Xen Management API Draft

Version: API Revision 0.2
Date: 22 June, 2006
Private Preview Release
DO NOT CIRCULATE

Ewan Mellor: `ewan@xensource.com`
Richard Sharp: `richard.sharp@xensource.com`
David Scott: `david.scott@xensource.com`

Chapter 1

Introduction

This document contains a proposal for a Xen Management API—an interface for remotely configuring and controlling virtualised guests running on a Xen-enabled host.

This document is an early draft for discussion purposes only

The API is presented here as a set of Remote Procedure Calls, with a wire format based on XML-RPC. No specific language bindings are prescribed, although examples will be given in the python programming language.

Although we adopt some terminology from object-oriented programming, future client language bindings may or may not be object oriented. The API reference uses the terminology *classes* and *objects*. For our purposes a *class* is simply a hierarchical namespace; an *object* is an instance of a class with its fields set to specific values. Objects are persistent and exist on the server-side. Clients may obtain opaque references to these server-side objects and then access their fields via get/set RPCs.

For each class we specify a list of fields along with their *types* and *qualifiers*. A qualifier is one of:

- RO_{run}: the field is Read Only. Furthermore, its value is automatically computed at runtime. For example: current CPU load and disk IO throughput.
- RO_{ins}: the field must be manually set when a new object is created, but is then Read Only for the duration of the object's life. For example, the maximum memory addressable by a guest is set before the guest boots.
- *RW*: the field is Read/Write. For example, the name of a VM.

A full list of types is given in Chapter 2. However, there are three types that require explicit mention:

- t *Ref*: signifies a reference to an object of type t.
- t *Set*: signifies a set containing values of type t.
- (t_1, t_2) *Map*: signifies a mapping from values of type t_1 to values of type t_2.

Note that there are a number of cases where *Ref*s are *doubly linked*—e.g. a VM has a field called `groups` of type (*VMGroup Ref*) *Set*; this field lists the VMGroups that a particular VM is part of. Similarly, the VMGroups class has a field called `VMs` of type (*VM Ref*) *Set* that contains the VMs that are part of a particular VMGroup. These two fields are *bound together*, in the sense that adding a new VMGroup to a VM causes the VMs field of the corresponding VMGroup object to be updated automatically.

The API reference explicitly lists the fields that are bound together in this way. It also contains a diagram that shows relationships between classes. In this diagram an edge signifies the existance of a pair of fields that are bound together, using standard crows-foot notation to signify the type of relationship (e.g. one-many, many-many).

1.1 RPCs associated with fields

Each field, `f`, has an RPC accessor associated with it that returns `f`'s value:

- "`get_f(Ref x)`": takes a `Ref` that refers to an object and returns the value of `f`.

Each field, `f`, with attribute *RW* and whose outermost type is *Set* has the following additional RPCs associated with it:

- an "`add_to_f(Ref x, v)`" RPC adds a new element v to the set[1];
- a "`remove_from_f(Ref x, v)`" RPC removes element `v` from the set;

Each field, `f`, with attribute *RW* and whose outermost type is *Map* has the following additional RPCs associated with it:

- an "`add_to_f(Ref x, k, v)`" RPC adds new pair `(k, v)` to the mapping stored in `f` in object `x`. Adding a new pair for duplicate key, `k`, overwrites any previous mapping for `k`.
- a "`remove_from_f(Ref x, k)`" RPC removes the pair with key `k` from the mapping stored in `f` in object `x`.

Each field whose outermost type is neither *Set* nor *Map*, but whose attribute is *RW* has an RPC acessor associated with it that sets its value:

- For *RW* (*R*ead/*W*rite), a "`set_f(Ref x, v)`" RPC function is also provided. This sets field `f` on object `x` to value `v`.

1.2 RPCs associated with classes

- Each class has a *constructor* RPC that takes as parameters all fields marked *RW* and RO_{ins}. The result of this RPC is that a new *persistent* object is created on the server-side with the specified field values.
- Each class has a "`get_all()`" RPC that returns a set of all persistent objects of that class that the system knows about. For example, `VM.get_all()` would return a set of `VM` objects that are currently installed.
- Each class has a `get_by_uuid(uuid)` RPC that returns the object of that class that has the specified `uuid`.
- Each class that has a `short_name` field has a "`get_by_short_name(name)`" RPC that returns a set of objects of that class that have the specified `name`.
- Each class has a "`to_XML()`" RPC that serialises the state of all fields as an XML string.
- Each class has a "`delete(Ref x)`" RPC that explicitly deletes the persistent object specified by `x` from the system.

1.2.1 Additional RPCs

As well as the RPCs enumerated above, some classes have additional RPCs associated with them. For example, the `VM` class have RPCs for cloning, suspending, starting etc. Such additional RPCs are described explicitly in the API reference.

[1] Since sets cannot contain duplicate values this operation has no action in the case that `v` was already in the set.

1.3 Wire Protocol for Remote API Calls

API calls are sent over a network to a Xen-enabled host using the XML-RPC protocol. In this Section we describe how the higher-level types used in our API Reference are mapped to primitive XML-RPC types.
In our API Reference we specify the signatures of API functions in the following style:

```
(vm_id Set)   Host.ListAllVMs()
```

This specifies that the function with name `Host.ListAllVMs` takes no parameters and returns a Set of `vm_ids`. These types are mapped onto XML-RPC types in a straight-forward manner:

- all our "`_id`" types (e.g. `vm_id` in the above example) map to XML-RPC's `String` type.
- for all our types, `t`, type `t Set` simply maps to XML-RPC's `Array` type[2].
- our type `void` maps onto an empty XML-RPC `String`.

1.3.1 Return Values/Status Codes

The return value of an RPC call is an XML-RPC `Struct`.

- The first element of the struct is named `Status`; it contains a string value indicating whether the result of the call was a "`Success`" or a "`Failure`".

If `Status` was set to `Success` then the Struct contains a second element named `Value`:

- The element of the struct named `Value` contains the function's return value.

In the case where `Status` is set to `Failure` then the struct contains a second element named `ErrorDescription`:

- The element of the struct named `ErrorDescription` contains an array of string values. The first element of the array represents an error code; the remainder of the array represents error parameters relating to that code.

For example, an XML-RPC return value from the `Host.ListAllVMs` function above may look like this:

```
<struct>
   <member> <name> Status </name>
            <value> Success </value>
   </member>
   <member> <name> Value </name>
      <array>
         <data>
           <value> vm-id-1 </value>
           <value> vm-id-2 </value>
           <value> vm-id-3 </value>
         </data>
      </array>
   </member>
</struct>
```

[2] XML-RPC does not explicitly support a parameterised array type so we have no means of specifying the type of elements at this level.

1.4 Making XML-RPC Calls

1.4.1 Transport Layer

We ought to support at least

- HTTP/S for remote administration
- HTTP over Unix domain sockets for local administration

1.4.2 Session Layer

The XML-RPC interface is session-based; before you can make arbitrary RPC calls you must login and initiate a session. For example:

```
session_id    Session.login_with_password(string uname, string pwd)
```

Where `uname` and `password` refer to your username and password respectively, as defined by the Xen administrator. The `session_id` returned by `Session.Login` is passed to subequent RPC calls as an authentication token.
A session can be terminated with the `Session.Logout` function:

```
void          Session.Logout(session_id session)
```

1.4.3 Synchronous and Asynchronous invocation

Each method call (apart from those on "Session" and "Task" objects) can be made either synchronously or asynchronously. A synchronous RPC call blocks until the return value is received; the return value of a synchronous RPC call is exactly as specified in Section 1.3.1.
Each of the methods specified in the API Reference is synchronous. However, although not listed explicitly in this document, each method call has an asynchronous analogue in the `Async` namespace. For example, synchronous call `VM.Install(...)` (described in Chapter 2) has an asynchronous counterpart, `Async.VM.Install(...)`, that is non-blocking.
Instead of returning its result directly, an asynchronous RPC call returns a `task-id`; this identifier is subsequently used to track the status of a running asynchronous RPC. Note that an asychronous call may fail immediately, before a `task-id` has even been created—to represent this eventuality, the returned `task-id` is wrapped in an XML-RPC struct with a `Status`, `ErrorDescription` and `Value` fields, exactly as specified in Section 1.3.1.
The `task-id` is provided in the `Value` field if `Status` is set to `Success`.
Two special RPC calls are provided to poll the status of asynchronous calls:

```
Array<task_id>  Async.Task.GetAllTasks (session_id s)
task_status     Async.Task.GetStatus   (session_id s, task_id t)
```

`Async.Task.GetAllTasks` returns a set of the currently executing asynchronous tasks belong to the current user[3].
`Async.Task.GetStatus` returns a `task_status` result. This is an XML-RPC struct with three elements:

- The first element is named `Progress` and contains an `Integer` between 0 and 100 representing the estimated percentage of the task currently completed.
- The second element is named `ETA` and contains a `DateTime` representing the estimated time the task will be complete.
- The third element is named `Result`. If `Progress` is not 100 then `Result` contains the empty string. If `Progress` *is* set to 100, then `Result` contains the function's return result (as specified in Section 1.3.1)[4].

[3] The current user is determined by the username that was provided to `Session.Login`.

[4] Recall that this itself is a struct potentially containing status, errorcode, value fields etc.

1.5 Example interactive session

This section describes how an interactive session might look, using the python XML-RPC client library.
First, initialise python and import the library `xmlrpclib`:

```
$ python2.4
...
>>> import xmlrpclib
```

Create a python object referencing the remote server:

```
>>> xen = xmlrpclib.Server("http://test:4464")
```

Acquire a session token by logging in with a username and password (error-handling ommitted for brevity; the session token is pointed to by the key `’Value’` in the returned dictionary)

```
>>> session = xen.Session.do_login_with_password("user", "passwd")[’Value’]
```

When serialised, this call looks like the following:

```
<?xml version=’1.0’?>
<methodCall>
  <methodName>Session.do_login_with_password</methodName>
  <params>
    <param>
      <value><string>user</string></value>
    </param>
    <param>
      <value><string>passwd</string></value>
    </param>
  </params>
</methodCall>
```

Next, the user may acquire a list of all the VMs known to the host: (Note the call takes the session token as the only parameter)

```
>>> all_vms = xen.VM.do_list(session)[’Value’]
>>> all_vms
[’b7b92d9e-d442-4710-92a5-ab039fd7d89b’, ’23e1e837-abbf-4675-b077-d4007989b0cc’, ’2045dbc0-0734-4eea
```

Note the VM references are internally UUIDs. Once a reference to a VM has been acquired a lifecycle operation may be invoked:

```
>>> xen.VM.do_start(session, all_vms[3], False)
{’Status’: ’Failure’, ’ErrorDescription’: ’Operation not implemented’}
```

In this case the `start` message has not been implemented and an error response has been returned. Currently these high-level errors are returned as structured data (rather than as XMLRPC faults), allowing for internationalised errors in future. Finally, here are some examples of using accessors for object fields:

```
>>> xen.VM.getname_label(session, all_vms[3])[’Value’]
’SMP’
>>> xen.VM.getname_description(session, all_vms[3])[’Value’]
’Debian for Xen’
```

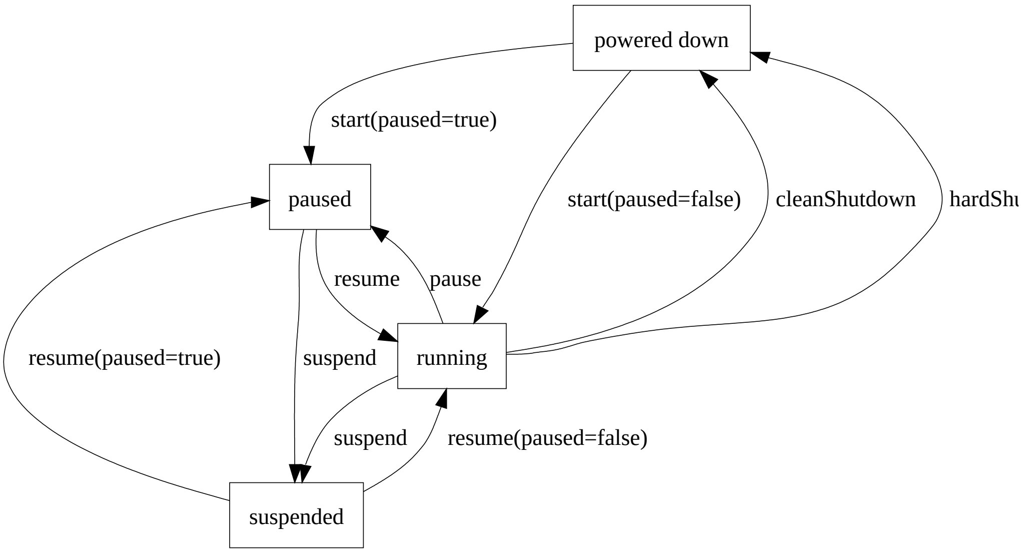


Figure 1.1: VM Lifecycle

1.6 To-Do

Lots and lots! Including:

- add places for people to store extra data ("otherConfig" perhaps)
- marking VDIs as exclusive / shareable (locking?)
- consider what happens when an object is deleted when references to it exist – do we want a cascade delete-style semantics?
- consider how to represent CDROMs (as VDIs?)
- define lists of exceptions which may be thrown by each RPC

1.7 VM Lifecycle

Figure 1.1 shows the states that a VM can be in and the API calls that can be used to move the VM between these states.

Chapter 2

API Reference

2.1 Classes

The following classes are defined:

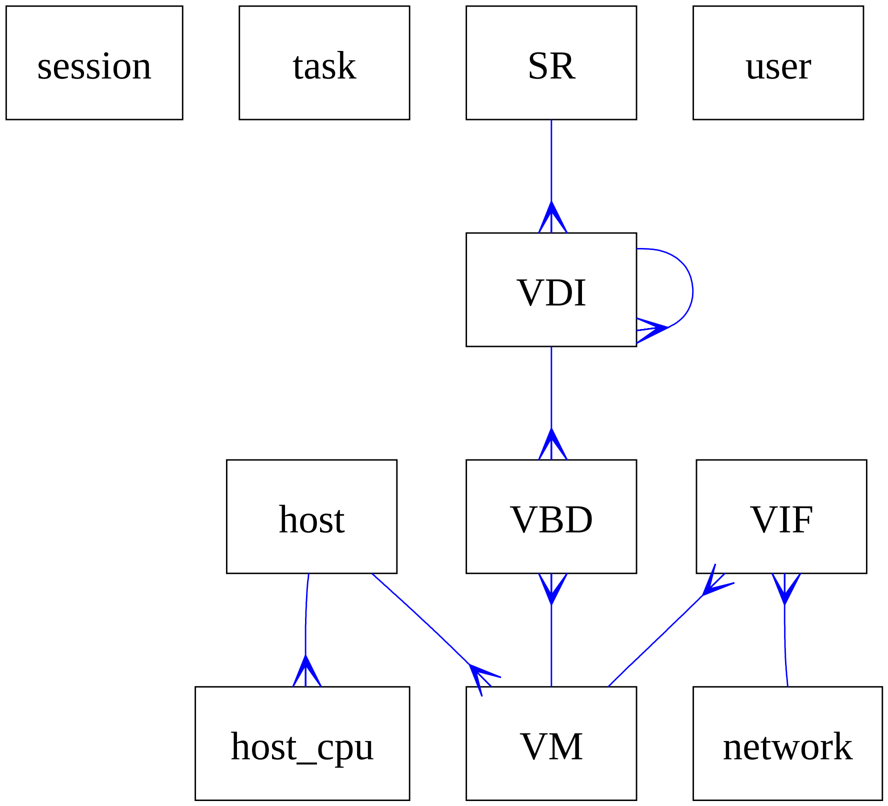
Name	Description
`session`	a session
`task`	a longrunning asynchronous task
`VM`	a virtual machine (or 'guest')
`host`	a physical host
`host_cpu`	a physical CPU
`network`	a virtual network
`VIF`	a virtual network interface
`SR`	a storage repository
`VDI`	a virtual disk image
`VBD`	a virtual block device
`user`	a user of the system

2.2 Relationships Between Classes

Fields that are bound together are shown in the following table:

object.field	*object.field*	*relationship*
VDI.VBDs	VBD.VDI	many-to-one
VDI.parent	VDI.children	one-to-many
VBD.VM	VM.VBDs	one-to-many
VIF.VM	VM.VIFs	one-to-many
VIF.network	network.VIFs	one-to-many
SR.VDIs	VDI.SR	many-to-one
host.resident_VMs	VM.resident_on	many-to-one
host.host_CPUs	host_cpu.host	many-to-one

The following represents bound fields (as specified above) diagramatically, using crows-foot notation to specify one-to-one, one-to-many or many-to-many relationships:



2.2.1 List of bound fields

2.3 Types

2.3.1 Primitives

The following primitive types are used to specify methods and fields in the API Reference:

Type	Description
String	text strings
Int	64-bit integers
Float	IEEE double-precision floating-point numbers
Bool	boolean
DateTime	date and timestamp
Ref (object name)	reference to an object of class name

2.3.2 Higher order types

The following type constructors are used:

Type	Description
List (t)	an arbitrary-length list of elements of type t
Map (a → b)	a table mapping values of type a to values of type b

2.3.3 Enumeration types

The following enumeration types are used:

enum vm_power_state	
Halted	Halted
Paused	Paused
Running	Running
Suspended	Suspended
ShuttingDown	Shutting Down
Unknown	Some other unknown state

enum on_normal_exit	
destroy	destroy the VM state
restart	restart the VM

enum on_crash_behaviour	
destroy	destroy the VM state
coredump_and_destroy	record a coredump and then destroy the VM state
restart	restart the VM
coredump_and_restart	record a coredump and then restart the VM
preserve	leave the crashed VM as-is
rename_restart	rename the crashed VM and start a new copy

enum bios_boot_option	
floppy	boot from emulated floppy
HD	boot from emulated HD
CDROM	boot from emulated CDROM

enum boot_type	
bios	boot an HVM guest using an emulated BIOS
grub	boot from inside the machine using grub
kernel_external	boot from an external kernel
kernel_internal	boot from a kernel inside the guest filesystem

enum cpu_feature	
FPU	Onboard FPU
VME	Virtual Mode Extensions
DE	Debugging Extensions
PSE	Page Size Extensions
TSC	Time Stamp Counter

MSR	Model-Specific Registers, RDMSR, WRMSR
PAE	Physical Address Extensions
MCE	Machine Check Architecture
CX8	CMPXCHG8 instruction
APIC	Onboard APIC
SEP	SYSENTER/SYSEXIT
MTRR	Memory Type Range Registers
PGE	Page Global Enable
MCA	Machine Check Architecture
CMOV	CMOV instruction (FCMOVCC and FCOMI too if FPU present)
PAT	Page Attribute Table
PSE36	36-bit PSEs
PN	Processor serial number
CLFLSH	Supports the CLFLUSH instruction
DTES	Debug Trace Store
ACPI	ACPI via MSR
MMX	Multimedia Extensions
FXSR	FXSAVE and FXRSTOR instructions (fast save and restore
XMM	Streaming SIMD Extensions
XMM2	Streaming SIMD Extensions-2
SELFSNOOP	CPU self snoop
HT	Hyper-Threading
ACC	Automatic clock control
IA64	IA-64 processor
SYSCALL	SYSCALL/SYSRET
MP	MP Capable.
NX	Execute Disable
MMXEXT	AMD MMX extensions
LM	Long Mode (x86-64)
3DNOWEXT	AMD 3DNow! extensions
3DNOW	3DNow!
RECOVERY	CPU in recovery mode
LONGRUN	Longrun power control
LRTI	LongRun table interface
CXMMX	Cyrix MMX extensions
K6_MTRR	AMD K6 nonstandard MTRRs
CYRIX_ARR	Cyrix ARRs (= MTRRs)
CENTAUR_MCR	Centaur MCRs (= MTRRs)
K8	Opteron, Athlon64
K7	Athlon
P3	P3
P4	P4
CONSTANT_TSC	TSC ticks at a constant rate
FXSAVE_LEAK	FXSAVE leaks FOP/FIP/FOP
XMM3	Streaming SIMD Extensions-3
MWAIT	Monitor/Mwait support
DSCPL	CPL Qualified Debug Store
EST	Enhanced SpeedStep
TM2	Thermal Monitor 2
CID	Context ID
CX16	CMPXCHG16B
XTPR	Send Task Priority Messages
XSTORE	on-CPU RNG present (xstore insn)
XSTORE_EN	on-CPU RNG enabled

XCRYPT	on-CPU crypto (xcrypt insn)
XCRYPT_EN	on-CPU crypto enabled
LAHF_LM	LAHF/SAHF in long mode
CMP_LEGACY	If yes HyperThreading not valid

enum vdi_type	
system	a disk that may be replaced on upgrade
user	a disk that is always preserved on upgrade
ephemeral	a disk that may be reformatted on upgrade

enum vbd_mode	
RO	disk is mounted read-only
RW	disk is mounted read-write

enum driver_type	
ioemu	use hardware emulation
paravirtualised	use paravirtualised driver

2.4 Class: session

2.4.1 Fields for class: session

Class session has no fields.

2.4.2 Additional RPCs associated with class: session

RPC name: login_with_password

Overview: Attempt to authenticate the user, returning a session_id if successful
Signature:

```
(session ref) login_with_password (string uname, string pwd)
```

Arguments:

type	name	description
`string`	uname	Username for login.
`string`	pwd	Password for login.

Return Type: `session ref`
ID of newly created session

RPC name: logout

Overview: Log out of a session
Signature:

```
void logout (session_id s)
```

Return Type: `void`

2.5 Class: task

2.5.1 Fields for class: task

Class task has no fields.

2.5.2 Additional RPCs associated with class: task

RPC name: get_status

Overview: Poll a running asynchronous RPC invocation and query its status
Signature:

```
XML get_status (session_id s, task ref task)
```

Arguments:

type	name	description
`task ref`	task	The ID of the RPC call to poll

Return Type: `XML`
XML string describing status of specified asynchronous RPC invocation, including estimated completion time

RPC name: get_all_tasks

Overview: List all asynchronous RPC calls currently executing
Signature:

```
((task ref) Set) get_all_tasks (session_id s)
```

Return Type: `(task ref) Set`
A list of tasks currently executing. Note that tasks are associated with users rather than sessions. Thus, if you logout and login again with a different session but the same user, this function will still return the user's running tasks.

2.6 Class: VM

2.6.1 Fields for class: VM

Name	**VM**		
Description	*a virtual machine (or 'guest')*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RO_{run}	`power_state`	vm_power_state	Current power state of the machine
RW	`name/label`	string	a human-readable name
RW	`name/description`	string	a notes field containg human-readable description
RW	`user_version`	int	a user version number for this machine
RW	`is_a_template`	bool	true if this is a template. Template VMs can never be started, they are used only for cloning other VMs
RO_{run}	`resident_on`	host ref	the host the VM is currently resident on
RO_{ins}	`memory/static_max`	int	Statically-set (i.e. absolute) maximum
RW	`memory/dynamic_max`	int	Dynamic maximum
RO_{run}	`memory/actual`	int	Guest's actual usage
RW	`memory/dynamic_min`	int	Dynamic minimum
RO_{ins}	`memory/static_min`	int	Statically-set (i.e. absolute) mininum
RW	`VCPUs/policy`	string	the name of the VCPU scheduling policy to be applied
RW	`VCPUs/params`	string	string-encoded parameters passed to selected VCPU policy
RO_{run}	`VCPUs/number`	int	Current number of VCPUs
RO_{run}	`VCPUs/utilisation`	(int → float) Map	Utilisation for all of guest's current VCPUs
RO_{ins}	`VCPUs/features/required`	(cpu_feature) Set	CPU features the guest demands the host supports
RO_{ins}	`VCPUs/features/can_use`	(cpu_feature) Set	CPU features the guest can use if available
RW	`VCPUs/features/force_on`	(cpu_feature) Set	CPU features to expose to the guest above the bare minimum
RW	`VCPUs/features/force_off`	(cpu_feature) Set	CPU features to hide to the guest
RW	`actions/after_shutdown`	on_normal_exit	action to take after the guest has shutdown itself
RW	`actions/after_reboot`	on_normal_exit	action to take after the guest has rebooted itself
RW	`actions/after_suspend`	on_normal_exit	action to take after the guest has suspended itself
RW	`actions/after_crash`	on_crash_behaviour	action to take if the guest crashes
RW	`VIFs`	(VIF ref) Set	virtual network interfaces
RW	`VBDs`	(VBD ref) Set	virtual block devices
RO_{ins}	`TPM/instance`	int	included for TPM support
RO_{ins}	`TPM/backend`	int	included for TPM support
RW	`bios/cdrom`	string	path for emulated CDROM e.g. /dev/cdrom or /foo.iso
RW	`bios/boot`	bios_boot_option	default device to boot the guest from

RW	`platform/std_VGA`	bool	emulate standard VGA instead of cirrus logic
RW	`platform/serial`	string	redirect serial port to pty
RW	`platform/localtime`	bool	set RTC to local time
RW	`platform/clock_offset`	bool	timeshift applied to guest's clock
RW	`platform/enable_audio`	bool	emulate audio
RW	`builder`	string	domain builder to use
RW	`boot_method`	boot_type	select how this machine should boot
RW	`kernel/kernel`	string	path to kernel e.g. /boot/vmlinuz
RW	`kernel/initrd`	string	path to the initrd e.g. /boot/initrd.img
RW	`kernel/args`	string	extra kernel command-line arguments
RW	`grub/cmdline`	string	grub command-line
RO_{ins}	`PCI_bus`	string	PCI bus path for pass-through devices
RO_{run}	`tools_version`	(string → string) Map	versions of installed paravirtualised drivers

2.6.2 Additional RPCs associated with class: VM

RPC name: clone

Overview: Clones the specified VM, making a new VM. Clone automatically exploits the capabilities of the underlying storage repository in which the VM's disk images are stored (e.g. Copy on Write). (This function can only be called when the VM is in the Halted State).
Signature:

```
(VM ref) clone (session_id s, VM ref vm, string new_name)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to be cloned
`string`	new_name	The name of the cloned VM

Return Type: `VM ref`
The ID of the newly created VM.

RPC name: start

Overview: Start the specified VM. (This function can only be called with the VM is in the Halted State).
Signature:

```
void start (session_id s, VM ref vm, bool start_paused)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to start
`bool`	start_paused	Instantiate VM in paused state if set to true.

Return Type: `void`

RPC name: pause

Overview: Pause the specified VM. This can only be called when the specified VM is in the Running state.
Signature:

```
void pause (session_id s, VM ref vm)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to pause

Return Type: `void`

RPC name: unpause

Overview: Resume the specified VM. This can only be called when the specified VM is in the Paused state.
Signature:

```
void unpause (session_id s, VM ref vm)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to pause

Return Type: `void`

RPC name: clean_shutdown

Overview: Attempt to cleanly shutdown the specified VM. (Note: this may not be supported—e.g. if a guest agent is not installed). Once shutdown has been completed perform poweroff action specified in guest configuration.
Signature:

```
void clean_shutdown (session_id s, VM ref vm)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to shutdown

Return Type: `void`

RPC name: clean_reboot

Overview: Attempt to cleanly shutdown the specified VM (Note: this may not be supported—e.g. if a guest agent is not installed). Once shutdown has been completed perform reboot action specified in guest configuration.
Signature:

```
void clean_reboot (session_id s, VM ref vm)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to shutdown

Return Type: `void`

RPC name: hard_shutdown

Overview: Stop executing the specified VM without attempting a clean shutdown. Then perform poweroff action specified in VM configuration.
Signature:

```
void hard_shutdown (session_id s, VM ref vm)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to destroy

Return Type: `void`

RPC name: hard_reboot

Overview: Stop executing the specified VM without attempting a clean shutdown. Then perform reboot action specified in VM configuration
Signature:

```
void hard_reboot (session_id s, VM ref vm)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to reboot

Return Type: `void`

RPC name: suspend

Overview: Suspend the specified VM to disk.
Signature:

```
void suspend (session_id s, VM ref vm, bool live)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to hibernate
`bool`	live	If set to true, perform a live hibernate; otherwise suspend the VM before commencing hibernate

Return Type: `void`

RPC name: resume

Overview: Awaken the specified VM and resume it.
Signature:

```
void resume (session_id s, VM ref vm, bool start_paused)
```

Arguments:

type	name	description
`VM ref`	vm	The VM to unhibernate
`bool`	start_paused	Unhibernate VM in paused state if set to true.

Return Type: `void`

RPC name: list

Overview: Return a list of all the VMs known to the system
Signature:

```
((VM ref) Set) list (session_id s)
```

Return Type: `(VM ref) Set`
A list of all the IDs of all the VMs

2.7 Class: host

2.7.1 Fields for class: host

Name	**host**		
Description	*a physical host*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RW	`name/label`	string	a human-readable name
RW	`name/description`	string	a notes field containg human-readable description
RO_{run}	`software_version`	(string → string) Map	version strings
RO_{run}	`resident_VMs`	(VM ref) Set	list of VMs currently resident on host
RO_{run}	`host_CPUs`	(host_cpu ref) Set	The physical CPUs on this host

2.7.2 Additional RPCs associated with class: host

RPC name: disable

Overview: Puts the host into a state in which no new VMs can be started. Currently active VMs on the host continue to execute.
Signature:

```
void disable (session_id s, host ref host)
```

Arguments:

type	name	description
`host ref`	host	The Host to disable

Return Type: `void`

RPC name: enable

Overview: Puts the host into a state in which new VMs can be started.
Signature:

```
void enable (session_id s, host ref host)
```

Arguments:

type	name	description
`host ref`	host	The Host to enable

Return Type: `void`

RPC name: shutdown

Overview: Shutdown the host. (This function can only be called if there are no currently running VMs on the host and it is disabled.)
Signature:

```
void shutdown (session_id s, host ref host)
```

Arguments:

type	name	description
`host ref`	host	The Host to shutdown

Return Type: `void`

RPC name: reboot

Overview: Reboot the host. (This function can only be called if there are no currently running VMs on the host and it is disabled.)
Signature:

```
 void reboot (session_id s, host ref host)
```

Arguments:

type	name	description
`host ref`	host	The Host to reboot

Return Type: `void`

RPC name: list

Overview: Return a list of all the hosts known to the system
Signature:

```
 ((host ref) Set) list (session_id s)
```

Return Type: `(host ref) Set`
A list of all the IDs of all the hosts

2.8 Class: host_cpu

2.8.1 Fields for class: host_cpu

Name	**host_cpu**		
Description	*a physical CPU*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RO_{ins}	`host`	host ref	the host the CPU is in
RO_{ins}	`number`	int	the number of the physical CPU within the host
RO_{ins}	`features`	(cpu_feature) Set	the features supported by the CPU
RO_{run}	`utilisation`	float	the current CPU utilisation

2.8.2 Additional RPCs associated with class: host_cpu

Class host_cpu has no additional RPCs associated with it.

2.9 Class: network

2.9.1 Fields for class: network

Name	**network**		
Description	*a virtual network*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RW	`name/label`	string	a human-readable name
RW	`name/description`	string	a notes field containg human-readable description
RW	`VIFs`	(VIF ref) Set	list of connected vifs
RW	`NIC`	string	ethernet device to use to access this network. Note: in this revision of the API all hosts will use the specified NIC to access this network
RW	`VLAN`	string	VLAN tag to use to access this network. Note: in this revision of the API all hosts will use the specified VLAN tag to access this network
RW	`default_gateway`	string	default gateway IP address. Used for auto-configuring guests with fixed IP setting
RW	`default_netmask`	string	default netmask. Used for auto-configuring guests with fixed IP setting

2.9.2 Additional RPCs associated with class: network

RPC name: list

Overview: Return a list of all the networks known to the system
Signature:

```
((network ref) Set) list (session_id s)
```

Return Type: `(network ref) Set`
A list of all the IDs of all the networks

2.10 Class: VIF

2.10.1 Fields for class: VIF

Name	**VIF**		
Description	*a virtual network interface*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RW	`name`	string	human-readable name of the interface
RW	`type`	driver_type	interface type
RW	`device`	string	name of network device as exposed to guest e.g. eth0
RW	`network`	network ref	virtual network to which this vif is connected
RW	`VM`	VM ref	virtual machine to which this vif is connected
RW	`MAC`	string	ethernet MAC address of virtual interface, as exposed to guest
RW	`MTU`	int	MTU in octets
RO_{run}	`network_read_kbs`	float	Incoming network bandwidth
RO_{run}	`network_write_kbs`	float	Outgoing network bandwidth
RO_{run}	`IO_bandwidth/incoming_kbs`	float	Read bandwidth (Kb/s)
RO_{run}	`IO_bandwidth/outgoing_kbs`	float	Write bandwidth (Kb/s)

2.10.2 Additional RPCs associated with class: VIF

Class VIF has no additional RPCs associated with it.

2.11 Class: SR

2.11.1 Fields for class: SR

Name	**SR**		
Description	*a storage repository*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RW	`name/label`	string	a human-readable name
RW	`name/description`	string	a notes field containg human-readable description
RW	`VDIs`	(VDI ref) Set	managed virtual disks
RO_{run}	`virtual_allocation`	int	sum of virtual_sizes of all VDIs in this storage repository (in bytes)
RO_{run}	`physical_utilisation`	int	physical space currently utilised on this storage repository (in bytes). Note that for sparse disk formats, physical_utilisation may be less than virtual_allocation
RO_{ins}	`physical_size`	int	total physical size of the repository (in bytes)
RO_{ins}	`type`	string	type of the storage repository
RO_{ins}	`location`	string	a string that uniquely determines the location of the storage repository; the format of this string depends on the repository's type

2.11.2 Additional RPCs associated with class: SR

RPC name: clone

Overview: Take an exact copy of the Storage Repository; the cloned storage repository has the same type as its parent
Signature:

```
(SR ref) clone (session_id s, SR ref sr, string loc, string name)
```

Arguments:

type	name	description
`SR ref`	sr	The Storage Repository to clone
`string`	loc	The location string that defines where the new storage repository will be located
`string`	name	The name of the new storage repository

Return Type: `SR ref`
The ID of the newly created Storage Repository.

RPC name: list

Overview: Return a list of all the Storage Repositories known to the system
Signature:

```
((SR ref) Set) list (session_id s)
```

Return Type: `(SR ref) Set`
A list of all the IDs of all the Storage Repositories

2.12 Class: VDI

2.12.1 Fields for class: VDI

Name	**VDI**		
Description	*a virtual disk image*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RW	`name/label`	string	a human-readable name
RW	`name/description`	string	a notes field containg human-readable description
RW	`SR`	SR ref	storage repository in which the VDI resides
RW	`VBDs`	(VBD ref) Set	list of vbds that refer to this disk
RW	`virtual_size`	int	size of disk as presented to the guest (in multiples of sector_size field)
RO_{run}	`physical_utilisation`	int	amount of physical space that the disk image is currently taking up on the storage repository (in bytes)
RO_{ins}	`sector_size`	int	sector size of VDI (in bytes)
RO_{ins}	`type`	vdi_type	type of the VDI
RO_{ins}	`parent`	VDI ref	parent disk (e.g. in the case of copy on write)
RO_{ins}	`children`	(VDI ref) Set	child disks (e.g. in the case of copy on write)
RW	`sharable`	bool	true if this disk may be shared
RW	`read_only`	bool	true if this disk may ONLY be mounted read-only

2.12.2 Additional RPCs associated with class: VDI

RPC name: snapshot

Overview: Take an exact copy of the VDI; the snapshot lives in the same Storage Repository as its parent.
Signature:

```
(VDI ref) snapshot (session_id s, VDI ref vdi)
```

Arguments:

type	**name**	**description**
`VDI ref`	vdi	The VDI to snapshot

Return Type: `VDI ref`
The ID of the newly created VDI.

2.13 Class: VBD

2.13.1 Fields for class: VBD

Name	**VBD**		
Description	*a virtual block device*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RW	`VM`	VM ref	the virtual machine
RW	`VDI`	VDI ref	the virtual disk
RW	`device`	string	device seen by the guest e.g. hda1
RW	`mode`	vbd_mode	the mode the disk should be mounted with
RW	`driver`	driver_type	the style of driver
RO_{run}	`IO_bandwidth/incoming_kbs`	float	Read bandwidth (Kb/s)
RO_{run}	`IO_bandwidth/outgoing_kbs`	float	Write bandwidth (Kb/s)

2.13.2 Additional RPCs associated with class: VBD

Class VBD has no additional RPCs associated with it.

2.14 Class: user

2.14.1 Fields for class: user

Name	**user**		
Description	*a user of the system*		
Quals	Field	Type	Description
RO_{run}	`uuid`	string	unique identifier/object reference
RO_{ins}	`short_name`	string	short name (e.g. userid)
RW	`fullname`	string	full name

2.14.2 Additional RPCs associated with class: user

Class user has no additional RPCs associated with it.

2.15 DTD

General notes:

- Values of primitive types (int, bool, etc) and higher-order types (Sets, Maps) are encoded as simple strings, rather than being expanded into XML fragments. For example "5", "true", "1, 2, 3, 4", "(1, 2), (2, 3), (3, 4)"
- Values of enumeration types are represented as strings (e.g. "PAE", "3DNow!")
- Object References are represented as UUIDs, written in string form