Designing Backup Solutions for VMware vSphere

VMware vStorage APIs for Data Protection

This document introduces code developers to the concepts and procedures necessary to create backup and restore software for virtual machines and VMware vSphere (formerly VMware Infrastructure, or VI). You should be familiar with programming concepts, practices, and techniques. You should also be familiar with these concepts: virtual machine, snapshot, and vSphere (ESX and vCenter). This document does not duplicate information available elsewhere. When necessary, it refers to other documents.

You should be familiar with navigating the Web-based *VMware vSphere API Reference Documentation*, and read parts of the *VMware vSphere Web Services SDK Programming Guide*, especially the introductory chapters. Both are available at http://www.vmware.com/support/developer/vc-sdk. This background reading covers many of the basic concepts that are important for vSphere deployments. This document is divided into four sections:

If you are looking for a high-level overview of backup and restore, read the first section. If you are designing your top-level program structure, also read the second section. Implementers of low-level code should also read the third section. Each section assumes that you have absorbed knowledge from the preceding section.

Conceptual Overview

This section summarizes the backup process and the restore process.

The Backup Process

Because snapshots are a view of a virtual machine corresponding to a certain point in time, they allow for a quick and clean backup operation. The following steps describe a typical backup workflow:

1   Contact the server machine containing the target virtual machine.

2   Command that server to produce a snapshot of the target virtual machine.

3   Use the server to gain access to the virtual disk(s) and files in the snapshot.

4   Capture the virtual disk data and virtual machine configuration information (`vim.vm.ConfigInfo`).

5   Command the server to destroy the backup snapshot.

A side-effect of step one is being able to determine the arrangement and description of virtual machines on the server. This information is useful for determining the target virtual machine.

The Restore Process

You can choose one of two restore scenarios:

To bring an existing virtual machine to a particular state

1 Contact the server and command it to halt and power off the target virtual machine.

2 Use the server to gain access to the virtual disk(s).

3 Transfer the image(s) of the disk(s) from the backup program.

To completely re-create a virtual machine

1 Contact the server.

2 Command the server to create a new virtual machine and its virtual disks using the configuration information in step 4 of the backup process above.

3 Transfer the virtual disk data to the newly created virtual disk(s). This includes the virtual disk formatting information, so it is unnecessary to build any kind of file system on the virtual disk(s).

Summary of Requirements

A backup program must be able to:

■ Contact a VMware host, display information about the virtual machines that it controls, and manipulate those virtual machines.

■ For backup, instruct each virtual machine to create a temporary snapshot, and transfer snapshot data to the backup application.

■ For restore, instruct the host to halt or re-create a target virtual machine, and restore the data for that virtual machine from the backup application.

High Level Implementation

This section introduces, at a high level, the VMware software abstractions needed to accomplish the steps outlined in the previous section. These abstractions fall into the following categories:

■ Contacting the server

■ Extracting information from the server

■ Causing the server to perform actions

■ Giving instance-specific commands and control

■ Transferring data

Except for the last, data transfer, all the above categories are supported by the VMware vSphere SDK, which contains a wide variety of management and control interfaces.

Communicating With the Server

In a typical VMware vSphere deployment that consists of multiple ESX hosts, an instance of the vCenter typically manages the ESX hosts. It is therefore recommended that applications communicate with the vCenter rather than with the individual ESX hosts.

VMware vCenter provides location transparency to the user of the vSphere SDK. VMware vCenter tracks virtual machines as they move (through VMotion) from one ESX host to another, and it directs vSphere SDK operations to the appropriate ESX host that currently hosts the virtual machine.

The handling of a vCenter or an individual ESX host is essentially equivalent when using the vSphere SDK. Thus when vCenter is present, there is no need to directly contact individual ESX hosts. The remainder of this document uses the term vSphere to indicate either a vCenter or an ESX host.

Using vCenter or the vSphere API, it is even possible to identify any VirtualApp (vApp) that is installed and what virtual machines are associated with the it. This makes it possible to back up the vApp as a unit.

To reduce the resources used by the vSphere, it is generally recommended that the number of connections (or Sessions) to it be minimized. As a result, it is in the best interests of any program that communicates with the vSphere to create one Session and share it with all elements of the program that need to exchange information with the server. This means that if your program supports multiple threads, then your program needs to multiplex the use of the connection object through the use of access control locks (mutex and the like).

It is also important to note that all vSphere SDK operations proceed from an instance of a "Session" object that your application requests after logging into the vSphere. Your application might also create objects through the vSphere SDK that are "Session specific" and therefore would not be known to portions of your application that are using a different Session.

Information Containers as Managed Objects

VMware documentation introduces you to the concept of the Managed Object and its handle (called a `MoRef` for Managed Object Reference). You might be tempted to get configuration and status information of Managed Objects using the piecemeal approach. This has the severe disadvantage of creating a lot of chatter over the connection to the server, and it is very slow. A mechanism has been created to provide this information efficiently. This mechanism is the `PropertyCollector`, discussed in "PropertyCollector Use" on page 4.

More About Managed Objects

When you read the documentation on the VMware vSphere Object Model, you are introduced to a large number of Managed Objects. There are only five basic types of Managed Objects that describe the organization of a server. Other Managed Objects are details expanding on these five basic types. These five basic types are:

- Folder

- Compute Resource

- Resource Pool

- Data Center

- Virtual Machine

It is a characteristic of all Managed Objects that they have a `MoRef` to the Managed Object that serves as the parent to the Managed Object. This "parent" `MoRef` allows you to reconstruct the object hierarchy exposed by the vSphere SDK. In general the hierarchy is a tree-like structure along the lines of:

Root Folder > Data Center > Compute Resource > Resource Pool > Virtual Machine

There are variations on this theme, depending on whether you connect to vCenter or directly to an ESX host, but the overall organization is like the structure above. Each Managed Object also has a `Name` property.

Managed Object References

A Managed Object Reference (`moRef`) is actually a handle and not the Managed Object itself. While it is absolutely certain that a `MoRef` always contains a unique value, the unique value is only relative to the instance of the vSphere to which you are connected. For example, if a vCenter manages a cluster of ESX hosts, each ESX host maintains its own Managed Object Reference namespace and the vCenter must maintain a Managed Object Reference namespace representing all of its servers. So, when an ESX host is represented by a vCenter, the vCenter must ensure that the ESX Managed Object References are unique. The vCenter accomplishes this by creating unique Managed Object Reference names inside its own namespace, which differ from the names that the ESX uses for the same Managed Objects.

A vSphere instance (vCenter or ESX) attempts to keep the Managed Object Reference for a virtual machine consistent across sessions, however consistency is not always guaranteed. For example, unregistering and reregistering a virtual machine could result in a change to the Managed Object Reference for the virtual machine. Hence, it is not a good idea to store a `MoRef` value and expect it to work correctly in a future session, or against a different vSphere instance.

Unique ID for a Virtual Machine

On one vCenter Server, the Managed Object Reference (`moRef`) uniquely identifies a virtual machine. If you need to inventory and track virtual machine backups across multiple vCenter Servers, you can use the `instanceUuid` together with `moRef`. You can see the `instanceUuid` at the following browser path:

`https://<vcserver>/mob/?moid=ServiceInstance&doPath=content.about`

For direct connections to ESX/ESXi, the host address and `moRef` uniquely identify a virtual machine. However this `moRef` could be different from the one that vCenter Server returns, hence the fallback to `instanceUuid`. `InstanceUuid` is new in VMware vSphere 4.0. In previous releases, the fallback is instead `Uuid`.

Gathering Status and Configuration Information

The PropertyCollector is the most efficient mechanism to specify, at the top level, all of the Managed Objects that are of interest to your application. It has methods for providing updates that indicate only changes to the previous state of these objects. There are two mechanisms for acquiring these updates:

- Polling – Checks for changes. The result is either "no change" or an object containing the changes. The advantage of this mechanism is that except for the poll request, it involves no chatter except for reporting.

- Wait for updates – "Wait for updates" is basically a blocking call to the `PropertyCollector`. This is only useful if you dedicate a program thread waiting for the call to unblock. The single advantage of this call is that there is no chatter at all on the communications thread unless something must be reported.

The *VMware vSphere Web Services SDK Programming Guide* contains a lot of information about the `PropertyCollector`. This is a very powerful interface, requiring great attention to detail. Backup-related features of the `PropertyCollector` are covered in "Low Level Procedures" on page 8 of this document. The next section highlights some characteristics of the `PropertyCollector` as an overview.

PropertyCollector Use

This document assumes that you want to keep up with changes in the configuration of the VMware server, and therefore plan to use the update tracking capability of the `PropertyCollector`. The `PropertyCollector` requires two fairly complex arguments: the `PropertySpec` and the `ObjectSpec`.

The `ObjectSpec` contains instructions to the `PropertyCollector` describing where to look for the desired data. Because configuration information in the VMware server is organized like a directory tree, the `ObjectSpec` must describe how to traverse this tree to obtain the desired information. The net result is a complex, nested, and recursive list of instructions. Fortunately, once you have determined the location of all the desired information is, the `ObjectSpec` needed to determine the layout of a vSphere object hierarchy can be a static unvarying object. See the code example in section "Understanding an ObjectSpec" on page 9.

The `PropertySpec` is a list of desired property information. Formulating a list that includes all of the desired information can take some effort to compile, but once determined, this can be a static object also.

The data returned from the `PropertyCollector` is a container class called `PropertyFilterUpdate`. This class is an item-by-item list of changes to the list of requested properties. Every item in this container is identified with one of the following keys: enter (add), leave (delete), and modify. On the first data request, every data item is included, and every data item is marked "enter".

The `PropertyCollector` presents its results in what amounts to random order. Since all Managed Objects have a "`parent`" property, you can reconstruct the configuration hierarchy by building a tree in memory, using the "`parent`" identification to organize. The root folder is identified as the only folder without a parent.

Useful Information

You can find most of the information that is useful to a backup program in the Virtual Machine Managed Object. This Managed Object contains a lot of information, but the following is probably most useful:

- Virtual Disks – Names, Types, and Capacities.

- Machine Type and Configuration – Whatever would be useful in (re)creating a virtual machine. This list might include such information as number of CPUs and memory size.

- Display Names – These names appear in VMware products such as the VMware vSphere Client. You should keep track of these names and correlate them for consistency between your product and what VMware displays in its products.

Something to remember is that VMware supports a number of virtual disk implementations. The type of disk implementation is important for two reasons:

- A disk backed by a physical compatibility RDM mostly bypasses the ESX storage stack. As a result, you cannot make a snapshot of this type of virtual disk. Therefore, this type of virtual disk cannot be backed up using the snapshot method described in this document.

- On restore, you should re-create virtual disk with the same disk type as the original virtual machine used.

Doing a Backup Operation

After you obtain the information about what is available to back up, you can perform the backup. The three steps to the backup process are:

- "Creating a Temporary Snapshot on the Target Virtual Machine" on page 5

- "Extracting the Backup Data from the Target Virtual Machine" on page 5, save configuration information

- "Deleting the Temporary Snapshot" on page 6

Creating a Temporary Snapshot on the Target Virtual Machine

The low-level procedure for creating a snapshot of a virtual machine is documented in the section "Creating a Snapshot" on page 13. The important thing to note is the flag for making the file system quiescent. If the file system is not quiescent, you could end up with a snapshot of a system in a transitional state. The disk entries may not be in a consistent state. Attempting to restore with this data could prove to be destructive.

There is also a second flag that allows you to include a dump of a powered on virtual machine's in-memory state in the snapshot. This is not needed for backup, so you should this flag to false.

As a best practice, you should search for, and delete, any existing snapshots with the same name that you selected for the temporary snapshot. These snapshots are possibly remnants from failed attempts to back up the virtual machine.

Extracting the Backup Data from the Target Virtual Machine

Associated with the snapshot you just created are "versions" of the virtual disk(s). In order to identify these disks, you obtain a `MoRef` to the snapshot just created. From this snapshot `MoRef`, you can extract the disk names and paths. How to do this is demonstrated in section "Backing Up a Virtual Disk" on page 13.

Within a specific snapshot, the name s of virtual disk files (with extension `.vmdk`) can be modified with a zero-filled 6-digit decimal sequence number to ensure that the `.vmdk` files are uniquely named. Depending on whether or not the current virtual machine had a pre-existing snapshot, the disk name for a snapshot could have this format: `<diskname>–<NNNNNN>.vmdk`. This unique name is no longer valid after the snapshot is destroyed, so any data for a snapshot disk should be stored in the backup program under its base disk name.

To access the data in a virtual disk, it is necessary to use the `VixDiskLib`. This library isolates the programmer from the (extremely) gory details of extracting data from a virtual disk and its "redo" logs. `VixDiskLib` allows access to disk data on sector boundaries only, and the transfer size is some multiple of the disk sector size.

When accessing disks on ESX hosts, `VixDiskLib` release 1.0 transferred virtual disk data over the network. `VixDiskLib` release 1.1 contains API enhancements so you can request more efficient data paths, such as direct SAN access or hot-adding disks to a virtual backup appliance. These efficient data paths require only small changes to your application, for example calling `VixDiskLib_ConnectEx()` instead of plain connect.

Part of the information in a virtual disk is a number of key/value pairs that describe the configuration of the virtual disk. This information can also be extracted from the virtual disk using the `VixDiskLib` function `VixDiskLib_GetMetadataKeys()`. You should save this metadata information in order to be able to re-create the virtual disk should it become necessary.

The `VixDiskLib` API allows a backup application to perform a full backup of a virtual machine. The newer `VixMntapi` library can extract information about a guest operating system from its virtual disks, so your backup application can determine what type of operating system is installed there. This allows mounting the volumes to device nodes, so that applications can perform file-oriented backups of a virtual machine.

Deleting the Temporary Snapshot

As the last part of the backup process, you should delete the temporary snapshot. It is no longer needed, and it represents storage space that could be put to other uses. The procedure for reverting a snapshot is outlined in the *VMware vSphere Web Services SDK Programming Guide*.

Doing a Restore Operation

The restore process has two modes of operation:

-
-

Restoring an Existing Virtual Machine to a Known State

The following steps restore a virtual machine to a known state:

1   Shut down the virtual machine (if it is not already shut down).

    For obvious security reasons, you are not granted write access to the disks of a running virtual machine. Before you shut it down, you need to determine the run-state of the virtual machine. This is information that is available from the `PropertyCollector`, and if you have been keeping this information up-to-date, then your application already knows the run-state of the virtual machine.

    To change the run-state of a virtual machine requires that you determine the `MoRef` of the virtual machine. You can then use this `MoRef` in a `PowerOnVM_Task` call through the connection you make to the server. The procedure for accomplishing virtual machine shutdown is outlined in the *VMware vSphere Web Services SDK Programming Guide* in the "Performing Power Operations on a Virtual Machine" section.

2   Restore the contents of the virtual disks.

    The process of restoring the disk data requires that you obtain the current names of the virtual disks. This process is similar to the one described in "Extracting the Backup Data from the Target Virtual Machine" on page 5, except in this case you obtain this information directly from the virtual machine and not from a snapshot. The target for the saved disk data must be the actual disk name (including any sequence number that may exist), because the current incarnation of the virtual machine may be derived from one or more snapshots.

    Restoring disk data requires the use of the `VixDiskLib` interface. Documentation of this interface is at http://www.vmware.com/support/developer/vddk/. The `VixDiskLib_Write()` function allows you to open the virtual machine's virtual disks and write your restore data. `VixDiskLib` functions transfer data to even sector boundaries only, and the transfer length must be an even multiple of a sector size.

    Since the virtual disk exists, it is not necessary to restore the disk configuration information mentioned in "Extracting the Backup Data from the Target Virtual Machine" on page 5.

Creating a New Virtual Machine

The process of building a virtual machine from backup data involves the following steps:

1   Create the virtual machine.

    The process for creating a virtual machine is described in the *VMware vSphere Web Services SDK Programming Guide* in the "Creating a Virtual Machine" section. To create a new virtual machine, you must use the information about virtual machine configuration that you derived and saved during the backup process. You should allow users the opportunity to rename the virtual machine during this process in case they want to "clone" the virtual machine. You also might consider offering users the opportunity to change the virtual machine's layout (for instance, storing the virtual disks on different data stores).

Creating the virtual disks for this machine is also done at the time you create the virtual machine. This process is fairly complicated. See the section "Low Level Procedures" on page 8 for details.

2   Restore the virtual disk data.

This process is similar to restoring the contents of virtual disks with the following exception: you must use a `VixDiskLib` function to set the disk configuration key/value data into the virtual disk before writing any backed-up data to the disk. Then use `VixDiskLib` to restore data to the disk, as described above.

3   Power on the virtual machine.

Changed Block Tracking

This relatively new feature provides the foundation for incremental or differential backup of virtual disks. Your application can back up only changed data as indicated by the `QueryChangedDiskAreas` method. Recently created virtual machines can support this capability, in which case the virtual machine contains `changeTrackingSupported` in the `capability` field of the `VirtualMachine` Managed Object. See "Changed Block Tracking on Virtual Disks" on page 21 for more details.

Accessing Files on Virtual Disks

It might be necessary for a backup application to access files on the virtual disks. You can find the interfaces to accomplish this in the `VixMntapi` library, associated with `VixDiskLib`. The `VixMntapi` API allows the disks or volumes contained within a virtual machine to be mounted and examined in any way necessary.

The procedure for mounting a virtual disk requires the following sequence of steps:

1   Locate the path names of all the virtual disks associated with a snapshot.

2   Call `VixDiskLib_Open()` to open all of these virtual disks. This gives you a number of `VixDiskLib` handles, which you should store in an array.

3   Call `VixMntapi_OpenDiskSet()` to create a `VixDiskSetHandle`, passing in the array of `VixDiskLib` handles that you created in step 2.

4   Pass `VixDiskSetHandle` as a parameter to `VixMntapi_GetVolumeHandles()` to obtain an array of `VixVolumeHandle` pointers to all volumes in the disk set.

5   Call `VixMntapi_GetOsInfo()` to determine what kind of operating system is involved, and where important pieces of information are to be found.

6   For important volumes, call `VixMntapi_MountVolume()` then `VixMntapi_GetVolumeInfo()`, which reveals how the volume is set up.

7   If you need information about how the guest operating system sees the data on this volume, you can look in the data structure `VixVolumeInfo` returned by `VixMntapi_GetVolumeInfo()`.

`VixVolumeInfo::symbolicLink`, obtained using `VixMntapi_GetVolumeInfo()`, is the path on the proxy where you can access the file system in the virtual disk using ordinary open, read, and write calls.

Once you are done with accessing files in a volume, there are `VixMntapi` procedures for taking down the abstraction that you have created. These calls are:

-   `VixMntapi_DismountVolume()`

-   `VixMntapi_FreeOsInfo()`

-   `VixMntapi_FreeVolumeInfo()`

-   `VixMntapi_CloseDiskSet()`

This leaves the `VixDiskLib` handles that you obtained in the beginning; you must dispose of those properly.

Summary

In the preceding sections you have seen at a high level:

- How to contact a server

- How to extract information from that server

- How to manipulate items on the server

- How to backup and restore data from virtual disks

- How to access file data on virtual disks

The following sections cover the same information at a lower level.

Low Level Procedures

This section describes the low level details that should be helpful in actually coding your backup application. It is not the intent of this document to impose a design. Instead, this document should serve as a guideline, providing helpful examples and exposition. The code samples in this section are definitely not complete. They lack appropriate error handling, and they ignore other details that may be necessary to make a program work. The code samples are mainly meant to demonstrate approaches to accomplish the backup task.

Communicating with the Server

Connections to the server machine require credentials: user name, password, and hostname (or IP address). The following code shows how to contact the server and extract the information that is most useful for manipulating it. This example assumes that you have already set up the vSphere SDK as described in the *VMware vSphere Web Services SDK Programming Guide* sections about installation and configuration of the SDK server.

1   Create the service instance:

```
ManagedObjectReference svcRef = new ManagedObjectReference();
svcRef.setType("ServiceInstance");
svcRef.setValue("ServiceInstance");
```

2   Locate the service:

```
VimServiceLocator locator = new VimServiceLocator();
locator.setMaintainSession(true);
VimPortType serviceConnection = locator.getVimPort("https://your_server/sdk");
ServiceInstanceContent serviceContent = serviceConnection.retrieveContent(svcRef);
ManagedObjectReference sessionManager = serviceInstance.getSessionManager();
UserSession us = serviceConnection.login(sessionManager, username, password, null);
```

The PropertyCollector

The PropertyCollector is described in some detail in the *VMware vSphere Web Services SDK Programming Guide*. This section applies those details to the backup task.

PropertyCollector Arguments

The PropertyCollector uses two relatively complicated argument structures. As was mentioned in "PropertyCollector Use" on page 4, these arguments are PropertySpec and ObjectSpec. The PropertySpec is a list of the information desired, and the ObjectSpec is a list of instructions indicating where the desired information can be found. In theory, you can almost directly address an object using its MoRef. In that case an ObjectSpec could be very simple. However, getting the MoRef in the first place can be a challenge when a complicated ObjectSpec is required. To formulate this complicated ObjectSpec, you need to understand the structure of the available data, which can be a confusing exercise. This is complicated by the fact that an ObjectSpec can contain recursive elements.

Understanding an ObjectSpec

An `ObjectSpec` is a list of `ObjectSpec` elements. Each element describes the type of object, and gives a "selection spec". First consider the type of object. "More About Managed Objects" on page 3 describes five types of Managed Objects. Here is how you "traverse" objects (how one Managed Object leads to another).

- Folder – If you read the *VMware vSphere SDK Reference Guide* section about a Folder Managed Object, you see that one of the items contained in the Folder is called `childEntity`, which is a list of `MoRefs` that can contain any of the following Managed Object types: Folder, Data Center, Compute Resource, or Virtual Machine. This means that a Folder can be a parent to any of the Managed Objects in this list.

- Data Center – Data Center has two items that lead to other Managed Objects. These are:

    - `hostFolder` – A `MoRef` to a Folder containing a list of Compute Resources comprising a Data Center.

    - `vmFolder` – A `MoRef` to a Folder containing the Virtual Machines that are part of the Data Center. If it is your objective to duplicate the display seen in a Virtual Client GUI, then this Folder is of limited use because it does not describe the Resource Pool that is the parent of the virtual machine.

- Compute Resource – A Compute Resource is basically hardware. A Compute Resource may be composed of multiple host systems. This hardware represents resources that you can use to implement a Virtual Machine object. However, a Virtual Machines is always a child of a Resource Pool, which is used to control the sharing of the real machine's resources among the Virtual Machine objects. A Compute Resource contains an item named `resourcePool`, which is a `MoRef` to a Resource Pool.

- VirtualApp – A VirtualApp (vApp) is a collection of Virtual Machines that make up a single application. This is a special form of Resource Pool (defined below). A VirtualApp may have three types of children:

    - Virtual Machine – A folder named `vm` contains a list of `MoRefs` to child Virtual Machines.

    - `resourcePool` – A folder containing a list of `MoRefs` pointing to child Resource Pools or VirtualApps.

        - VirtualApp – A VirtualApp can be composed of other VirtualApps.

        - Resource Pool – You can segment the resources of a VirtualApp using a Resource Pool.

- Resource Pool – Resource Pool contains two child items:

    - `resourcePool` – A folder containing a list of `MoRefs` pointing to child Resource Pools or VirtualApps.

    - `vm` – A list of `MoRefs` to child Virtual Machines that employ the resources of the parent Resource Pool. A Virtual Machine always lists a Resource Pool as its parent.

- Virtual Machine – Virtual Machine can be considered to be an "end object" and as such you need not describe any traversal for this object.

You must understand that the `ObjectSpec` does not have to lead you any farther than the `MoRef` of a target object. You can gather information about the Managed Object itself using the `MoRef` and the `PropertySpec`. This is described in detail in the section "Understanding a PropertySpec" on page 10.

In the document *VMware vSphere Web Services SDK Programming Guide*, an `ObjectSpec` element (called a `TraversalSpec`) is described as containing the following elements:

- `Type` – The type of object being referenced.

- `Path` – The element contained in the object that is used to steer traversal.

- `Name` – Optional reference name that you can use to reference this `TraversalSpec` in another `SelectSet`.

- `SelectSet` – An array containing either `SelectionSpec` or `TraversalSpec` elements.

`SelectionSpec` is a direct target for traversal, as is `TraversalSpec` (a class extending `SelectionSpec`). It is in the `SelectSet` that recursion can occur.

If you wish to traverse the entire configuration tree for a server, then you need only the "root node" `MoRef`, which is always a Folder. This root folder name is available by using the function `getRootFolder` from the service instance that connects to the vSphere. All of the above goes into this Java code sample:

```
// Remember, TraversalSpec objects can use a symbolic name.
```

```
// In this case we use the symbolic name "folderTSpec".
// First we must define the TraversalSpec objects used to fill in the ObjectSpec.
//
// This TraversalSpec traverses Datacenter to vmFolder
TraversalSpec dc2vmFolder = new TraversalSpec();
dc2vmFolder.setType("Datacenter");  // Type of object for this spec
dc2vmFolder.setPath("vmFolder");  // Property name defining the next object
dc2vmFolder.setSelectSet(new SelectionSpec[] {"folderTSpec"});
//
// This TraversalSpec traverses Datacenter to hostFolder
TraversalSpec dc2hostFolder = new TraversalSpec();
dc2hostFolder.setType("Datacenter");
dc2hostFolder.setPath("hostFolder");
dc2vmFolder.setSelectSet(new SelectionSpec[] {"folderTSpec"});
//
// This TraversalSpec traverses ComputeResource to resourcePool
TraversalSpec cr2resourcePool = new TraversalSpec();
cr2resourcePool.setType("ComputeResource");
cr2resourcePool.setPath("resourcePool");
//
// This TraversalSpec traverses ComputeResource to host
TraversalSpec cr2host = new TraversalSpec();
cr2host.setType("ComputeResource");
cr2host.setPath("host");
//
// This TraversalSpec traverses ResourcePool to resourcePool
TraversalSpec rp2rp = new TraversalSpec();
rp2rp.setType("ResourcePool");
rp2rp.setPath("resourcePool");
//
// Finally, we tie it all together with the Folder TraversalSpec
TraversalSpec folderTS = new TraversalSpec();
folderTS.setName{"folderTSpec");  // Used for symbolic reference
folderTS.setType("Folder");
folderTS.setPath("childEntity");
folderTS.setSelectSet(new SelectionSpec[]{ "folderTSpec",
                                           dc2vmFolder,
                                           dc2hostFolder,
                                           cr2resourcePool,
                                           rp2rp});
ObjectSpec ospec = new ObjectSpec();
ospec.setObj(startingPoint); // This is where you supply the starting MoRef (usually root folder)
ospec.setSkip(Boolean.FALSE);
ospec.setSelectSet(folderTS); // Attach the TraversalSpec we designed above
```

Understanding a PropertySpec

A `PropertySpec` is a list of individual properties that can be found at places identified by the `ObjectSpec`. Once the `PropertyCollector` has a `MoRef`, it can then return the properties associated with that `MoRef`. This can include "nested" properties. Nested properties are properties that can be found inside of properties identified at the top level of the Managed Object. Nested properties are identified by a "dot" notation.

An example of nested properties can be drawn from the Virtual Machine Managed Object. If you examine the *VMware vSphere SDK Reference Guide* you see that a Virtual Machine has the property identified as `summary`, which identifies a `VirtualMachineSummary` Managed Object. In turn, the `VirtualMachineSummary` contains property `config`, which identifies a `VirtualMachineConfigSummary`. `VirtualMachineConfigSummary` has a property called `name`, which is a string containing the display name of the Virtual Machine. You can access this final property using string value `summary.config.name`. For all the `VirtualMachineConfigSummary` information, string value `summary.config` causes return of all `VirtualMachineConfigSummary` properties.

The `PropertyCollector` requires an array of `PropertySpec` elements. Each element includes:

■ `Type` – The type of object that identifies the enclosed list of properties.

■ `PathSet` – An array of strings containing names of properties to be returned, including nested properties.

It is necessary to add an element for each type of object that you wish to query for properties. The following is a code sample of a `PropertySpec`:

```
// This code demonstrates how to specify a PropertySpec for several types of target objects:
PropertySpec folderSp = new PropertySpec();
folderSp.setType("Folder");
folderSp.setAll(Boolean.FALSE);
folderSp.setPathSet(new String [] {"parent", "name"});
PropertySpec dcSp = new PropertySpec();
dcSp.setType("Datacenter");
dcSp.setAll(Boolean.FALSE);
dcSp.setPathSet(new String [] {"parent","name"});
PropertySpec rpSp = new PropertySpec();
rpSp.setType("ResourcePool");
rpSp.setAll(Boolean.FALSE);
rpSp.setPathSet(new String [] {"parent","name","vm"});
PropertySpec crSp = new PropertySpec();
crSp.setType("ComputeResource");
crSp.setAll(Boolean.FALSE);
crSp.set:PathSet(new String [] {"parent","name"});
PropertySpec vmSp = new PropertySpec();
vmSp.setType("VirtualMachine");
vmSp.setAll(Boolean.FALSE);
vmSp.setPathSet(new String [] {"parent",
                              "name",
                              "summary.config",
                              "snapshot",
                              "config.hardware.device"});
// Tie it all together
PropertySpec [] pspec = new PropertySpec [] {folderSp,
                                             dcSp,
                                             rpSp,
                                             crSp,
                                             vmSp};
```

Getting the Data from the PropertyCollector

Now that we have defined `ObjectSpec` and `PropertySpec` (the where and what), we need to put them into a `FilterSpec` that combines the two. An array of `FilterSpec` elements is passed to the `PropertyCollector` (the minimum number of elements is one). Two mechanisms can retrieve data from `PropertyCollector`:

- `RetrieveProperties` – A one-time request for all of the desired properties.

- Updates – `PropertyCollector` update requests take two forms: polling and waiting, discussed below.

Checking for Updates

The `RetrieveProperties` operation is rather obvious, so consider the update method. In either Polling or Waiting, it is first necessary to register your `FilterSpec` array object with the `PropertyCollector`. You accomplish this using the `CreateFilter` function, which sends a copy of your `FilterSpec` to the server. Unlike the `RetrieveProperties` function, `FilterSpec` is not discarded after the `CreateFilter` operation. The following code shows how to set your `FilterSpec`:

```
// We already showed examples of creating pspec and ospec in the examples above.
// Remember, the PropertyCollector wants an array of FilterSpec objects, so:
PropertyFilterSpec fs = new PropertyFilterSpec();
fs.setPropSet(pspec);
fs.setObjectSet(ospec);
PropertyFilterSpec [] fsa = new PropertyFilterSpec [] {fs};
ManagedObjectReference pcRef = serviceContent.getPropertyCollector();
// This next statement sends the filter to the server for reference by the PropertyCollector
ManagedObjectReference pFilter = serviceConnection.CreateFilter(pcRef, fsa, Boolean.FALSE);
```

If you wish to begin polling, you may then call the function `CheckForUpdates`, which on the first try (when it must contain an empty string for the version number) returns a complete dump of all the requested properties, along with a version number. Subsequent calls to `CheckForUpdates` must contain this version number to indicate to the PropertyCollector that you seek any changes that deviate from this version. The result is either a partial list containing only the changes from the previous version (including a new version number), or a return code indicating no data has changed. The following code sample shows how to check for updates:

```
String updateVersion = "";  // Start with no version
UpdateSet changeData = serviceConnection.CheckForUpdates(pcRef, version);
if (changeData != nil) {
    version = changeData.getVersion();  // Extract the version of the data set
}
...
// Get changes since the last version was sent.
UpdateSet latestData = serviceConnection.CheckForUpdates(pcRef, version);
```

If instead you wish to wait for updates to occur, you must create a task thread that blocks on the call `WaitForUpdates`. This task returns changes only as they occur and not at any other time.

Extracting Information from the Change Data

The data returned from `CheckForUpdates` (or `WaitForUpdates`) is an array of `PropertyFilterUpdate` entries. Since a `PropertyFilterUpdate` entry is very generic, here is some code showing how to extract information from the `PropertyFilterUpdate`.

```
// Extract the PropertyFilterUpdate set from the changeData
//
PropertyFilterUpdate [] updateSet = changeData.getFilterSet();
//
// There is one entry in the updateSet for each filter you registered with the PropertyCollector.
// Since we currently have only one filter, the array length should be one.
//
PropertyFilterUpdate myUpdate = updateSet[0];
ObjectUpdate [] changes = myUpdate.getObjectSet();
for (a = 0; a < changes.length; ++a) {
    ObjectUpdate theObject = changes[a];
    String objName = theObject.getObj().getMoType().getName();
    // Must decide how to handle the value based on the name returned.
    // The only names returned are names found in the PropertySpec lists.
    ...
}
```

Getting Specific Data

From time to time, you might need to get data that is relevant to a single item. In that case you can create a simple `ObjectSpec` naming the `MoRef` for the item of interest. The `PropertySpec` can then be set to obtain the properties you want, and you can use `RetrieveProperties` to get the data. You can obtain `MoRef` from your general examination of the properties outlined above (where you search for information from the root Folder).

Identifying Virtual Disks for Backup and Restore

When attempting to back up a virtual machine, you first need to create a snapshot. Once the snapshot is created, you then need to find and identify the virtual disks associated with this snapshot. If you are attempting to restore the virtual disks in the virtual machine, then you need to find and identify the virtual disks that are currently active. To understand why this might be a problem, it is first necessary to understand that a virtual machine may have multiple snapshots associated with it. Each snapshot has a virtual "copy" of the virtual disks for the virtual machine. These "copies" are named with the base name of the disk, and a unique decimal number appended to the name. The format of the number is a hyphen character followed by a 6-digit zero-filled number. An example disk "copy" name would be: `mydisk–NNNNNN.vmdk` where NNNNNN would be some number like: `000032`.

In addition to the name of the disk, there should be a path to this disk on the server, which is stored as a datastore path: [storagex] myvmname/mydisk–NNNNNN.vmdk. The path component in the square brackets corresponds to the name of the datastore that contains this virtual disk while the remainder of the path string represents the location relative to the root of this data store. Usually you do not have to be concerned with this, since the vSphere SDK takes care of this. So how do you obtain this path? Whether we are talking about a snapshot or obtaining the information from the base virtual machine, the process is the same. The only difference is in the Managed Object that we use to extract the information.

Whether you use the PropertyCollector to get a Virtual Machine or a Virtual Machine Snapshot, you need to select the property: config.hardware.device. This returns an array of Virtual Devices associated with the Virtual Machine or Snapshot. You must scan this list of devices to extract the Virtual Disks. All that is necessary is to see if each VirtualDevice entry extends to VirtualDisk. When you find such an entry, examine the BackingInfo property. You must extend the type of the backing property to one of the following:

- VirtualDiskFlatVer1BackingInfo

- VirtualDiskFlatVer2BackingInfo

- VirtualDiskRawDiskMappingVer1BackingInfo

- VirtualDiskSparseVer1BackingInfo

- VirtualDiskSparseVer2BackingInfo

It is important to know which backing type is in use in order to be able to re-create the Virtual Disk.It is also important to know that you cannot snapshot a disk of type VirtualDiskRawDiskMappingVer1BackingInfo, and therefore you cannot back up this type of Virtual Disk.

The properties of interest are the backing fileName and the VirtualDisk capacityInKB. Additionally, when change tracking is in place, you should also save the changeID.

Creating a Snapshot

In order to perform a backup operation, you must first create a snapshot of the target virtual machine. The following code sample shows how to create a snapshot on a specific virtual machine:

```
// At this point we assume the virtual machine is identified as ManagedObjectReference vmMoRef.
String SnapshotName = "Backup";
String SnapshotDescription = "Temporary Snapshot for Backup";
boolean memory_files = false;
boolean quiesce_filesystem = true;
ManagedObjectReference taskRef = serviceConnection.CreateSnapshot_Task(vmMoRef,
     SnapshotName, SnapshotDescription, memory_files, quiesce_filesystem);
```

Backing Up a Virtual Disk

This section describes how to get data from the Virtual Disk after you have identified it. In order to access a virtual disk, you must use the VixDiskLib. The following code shows how to initialize the VixDiskLib and use it for accessing a virtual disk. All operations require a VixDiskLib connection to access virtual disk data. At the present time VixDiskLib is implemented only for the C language, so this is not Java code:

```
VixDiskLibConnectParams connectParams;
VixDiskLibConnection srcConnection;
connectParams.serverName = strdup("TargetServer");
connectParams.creds.uid.userName = strdup("root");
connectParams.creds.uid.password = strdup("yourPasswd");
connectParams.port = 902;
VixError vixError = VixDiskLib_Init(1, 0, &logFunc, &warnFunc, &panicFunc, libDir);
vixError = VixDiskLib_Connect(&connectParams, &srcConnection);
```

This next section of code shows how to open and read a specific virtual disk:

```
VixDiskLibHandle diskHandle;
vixError = VixDiskLib_Open(srcConnection, diskPath, flags, &diskHandle);
uint8 mybuffer[some_multiple_of_512];
vixError = VixDiskLib_Read(diskHandle, startSector, numSectors, &mybuffer);
```

```
// Also getting the disk metadata:
size_t requiredLength = 1;
char *buf = new char [1];
// This next operation fails, but updates "requiredLength" with the proper buffer size
vixError = VixDiskLib_GetMetadataKeys(diskHandle, buf, requiredLength, &requiredLength);
delete [] buf;
buf = new char[requiredLength];  // Create a large enough buffer
vixError = VixDiskLib_GetMetadataKeys(diskHandle, buf, requiredLength, NULL);
```

And finally, close the `diskHandle`:

```
vixError = VixDiskLib_Close(diskHandle);
// And if you are completely done with the VixDiskLib
VixDiskLib_Disconnect(srcConnection);
VixDiskLib_Exit();
```

Deleting a Snapshot

When you are done performing a backup, you need to delete the temporary snapshot. You should already have the `MoRef` for the snapshot from the operation that created the snapshot in the first place. The following Java code demonstrates how to delete the snapshot:

```
ManagedObjectReference removeSnapshotTask;
SnapshotManagedObject snapshot;
removeSnapshotTask = serviceConnection.removeSnapshot_Task(snapshot, Boolean FALSE);
```

Restoring a Virtual Disk

This section describes how to restore the data into the Virtual Disk.

No matter how you attempt to do it, you cannot get write access to a Virtual Disk that is in active use. You first must ensure that the disk is not in use by halting the parent Virtual Machine, then performing the "power off" sequence. The following code sample demonstrates how to "power off" a Virtual Machine:

```
// At this point we assume that you have a ManagedObjectReference to the VM – vmMoRef.
// You also need a ManagedObjectReference to the host running the VM – hostMoRef.
ManagedObjectReference taskRef = serviceConnection.powerOffVm(vmMoRef, hostMoRef);
```

This is the phase where it is necessary to use VixDiskLib to reload the contents of the Virtual Disk, so the following code is C, not Java:

```
// At this point we assume that you already have a VixDiskLib connection to the server machine.
uint8 mybuffer[some_multiple_of_512];
int mylocalfile = open("localfile", openflags);
read(mylocalfile, mybuffer, sizeof mybuffer);
vixError = VixDiskLib_Open(srcConnection, path, flags, &diskHandle);
VixDiskLib_Write(diskHandle, startsector, (sizeof mybuffer) / 512, mybuffer);
```

Creating a Virtual Machine

This section shows how to create a Virtual Machine object. The process of creating a Virtual Machine is fairly complicated. In order to accomplish this it is necessary to create a `VirtualMachineConfigSpec` describing the Virtual Machine and all of its supporting virtual devices. Almost all of the required information is available from the Virtual Machine property `config.hardware.device`, which is a table containing all of the device configuration information. The relationships between the devices are described by the value `key`, which is a unique identifier for the device. In turn, each device has a `controllerKey`, which is the key identifier of the controller where the device is connected. It is a necessary practice to specify the controller/key relationships using negative key numbers. This guarantees that the key number does not conflict with the real key number when it is assigned by the server. Below is an actual `VirtualMachineConfigSpec` that was used to create a Virtual Machine (`vim.vm.ConfigSpec`).

```
// beginning of VirtualMachineConfigSpec
// ends several pages later
{
   dynamicType = <unset>,
   changeVersion = <unset>,
```

```
    //This is the display name of the VM
        name = "My New VM",
        version = "vmx-04",
        uuid = <unset>,
        instanceUuid = <unset>,
        npivWorldWideNameType = <unset>,
        npivDesiredNodeWwns = <unset>,
        npivDesiredPortWwns = <unset>,
        npivTemporaryDisabled = <unset>,
        npivOnNonRdmDisks = <unset>,
        npivWorldWideNameOp = <unset>,
        locationId = <unset>,
    //
    // This is advisory, the disk determines the O/S
    guestId = "winNetStandardGuest",
        alternateGuestName = "Microsoft Windows Server 2003, Standard  Edition (32-bit)",
        annotation = <unset>,
        files = (vim.vm.FileInfo) {
            dynamicType = <unset>,
            vmPathName = "[plat004-local]",
            snapshotDirectory = "[plat004-local]",
            suspendDirectory = <unset>,
            logDirectory = <unset>,
        },
        tools = (vim.vm.ToolsConfigInfo) {
            dynamicType = <unset>,
            toolsVersion = <unset>,
            afterPowerOn = true,
            afterResume = true,
            beforeGuestStandby = true,
            beforeGuestShutdown = true,
            beforeGuestReboot = true,
            toolsUpgradePolicy = <unset>,
            pendingCustomization = <unset>,
            syncTimeWithHost = <unset>,
        },
        flags = (vim.vm.FlagInfo) {
            dynamicType = <unset>,
            disableAcceleration = <unset>,
            enableLogging = <unset>,
            useToe = <unset>,
            runWithDebugInfo = <unset>,
            monitorType = <unset>,
            htSharing = <unset>,
            snapshotDisabled = <unset>,
            snapshotLocked = <unset>,
            diskUuidEnabled = <unset>,
            virtualMmuUsage = <unset>,
            snapshotPowerOffBehavior = "powerOff",
            recordReplayEnabled = <unset>,
        },
        consolePreferences = (vim.vm.ConsolePreferences) null,
        powerOpInfo = (vim.vm.DefaultPowerOpInfo) {
            dynamicType = <unset>,
            powerOffType = "preset",
            suspendType = "preset",
            resetType = "preset",
            defaultPowerOffType = <unset>,
            defaultSuspendType = <unset>,
            defaultResetType = <unset>,
            standbyAction = "powerOnSuspend",
        },
    // the number of CPUs
        numCPUs = 1,
    // the number of memory megabytes
        memoryMB = 256,
        memoryHotAddEnabled = <unset>,
        cpuHotAddEnabled = <unset>,
```

```
            cpuHotRemoveEnabled = <unset>,
            deviceChange = (vim.vm.device.VirtualDeviceSpec) [
                (vim.vm.device.VirtualDeviceSpec) {
                    dynamicType = <unset>,
                    operation = "add",
                    fileOperation = <unset>,
// CDROM
                    device = (vim.vm.device.VirtualCdrom) {
                        dynamicType = <unset>,
// key number of CDROM
                        key = -42,
                        deviceInfo = (vim.Description) null,
                        backing = (vim.vm.device.VirtualCdrom.RemotePassthroughBackingInfo) {
                            dynamicType = <unset>,
                            deviceName = "",
                            useAutoDetect = <unset>,
                            exclusive = false,
                        },
                        connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
                            dynamicType = <unset>,
                            startConnected = false,
                            allowGuestControl = true,
                            connected = false,
                        },
// connects to this controller
                        controllerKey = 200,
                        unitNumber = 0,
                    },
                },
                (vim.vm.device.VirtualDeviceSpec) {
                    dynamicType = <unset>,
                    operation = "add",
                    fileOperation = <unset>,
// SCSI controller
                    device = (vim.vm.device.VirtualLsiLogicController) {
                        dynamicType = <unset>,
// key number of SCSI controller
                        key = -44,
                        deviceInfo = (vim.Description) null,
                        backing = (vim.vm.device.VirtualDevice.BackingInfo) null,
                        connectable = (vim.vm.device.VirtualDevice.ConnectInfo) null,
                        controllerKey = <unset>,
                        unitNumber = <unset>,
                        busNumber = 0,
                        hotAddRemove = <unset>,
                        sharedBus = "noSharing",
                        scsiCtlrUnitNumber = <unset>,
                    },
                },
                (vim.vm.device.VirtualDeviceSpec) {
                    dynamicType = <unset>,
                    operation = "add",
                    fileOperation = <unset>,
// Network controller
                    device = (vim.vm.device.VirtualPCNet32) {
                        dynamicType = <unset>,
// key number of Network controller
                        key = -48,
                        deviceInfo = (vim.Description) null,
                        backing = (vim.vm.device.VirtualEthernetCard.NetworkBackingInfo) {
                            dynamicType = <unset>,
                            deviceName = "Virtual Machine Network",
                            useAutoDetect = <unset>,
                            network = <unset>,
                            inPassthroughMode = <unset>,
                        },
                        connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
                            dynamicType = <unset>,
```

```
                startConnected = true,
                allowGuestControl = true,
                connected = true,
            },
            controllerKey = <unset>,
            unitNumber = <unset>,
            addressType = "generated",
            macAddress = <unset>,
            wakeOnLanEnabled = true,
        },
    },
    (vim.vm.device.VirtualDeviceSpec) {
        dynamicType = <unset>,
        operation = "add",
        fileOperation = "create",
// SCSI disk one
        device = (vim.vm.device.VirtualDisk) {
            dynamicType = <unset>,
// key number for SCSI disk one
            key = -1000000,
            deviceInfo = (vim.Description) null,
            backing = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) {
                dynamicType = <unset>,
                fileName = "",
                datastore = <unset>,
                diskMode = "persistent",
                split = false,
                writeThrough = false,
                thinProvisioned = <unset>,
                eagerlyScrub = <unset>,
                uuid = <unset>,
                contentId = <unset>,
                changeId = <unset>,
                parent = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) null,
            },
            connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
                dynamicType = <unset>,
                startConnected = true,
                allowGuestControl = false,
                connected = true,
            },
// controller for SCSI disk one
            controllerKey = -44,
            unitNumber = 0,
// size in MB SCSI disk one
            capacityInKB = 524288,
            committedSpace = <unset>,
            shares = (vim.SharesInfo) null,
        },
    },
    (vim.vm.device.VirtualDeviceSpec) {
        dynamicType = <unset>,
        operation = "add",
        fileOperation = "create",
// SCSI disk two
        device = (vim.vm.device.VirtualDisk) {
            dynamicType = <unset>,
// key number of SCSI disk two
            key = -100,
            deviceInfo = (vim.Description) null,
            backing = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) {
                dynamicType = <unset>,
                fileName = "",
                datastore = <unset>,
                diskMode = "persistent",
                split = false,
                writeThrough = false,
                thinProvisioned = <unset>,
```

```
                  eagerlyScrub = <unset>,
                  uuid = <unset>,
                  contentId = <unset>,
                  changeId = <unset>,
                  parent = (vim.vm.device.VirtualDisk.FlatVer2BackingInfo) null,
              },
              connectable = (vim.vm.device.VirtualDevice.ConnectInfo) {
                  dynamicType = <unset>,
                  startConnected = true,
                  allowGuestControl = false,
                  connected = true,
              },
// controller for SCSI disk two
              controllerKey = –44,
              unitNumber = 1,
// size in MB SCSI disk two
              capacityInKB = 131072,
              committedSpace = <unset>,
              shares = (vim.SharesInfo) null,
          },
      }
   },
   cpuAllocation = (vim.ResourceAllocationInfo) {
      dynamicType = <unset>,
      reservation = 0,
      expandableReservation = <unset>,
      limit = <unset>,
      shares = (vim.SharesInfo) {
         dynamicType = <unset>,
         shares = 100,
         level = "normal",
      },
      overheadLimit = <unset>,
   },
   memoryAllocation = (vim.ResourceAllocationInfo) {
      dynamicType = <unset>,
      reservation = 0,
      expandableReservation = <unset>,
      limit = <unset>,
      shares = (vim.SharesInfo) {
         dynamicType = <unset>,
         shares = 100,
         level = "normal",
      },
      overheadLimit = <unset>,
   },
   cpuAffinity = (vim.vm.AffinityInfo) null,
   memoryAffinity = (vim.vm.AffinityInfo) null,
   networkShaper = (vim.vm.NetworkShaperInfo) null,
   swapPlacement = <unset>,
   swapDirectory = <unset>,
   preserveSwapOnPowerOff = <unset>,
   bootOptions = (vim.vm.BootOptions) null,
   appliance = (vim.vService.ConfigSpec) null,
   ftInfo = (vim.vm.FaultToleranceConfigInfo) null,
   applianceConfigRemoved = <unset>,
   vAssertsEnabled = <unset>,
   changeTrackingEnabled = <unset>,
}
// end of VirtualMachineConfigSpec
```

The information above appears to be quite complex, but much of the input consists of defaulted values to be assigned by the system. The remainder of the supplied information can be extracted from the output of the `config.hardware.device` table from the `PropertyCollector`. Borrowing heavily from the code example in the *VMware vSphere Web Services SDK Programming Guide*, we get the following code that duplicates functionality of the configuration specification seen above:

```
// Duplicate virtual machine configuration
```

```
VirtualMachineConfigSpec configSpec = new VirtualMachineConfigSpec();
// Set the VM values
configSpec.setName("My New VM");
configSpec.setVersion("vmx-04");
configSpec.setGuestId("winNetStandardGuest");
configSpec.setNumCPUs(1);
configSpec.setMemoryMB(256);
// Set up file storage info
VirtualMachineFileInfo vmfi = new VirtualMachineFileInfo();
vmfi.setVmPathName("[plat004-local]");
configSpec.setFiles(vmfi);
vmfi.setSnapshotDirectory("[plat004-local]");
// Set up tools config info
ToolsConfigInfo tools = new ToolsConfigInfo();
configSpec.setTools(tools);
tools.setAfterPowerOn(new Boolean(true));
tools.setAfterResume(new Boolean(true));
tools.setBeforeGuestStandby(new Boolean(true));
tools.setBeforeGuestShutdown(new Boolean(true));
tools.setBeforeGuestReboot(new Boolean(true));
// Set flags
VirtualMachineFlagInfo flags = new VirtualMachineFlagInfo();
configSpec.setFlags(flags);
flags.setSnapshotPowerOffBehavior("powerOff");
// Set power op info
VirtualMachineDefaultPowerOpInfo powerInfo = new VirtualMachineDefaultPowerOpInfo();
configSpec.setPowerOpInfo(powerInfo);
powerInfo.setPowerOffType("preset");
powerInfo.setSuspendType("preset");
powerInfo.setResetType("preset");
powerInfo.setStandbyAction("powerOnSuspend");
// Now add in the devices
VirtualDeviceConfigSpec[] deviceConfigSpec = new VirtualDeviceConfigSpec [5];
configSpec.setDeviceChange(deviceConfigSpec);
// Formulate the CDROM
deviceConfigSpec[0].setOperation(VirtualDeviceConfigSpecOperation.add);
VirtualCdrom cdrom = new VirtualCdrom();
VirtualCdromIsoBackingInfo cdDeviceBacking = new  VirtualCdromRemotePassthroughBackingInfo();
cdDeviceBacking.setDatastore(datastoreRef);
cdrom.setBacking(cdDeviceBacking);
cdrom.setKey(-42);
cdrom.setControllerKey(new Integer(200));
cdrom.setUnitNumber(new Integer(0));
deviceConfigSpec[0].setDevice(cdrom);
// Formulate the SCSI controller
deviceConfigSpec[1].setOperation(VirtualDeviceConfigSpecOperation.add);
VirtualLsiLogicController scsiCtrl = new VirtualLsiLogicController();
scsiCtrl.setBusNumber(0);
deviceConfigSpec[1].setDevice(scsiCtrl);
scsiCtrl.setKey(-44);
scsiCtrl.setSharedBus(VirtualSCSISharing.noSharing);
// Formulate SCIS disk one
deviceConfigSpec[2].setFileOperation(VirtualDeviceConfigSpecFileOperation.create);
deviceConfigSpec[2].setOperation(VirtualDeviceConfigSpecOperation.add);
VirtualDisk disk =  new VirtualDisk();
VirtualDiskFlatVer2BackingInfo diskfileBacking = new VirtualDiskFlatVer2BackingInfo();
diskfileBacking.setDatastore(datastoreRef);
diskfileBacking.setFileName(volumeName);
diskfileBacking.setDiskMode("persistent");
diskfileBacking.setSplit(new Boolean(false));
diskfileBacking.setWriteThrough(new Boolean(false));
disk.setKey(-1000000);
disk.setControllerKey(new Integer(-44));
disk.setUnitNumber(new Integer(0));
disk.setBacking(diskfileBacking);
disk.setCapacityInKB(524288);
deviceConfigSpec[2].setDevice(disk);
// Formulate SCSI disk two
```

```
deviceConfigSpec[3].setFileOperation(VirtualDeviceConfigSpecFileOperation.create);
deviceConfigSpec[3].setOperation(VirtualDeviceConfigSpecOperation.add);
VirtualDisk disk2 =  new VirtualDisk();
VirtualDiskFlatVer2BackingInfo diskfileBacking2 = new VirtualDiskFlatVer2BackingInfo();
diskfileBacking2.setDatastore(datastoreRef);
diskfileBacking2.setFileName(volumeName);
diskfileBacking2.setDiskMode("persistent");
diskfileBacking2.setSplit(new Boolean(false));
diskfileBacking2.setWriteThrough(new Boolean(false));
disk2.setKey(-100);
disk2.setControllerKey(new Integer(-44));
disk2.setUnitNumber(new Integer(1));
disk2.setBacking(diskfileBacking2);
disk2.setCapacityInKB(131072);
deviceConfigSpec[3].setDevice(disk2);
// Finally, formulate the NIC
deviceConfigSpec[4].setOperation(VirtualDeviceConfigSpecOperation.add);
com.VMware.vim.VirtualEthernetCard nic =  new VirtualPCNet32();
VirtualEthernetCardNetworkBackingInfo nicBacking = new VirtualEthernetCardNetworkBackingInfo();
nicBacking.setNetwork(networkRef);
nicBacking.setDeviceName(networkName);
nic.setAddressType("generated");
nic.setBacking(nicBacking);
nic.setKey(-48);
deviceConfigSpec[4].setDevice(nic);
// Now that it is all put together, create the virtual machine
// Note that folderMo, hostMo, and resourcePool are MoRefs to the Folder, Host, and ResourcePool
// where the VM is to be created
ManagedObjectReference taskMoRef = serviceConnection.getService().createVM_Task(
    folderMo, configSpec, resourcePool, hostMo);
```

Using VirtualMachineConfigInfo to Create a Virtual Machine

A backup application can also use the information contained in a `VirtualMachineConfigInfo`. If you preserve all of the information in a `VirtualMachineConfigInfo` that describes the virtual machine, you can transfer much of this information into a `VirtualMachineConfigSpec` in order to create a virtual machine. However, some of the information in `VirtualMachineConfigInfo` is not needed, and if used in the `Spec`, virtual machine creation can fail. For example, a `VirtualMachineConfigSpec` that contains information about so called "Default Devices" usually fails. The list of default devices contains:

```
vim.vm.device.VirtualIDEController
vim.vm.device.VirtualPS2Controller
vim.vm.device.VirtualPCIController
vim.vm.device.VirtualSIOController
vim.vm.device.VirtualKeyboard
vim.vm.device.VirtualVMCIDevice
vim.vm.device.VirtualPointingDevice
```

However, other controllers and devices must be explicitly included in the `VirtualMachineConfigSpec`. Some information about devices is also unneeded and can cause problems if supplied. Each controller device has its `vim.vm.device.VirtualController.device` field, which is an array of devices that report to the controller. The server rebuilds this list when a virtual machine is created using the (negative) device key numbers supplied as a guide.

The relationship between controller and device must be preserved using negative key numbers in the same relationship as in the hardware array of the `VirtualMachineConfigInfo`.

One other feature of a `VirtualMachineConfigInfo` requires substitution. `VirtualMachineConfigInfo` contains the field `cpuFeatureMask`, which is an array of `HostCpuIdInfo`. The entries in this array must be converted to `ArrayUpdateSpec` entries that contain the `VirtualMachineCpuIdInfoSpec` along with a field called "operation" that must contain the value `ArrayUpdateOperation::add`.

The `VirtualMachineCpuIdInfoSpec` also contains a `HostCpuIdInfo` that you can copy from the `cpuFeatureMask` array in `VirtualMachineConfigInfo`.

Everything else can be copied intact from the `VirtualMachineConfigInfo` data.

Restoring the Data

As in the section "Restoring a Virtual Disk" on page 14, `VixDiskLib` functions provide interfaces for writing the data to virtual disk, either locally or remotely.

Changed Block Tracking on Virtual Disks

On hosts running ESX 4.0 and later, virtual machines can keep track of disk sectors that have changed. This is called Changed Block Tracking. Its method in the VMware vSphere API is `QueryChangedDiskAreas`, which takes the following parameters:

- `_this` – Managed Object Reference to the virtual machine.

- `snapshot` – Managed Object Reference to a Snapshot of the virtual machine.

- `deviceKey` – Virtual disk for which to compute the changes.

- `startOffset` – Byte offset where to start computing changes to virtual disk. The length of virtual disk sector(s) examined is returned in `DiskChangeInfo`.

- `changeId` – An identifier for the state of a virtual disk at a specific point in time. A new `ChangeId` results every time someone creates a snapshot. You should retain this value with the version of change data that you extract (using `QueryChangedDiskAreas`) from the snapshot's virtual disk.

  When you back up a snapshot, you either have a previously saved `ChangeId` or you do not. If you have a saved `ChangeId`, it identifies the last time you performed a backup, and tells the changed block tracking logic to identify those that have occurred since the time indicated by the `ChangeId`. If you do not have a saved `ChangeId`, then you must save a baseline (full) backup of the virtual disk. There are two ways to get this baseline backup: (1) is to directly save all of the virtual disk's contents, and (2) is to provide the special `ChangeId` "*" (star). The star `ChangeId` indicates that only active portions of the virtual disk should be returned by `QueryChangedDiskAreas`. For both thin provisioned (sparse) virtual disks and for ordinary virtual disks, this represents a substantial reduction in the amount of data to be saved.

- Identifier for a time in the past. This can be star "*" to identify all allocated areas of virtual disk, ignoring unallocated areas (of sparse disk), or it could be a `changeId` string saved at the time of snapshot backup.

  It only makes sense to use the special `ChangeId` = "*" when no previous `ChangeId` exists. If a previous `ChangeId` does exist, then `QueryChangedDiskAreas` returns the disk sectors that changed since the new `ChangeId` was collected. The following table shows the algorithm:

New Change ID	Old Change ID	Used for Query	Result
change 0	none	*	All in-use sectors of the disk.
change 1	change 0	change 0	All sectors altered since change 0.

Enabling Changed Block Tracking

This feature is disabled by default, because it reduces performance by a small but measurable amount. If you query the virtual machine configuration, you can determine if it is capable of Changed Block Tracking. Use the property collector to retrieve the capability field from the `VirtualMachineManagedObject`. If the capability field contains the flag `changeTrackingSupported`, then you can proceed. The virtual machine version must be 7 or higher to support this. If the virtual machine version is lower than 7, upgrade the virtual hardware.

If supported, you enable Changed Block Tracking using an abbreviated form of `VirtualMachineConfigSpec`, then use the `ReconfigVM_Task` method to reconfigure the virtual machine with Changed Block Tracking:

```
VirtualMachineConfigSpec configSpec = new VirtualMachineConfigSpec;
configSpec.SetChangeTrackingEnabled(new Boolean(true));
ManagedObjectReference taskMoRef =
   serviceConnection.getService().ReconfigureVm_Task(targetVM_MoRef, configSpec);
```

Powered-on virtual machines must go through a stun-unstun cycle (power on, resume after suspend, migrate, or snapshot create/delete/revert) before the reconfiguration takes effect.

To enable Changed Block Tracking with the VMware vSphere Client:

1   Select the virtual machine and ensure that **Summary > VM Version** says "7" for virtual hardware version.

2   In the Summary tab, click **Edit Settings > Options > Advanced > General**.

3   In the right side of the dialog box, click **Configuration Parameters...**

4   In the new dialog box, locate the row for name `ctkEnabled`, and change its value from `false` to `true`.

5   See above concerning the stun-unstun cycle.

To enable Changed Block Tracking and back up with the VMware vSphere API:

1   Query change tracking status of the virtual machine. If false, activate Changed Block Tracking.

2   Create a snapshot of the virtual machine. The snapshot operation causes a stun-unstun cycle.

3   Starting from the snapshot's `ConfigInfo`, work your way to the `BackingInfo` of all virtual disks in the snapshot. This gives you the change IDs for all the disks of the virtual machine.

4   Hold onto the change IDs and do a full backup of the snapshot, since this is the first time for backup.

5   Delete the snapshot when your backup has completed.

6   Next time you back up this virtual machine, create a snapshot and use `QueryChangedDiskAreas` with the change IDs from your previous backup to take advantage of Changed Block Tracking.

Gathering Changed Block Information

Associated with Changed Block Tracking is `changeId`, an identifier for versions of changed block data. Whenever a virtual machine snapshot is created, associated with that snapshot is a `changeId` that functions as a landmark to identify changes in virtual disk data. So it follows that when a snapshot is created for the purpose of creating an initial virtual disk backup, the `changeId` associated with that snapshot can be used to retrieve changes that have occurred since snapshot creation.

To obtain the `changeId` associated with any disk in a snapshot, you examine the "hardware" array from the snapshot. Any item in the devices table that is of type `vim.vm.device.VirtualDevice.VirtualDisk` encloses a class describing the "backing storage" (obtained using `getBacking`) that implements virtual disk. If backing storage is one of the following types, you can use the `getChangeId` method to obtain the `changeId`:

```
vim.vm.device.VirtualDevice.VirtualDisk.FlatVer2BackingInfo
vim.vm.device.VirtualDevice.VirtualDisk.SparseVer2BackingInfo
vim.vm.device.VirtualDevice.VirtualDisk.RawDiskVer2BackingInfo
```

Information returned from the `QueryChangedDiskAreas` method is a `VirtualMachine.DiskChangeInfo` data object containing a description of the area of the disk covered (start offset and length), and an array of `VirtualMachine.DiskChangeInfo.DiskChangeExtent` items that describe the starting offset and length of various disk areas that have changed.

To find change information, you can use the Managed Object Browser at http://<ESXhost>/mob to follow path **content > ha-folder-root > ha-datacenter > datastore > vm > snapshot > config > hardware > virtualDisk**. Changed block tracking is associated with the snapshot, not with the base disk.

The following code sample assumes that, in the past, you obtained a complete copy of the virtual disk, and at the time when the `changeId` associated with the snapshot was collected, you stored it for use at a later time, which is now. A new snapshot has been created, and the appropriate `MoRef` is available:

```
String changeId; // Already initialized
ManagedObjectReference theSnapshot;
ManagedObjectReference theVM;
int diskDeviceKey;
VirtualMachine.DiskChangeInfo changes;
long position = 0;
do {
   changes = theVM.queryChangedDiskAreas(theSnapshot,
                                  diskDeviceKey,
                                  position, changeId);
```

```
      for (int i = 0; i < changes.changedArea.length; i++) {
         long length = changes.changedArea[i].length;
         long offset = changes.changedArea[i].startOffset;
         //
         // Go get and save disk data here
      }
      position = changes.startOffset + changes.length;
   } while (position < diskCapacity);
```

In the above code, `QueryChangedDiskAreas` is called repeatedly, as `position` moves through the virtual disk. This is because the number of entries in the `ChangedDiskArea` array could end up occupying a large amount of memory for describing many changes to a large virtual disk.

The `changeId` (changed block ID) contains a sequence number in the form `<UUID>/<nnn>`. If the `<UUID>` changes, it indicates that tracking information has become invalid, necessitating a full backup. Otherwise incremental backups can continue in the usual pattern.

Troubleshooting

If you reconfigure a virtual machine to set `changeTrackingEnabled`, but the property remains false, check that you have queried the virtual machine status with `VirtualMachine->config()` after reconfiguration with `VirtualMachine->reconfigure()`. Yes, you must do it again. Also make sure that the virtual machine is hardware version 7 and has undergone a stun-unstun cycle since reconfiguration.

Checking for Namespace

You can avoid using the `queryChangedDiskAreas` API on ESX 3.5/3.5i (and earlier) storage by parsing XML files for the namespace. For prepackaged methods that do this, see these SDK code samples:

```
Axis/java/com/vmware/samples/version/displaynewpropertieshost/DisplayNewPropertiesHostV25.java
Axis/java/com/vmware/samples/version/getvirtualdiskfiles/GetVirtualDiskFilesV25.java
DotNet/cs/DisplayNewProperties/DisplayNewPropertiesV25.cs
DotNet/cs/GetVirtualDiskFiles/GetVirtualDiskFilesV25.cs
```

Limitations on Changed Block Tracking

Changed Block Tracking does not work in any of the following cases:

- Virtual hardware version is 6 or earlier.

- The virtual disk is a "physical compatibility RDM."

- The virtual disk is attached to a shared virtual SCSI bus.

Changed Block Tracking can be enabled on virtual machines that have disks like these, but when queried for their change ID, these disks always return an empty string. So if you have a virtual machine with a regular "O/S disk" and a pass-through RDM as a "data disk" you can track changes on the "O/S disk" only.

Incremental Restore of Backup Data

At some point you might need to restore a virtual disk from the backup data that you gathered as described in section "Changed Block Tracking on Virtual Disks" on page 21. The essential procedure is as follows:

1   Power off the virtual machine, if powered on.

2   Using the `VirtualMachineConfigInfo` that corresponds to the last known good state of the guest operating system, re-create the virtual machine as described in section "Using VirtualMachineConfigInfo to Create a Virtual Machine" on page 20.

3   Completely reload the base virtual disk using the full backup that started the most recent series of incremental backups.

4   Create a snapshot, as a child of the base virtual disk.

5   Sequentially restore the incremental backup data. You can do this either forwards or backwards. If you work forwards, the restore might write some sectors more than once. If you work backwards, you must keep track of which sectors were restored so as to avoid restoring them again from older data.

6   Get the change ID of the backup this is being restored, and use the change tracking APIs to find the areas of the disk that need to be restored. This information must be cached, because once you start restoring a virtual disk, the change tracking mechanism will fail.

7   Restore only changed areas to the virtual disks referred to by the snapshot. This ensures that you do not write the data to the redo log created by the snapshot. When restoring a thin provisioned (sparse) disk, use the star "*" change ID to avoid writing zeroes to the unallocated blocks.

8   Repeat Step 6 and Step 7 as necessary by applying incremental backup data sets in order.

9   Optionally, revert to the base virtual disk, thus eliminating the snapshot.

Working with Microsoft Shadow Copy

Microsoft Shadow Copy, also called Volume Snapshot Service (VSS), is a Windows Server data backup feature for creating consistent point-in-time copies of data (called shadow copies).

When VMware Tools perform VSS quiescing while creating a snapshot in a Windows guest, they generate a `vss-manifest.zip` file. This file contains the backup component document (BCD) and writer manifests. The host agent stores this manifest file in the `snapshotDir` of the virtual machine. Backup applications should use the HTTP protocol to obtain the `vss-manifest.zip` file so they can save it.

You can access this HTTP protocol either through the `CopyDatastoreFile_Task` method in the vSphere API, or using the `vifs` command in the vCLI (in ESX/ESXi 4), formerly called the RCLI.

Restore must be done using the backup application's guest agent. VMware vStorage APIs for Data Protection provide no host agent support for this. Applications authenticating with SSPI might not work right because HTTP access will demand a user name and password, unless the session was recently authenticated.

For information about VSS, see the Microsoft TechNet article, *How Volume Shadow Copy Service Works*. For information about Security Support Provider Interface (SSPI), see the MSDN Web site.

Conclusion

This technical note introduced the VMware vStorage APIs for Data Protection, giving an overview of backup and restore operations in VMware vSphere. The information presented here should help you understand the underlying concepts and steps to take for implementing applications to protect virtual-machine data.