional updates disk from the VM, switches the bootloader to **pygrub**, and then begins a normal boot.

This sequence is required since Red Hat does not provide a Xen kernel for these distributions, and so the XenServer custom kernels for those distributions are used instead.

5.2.2. Red Hat Enterprise Linux 4.5/5.0

Similar to the RHEL4.4 installation, except that the kernel and ramdisk are downloaded directly form the network repository that was specified by the user, and switch the bootloader to **pygrub** immediately. Note that pygrub is not executed immediately, and so will only be parsed on the next boot.

The network retrieval enables users to install the upstream Red Hat vendor kernel directly from their network repository. An updated XenServer kernel is also provided on the `xs-tools.iso` built-in ISO image which fixes various Xen-related bugs.

5.2.3. SUSE Enterprise Linux 10 SP1

This requires a two-round boot process. The first round downloads the kernel and ramdisk from the network repository and boots them. The second round then inspects the disks to find the installed kernel and ramdisk, and sets the *`PV-bootloader-args`* to reflect these paths within the guest filesystem. This process emulates the **domUloader** which SUSE use as an alternative to **pygrub**. Finally, the bootloader is set to **pygrub** and is executed to begin a normal boot.

The SLES 10 installation method means that the path for the kernel and ramdisk is stored in the VM record rather than in the guest `menu.lst`, but this is the only way it would ever work since the YAST package manager doesn't write a valid `menu.lst`.

5.2.4. CentOS 4.5/5.0

The CentOS installation mechanism is similar to that of the Red Hat installation notes above, save that some MD5 checksums are different which **eliloader** recognizes.

5.3. Adding Xenstore entries to VMs

Developers may wish to install guest agents into VMs which take special action based on the type of the VM. In order to communicate this information into the guest, a special Xenstore name-space known as *`vm-data`* is available which is populated at VM creation time. It is populated via the *`xenstore_data`* map in the VM record.

Procedure 5.3. To populate a Xenstore node *`foo`* in a VM

1. Set the *`xenstore_data`* parameter in the VM record:

   ```
   xe vm-param-set uuid=vm-uuid xenstore_data:vm-data/foo=bar
   ```

2. Start the VM.

3. If it is a Linux-based VM, install the guest tools and use the **xenstore-read** to verify that the node exists in Xenstore.

Note that only prefixes beginning with *`vm-data`* are permitted, and anything not in this name-space will be silently ignored when starting the VM.

5.4. Security enhancements

The control domain in XenServer 4.1.0 and above has various security enhancements in order to harden it against attack from malicious guests. Developers should never notice any loss of correct functionality as a result of these changes, but they are documented here as variations of behaviour from other distributions.

- The control domain privileged user-space interfaces can now be restricted to only work for certain domains. There are three interface affected by this change:
  - The **xenstored** socket interface, access via `libxenstore`. Interfaces are restricted via `xs_restrict()`.

  - The device `/dev/xen/evtchn`, which is accessed via `xs_evtchn_open()` in `libxenctrl`. A handle can be restricted using `xs_evtchn_restrict()`.
  - The device `/proc/xen/privcmd`, accessed through `xs_interface_open()` in `libxenctrl`. A handle is restricted using `xc_interface_restrict()`. Some privileged commands are naturally hard to restrict (e.g. the ability to make arbitrary hypercalls), and these are simply prohibited on restricted handles.
  - A restricted handle cannot later be granted more privilege, and so the interface must be closed and reopened. Security is only gained if the process cannot subsequently open more handles.
- The **qemu** device emulation processes and **vncterm** terminal emulation processes run as a non-root user ID and are restricted into an empty directory. They uses the restriction API above to drop privileges where possible.
- Access to xenstore is rate-limited to prevent malicious guests from causing a denial of service on the control domain. This is implemented as a token bucket with a restricted fill-rate, where most operations take one token and opening a transaction takes 20. The limits are set high enough that they should never be hit when running even a large number of concurrent guests under loaded operation.
- The VNC guest consoles are bound only to the `localhost` interface, so that they are not exposed externally even if the control domain packet filter is disabled by user intervention.

5.5. Advanced settings for network interfaces

Virtual and physical network interfaces have some advanced settings that can be configured using the *other-config* map parameter. There are a set of custom ethtool settings and some miscellaneous settings.

5.5.1. ethtool settings

Developers might wish to configure custom ethtool settings for physical and virtual network interfaces. This is accomplished with *ethtool-<option>* keys via the *other-config* map parameter.

Key	Description	Valid settings
ethtool-rx	Specify if RX checksumming is enabled	*on* or *true* to enable the setting, *off* or *false* to disable it
ethtool-tx	Specify if TX checksumming is enabled	*on* or *true* to enable the setting, *off* or *false* to disable it
ethtool-sg	Specify if scatter-gather is enabled	*on* or *true* to enable the setting, *off* or *false* to disable it
ethtool-tso	Specify if tcp segmentation offload is enabled	*on* or *true* to enable the setting, *off* or *false* to disable it
ethtool-ufo	Specify if UDP fragmentation offload is enabled	*on* or *true* to enable the setting, *off* or *false* to disable it
ethtool-gso	Specify if generic segmentation offload is enabled	*on* or *true* to enable the setting, *off* or *false* to disable it
ethtool-autoneg	Specify if autonegotiation is enabled	*on* or *true* to enable the setting, *off* or *false* to disable it
ethtool-speed	Set the device speed in Mb/s	10, 100. or 1000
ethtool-duplex	Set full or half duplex mode	half or full

For example, to enable TX checksumming on a virtual NIC via the xe CLI:

```
xe vif-param-set uuid=<VIF UUID> other-config:ethtool-tx="on"
```

or:

```
xe vif-param-set uuid=<VIF UUID> other-config:ethtool-tx="true"
```

To set the duplex setting on a physical NIC to half duplex via the xe CLI:

```
xe vif-param-set uuid=<VIF UUID> other-config:ethtool-duplex="half"
```

5.5.2. Miscellaneous settings

You can also set a VIF or PIF to enable promiscuous mode via the *promiscuous* key. For example, to enable promiscuous mode on a physical NIC via the xe CLI:

```
xe pif-param-set uuid=<PIF UUID> other-config:promiscuous="on"
```

or:

```
xe pif-param-set uuid=<PIF UUID> other-config:promiscuous="true"
```

The VIF and PIF objects have a *MTU* parameter that is read-only and provide the current setting of the maximum transmission unit for the interface. You can override the default maximum transmission unit of a physical or virtual NIC with the *mtu* key via the *other-config* map parameter. For example, to reset the MTU on a virtual NIC to use jumbo frames via the xe CLI:

```
xe vif-param-set uuid=<VIF UUID> other-config:mtu=9000
```

Note that changing the MTU of underlying interfaces is an advanced and experimental feature, and may lead to unexpected side-effects if you have varying MTUs across NICs in a single resource pool.

5.6. Internationalization for SR names

The SRs created at install time now have an *other_config* key indicating how their names may be internationalized.

other_config["i18n-key"] may be one of

- *local-hotplug-cd*
- *local-hotplug-disk*

- *local-storage*
- *xenserver-tools*

Additionally, *other_config["i18n-original-value-<field name>"]* gives the value of that field when the SR was created. If XenCenter sees a record where *SR.name_label* equals *other_config["i18n-original-value-name_label"]* (that is, the record has not changed since it was created during XenServer installation), then internationalization will be applied. In other words, XenCenter will disregard the current contents of that field, and instead use a value appropriate to the user's own language.

If you change *SR.name_label* for your own purpose, then it no longer is the same as *other_config["i18n-original-value-name_label"]*. Therefore, XenCenter does not apply internationalization, and instead preserves your given name.

5.7. Hiding objects from XenCenter

Networks, PIFs, and VMs can be hidden from XenCenter by adding the key *HideFromXenCenter* set to *true* to the *other_config* parameter for the object. This capability is intended for ISVs who know what they are doing, not general use by everyday users. For example, you might want to hide certain VMs because they are cloned VMs that shouldn't be used directly by general users in your environment.

In XenCenter, hidden Networks, PIFs, and VMs can be made visible, using the View menu.

Index