vSAN SDKs Programming Guide

VMware vSAN 6.7

vmware®

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

If you have comments about this documentation, submit your feedback to

docfeedback@vmware.com

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

1 Introduction to the vSAN Management SDKs 4

2 Using the vSAN Management SDKs 5

- vSAN Management SDK for Java 5
- vSAN Management SDK for .NET 6
- vSAN Management SDK for Python 7
- vSAN Management SDK for Perl 8
- vSAN Management SDK for Ruby 9

3 Setting Up a vSAN Cluster 11

- Connecting to vCenter Server and Selecting a Cluster for vSAN 11
- Configuring VMkernel Networking for vSAN 12
- Enabling vSAN on a Cluster 13
- Claiming and Managing Disks 14
- Enabling Deduplication and Compression on All-Flash Clusters 16
- Configuring Fault Domains 17
- Assigning the vSAN License 18

4 Configuring Stretched and Two-Host Clusters 19

- Deploying the vSAN Witness Appliance 19
- Adding the vSAN Witness Appliance to vCenter Server 23
- Configuring a vSAN Stretched Cluster or Two-Host Cluster 24

5 Upgrading the vSAN On-Disk Format 26

- Determining the Current vSAN On-Disk Format 26
- Performing the On-Disk Upgrade Preflight Check 27
- Upgrading with Reduced Redundancy 28

6 Managing iSCSI Service 29

- Enabling vSAN iSCSI Service 29
- Creating iSCSI Targets and LUNs 30
- Disabling iSCSI Service 30

7 Monitoring vSAN 32

- Viewing vSAN Health Check Status 32
- Monitoring vSAN Performance 32

Introduction to the vSAN Management SDKs

1

The vSAN Management SDKs bundle language bindings for accessing the vSAN Management API and creating client applications for automating vSAN management tasks.

The vSAN Management API

The vSAN Management API is an extension of the vSphere API. Both vCenter Server® and ESXi hosts expose the vSAN Management API. You can use the vSAN Management API to implement the client applications that perform the following tasks:

- Configure a vSAN cluster - Configure all aspects of a vSAN cluster, such as set up VMkernel networking, claim disks, configure fault domains, enable the deduplication and compression of all flash clusters, and assign the vSAN license.
- Configure a vSAN stretched cluster - Deploy the vSAN Witness Appliance and configure a vSAN stretched cluster.
- Upgrade the vSAN on-disk format.
- Track the vSAN performance.
- Monitor the vSAN health.

The vSAN Management SDKs

The vSAN Management SDKs are separated into five programming languages that you can use to access the vSAN Management API with similar functionality and develop client applications for managing vSAN clusters.

Using the vSAN Management SDKs

2

The vSAN Management SDKs are separated into five different programming languages, Java, .NET, Python, Perl, and Ruby. Each of the five vSAN Management SDKs depends on the vSphere SDK with similar functionality delivered for the corresponding programming language.

You can download these vSphere SDKs from https://code.vmware.com/home or from Github.

This chapter includes the following topics:

- vSAN Management SDK for Java
- vSAN Management SDK for .NET
- vSAN Management SDK for Python
- vSAN Management SDK for Perl
- vSAN Management SDK for Ruby

vSAN Management SDK for Java

The vSAN Management SDK for Java provides WSDL files, sample code, and API reference for developing custom Java clients against the vSAN Management API. The vSAN Management SDK for Java depends on the vSphere Web Services SDK of similar level. You use the vSphere Web Services SDK for logging in to vCenter Server and for retrieving vCenter Server managed objects.

API Reference

The vSAN API reference documentation is included in the `/docs` directory. To view the API Reference, open `index.html` with a Web browser.

WSDL Files and vSAN Java Bindings

The vSAN Management SDK for Java includes the `vsan.wsdl` and `vsanService.wsdl` files in the `bindings/wsdl` directory. You can use the WSDL definitions to build Java bindings for accessing the vSAN Management API. You can build Java bindings using the `build.py` script.

Note You must have Python 2.7.13 or later to run the `build.py` script.

Running the Sample Applications

The vSAN Management SDK for Java includes sample applications, `build` and `run` scripts, and dependent libraries. They are located under the `samplecode` directory in the SDK.

You can use the sample code to get vSAN managed objects on vCenter Server or ESXi hosts.

Before running the sample applications, make sure that you have the vSphere Web Services SDK on your development environment, with the following directory structure:

```
VMware-vSphere-SDK-<version number>-build
    SDK
        vsphere-ws
```

Then copy the `vsan-sdk-java` directory at the same level as the `vsphere-vs` directory in the vSphere Web Services SDK:

```
VMware-vSphere-SDK-<version number>-build
    SDK
        vsphere-ws
        vsan-sdk-java
```

Build the sample applications by running the `build.py` command.

Run the sample applications using the `run.sh` script on Linux, or the `run.bat` script on Windows:

```
./run.sh com.vmware.vsan.samples.<sample_name>
    --url https://<vCenter Server or host address>/sdk
    --username <username>
    --password <password>
```

To get information about the parameter usage, use -h or --help.

vSAN Management SDK for .NET

The vSAN Management SDK for .NET provides libraries, sample code, and API reference for developing custom .NET clients against the vSAN Management API. The vSAN Management SDK for .NET depends on the vSphere Web Services SDK of similar level. You use the vSphere Web Services SDK for logging in to vCenter Server and for retrieving vCenter Server managed objects.

API Reference

The vSAN API reference documentation is included in the `/docs` directory. To view the API Reference, open `index.html` with a Web browser.

WSDL Files

The vSAN Management SDK for .NET includes `vsan.wsdl` and `vsanService.wsdl` in the `bindings/wsdl` directory. You can use the WSDL definitions to build C# bindings for accessing the vSAN Management API.

Building the vSAN C# DLL

You must have the following components to build the vSAN C# DLL:

- `csc.exe`. A C# compiler
- `sgen.exe`. An XML serializer generator tool
- `wsdl.exe`. Web Service Description Language 4.0 for Microsoft .NET
- `Microsoft.Web.Services3.dll`
- .NET Framework 4.0
- Python 2.7.6

To build the vSAN C# DLL, run the following command:

```
$ python builder.py vsan_wsdl vsanservice_wsdl
```

This command generates the following DLL files:

- `VsanhealthService.dll`
- `VsanhealthService.XmlSerializers.dll`

Running the Sample Applications

To run the sample applications, run the following command:

```
.\VsanHealth.exe --username <host or vCenter Server username>
    --url https://<host or vCenter Server address>/sdk
    --hostName <host or cluster name> --ignorecert --disablesso
```

To view information about the parameters, use `--help`.

vSAN Management SDK for Python

The vSAN Management SDK for Python provides language bindings, sample code, and API reference for developing custom Python clients against the vSAN Management API. The vSAN Management SDK for Python depends on `pyVmomi` of similar release level, which is the Python SDK for the vSphere API. You use `pyVmomi` for logging in to vCenter Server and for retrieving vCenter Server managed objects.

Note VMware does not officially support `pyVmomi`, but you can download it from GitHub.

API Reference

The vSAN API reference documentation is included in the `/docs` directory. To view the API Reference, open `index.html` with a Web browser.

Python Bindings

You can access the vSAN Management API by using the Python `vsanmgmtObjects.py` script under the `bindings` directory.

To use the Python bindings, place `vsanmgmtObjects.py` on a path where your Python applications import.

Running the Sample Applications

The vSAN Management SDK for Python provides sample applications, which you can find under the `samplecode` directory.

You can use the sample code to get vSAN managed objects on vCenter Server or ESXi hosts. The code automatically identifies the target server type.

The `vsaniscsisamples.py` and `vsaniscsisamples.py` depend on the `vsanapiutis.py`, which provides utility libraries for retrieving vSAN managed objects.

To run the sample applications, use the following commands:

```
python vsanapisamples.py –s <host or vCenter Server address> –u <username> –p <password>
    ––cluster <cluster name>
python vsaniscsisamples.py –s <host or vCenter Server address> –u <username> –p <password>
    ––cluster <cluster name>
```

To view information about the parameter usage, use `–h` or `––help`.

vSAN Management SDK for Perl

The vSAN Management SDK for Perl provides libraries, sample code, and API reference for developing custom Perl clients against the vSAN Management API. The vSAN Management SDK for Perl depends on viperl of similar release level, which is the Perl SDK for the vSphere API. You use viperl for logging in to vCenter Server and for retrieving vCenter Server managed objects. VI Perl Toolkit, which is a client-side framework from VMware that simplifies the programming effort associated with the VI API.

API Reference

The vSAN API reference documentation is included in the `/docs` directory. To view the API Reference, open `index.html` with a Web browser.

Perl Bindings

You can access the vSAN Management API by using the `VIM25VsanmgmtRuntime.pm` and `VIM25VsanmgmtStub.pm` files that are located under the `bindings` directory. To use the Perl bindings, place these files on a path where Perl can find them.

Running the Sample Applications

The vSAN Management SDK for Perl SDK provides sample applications that are located under the `samplecode` directory.

You can use the sample code to get vSAN managed objects on vCenter Server or ESXi hosts. The code automatically identifies the target server type.

The `vsanapisamples.pl` depends on the `VsanapiUtil.pm`, which provides a utility library for retrieving vSAN managed objects.

To test the vCenter Server side API, run the following sample:

```
vsanapisample.pl --url https://<host>:<port>/sdk/vimService
    --username <username> --password <mypassword> --cluster_name <cluster name>
vsanapisample.pl --url https://<host>:<port>/sdk/vimService
    --username <username> --password <mypassword> --cluster_moid <cluster manager object ID>
```

Use this sample to test the iSCSI target service:

```
vsaniscsisample.pl --url https://<host>:<port>/sdk/vimService
    --username <username> --password <mypassword> --cluster_name <cluster name>
vsaniscsisample.pl --url https://<host>:<port>/sdk/vimService
    --username <username> --password <mypassword> --cluster_moid <cluster manager object ID>
```

To test the ESXi side API:

```
vsanapisample.pl --url https://<host>:<port>/sdk
    -username <username> --password <mypassword>
```

To view information about the parameters, use `--help`.

vSAN Management SDK for Ruby

The vSAN Management SDK for Ruby provides language bindings, sample code, and API reference for developing custom Ruby clients against the vSAN Management API. The
vSAN Management SDK for Ruby depends on `RbVmomi` of similar release level, which is the Ruby SDK for the vSphere API. You use RbVmomi for logging in to vCenter Server and to retrieve vCenter Server managed objects.

Note VMware does not officially support `rbVmomi`, but you can download it from GitHub.

API Reference

The vSAN API reference documentation is included in the `/docs` directory. To view the API Reference, open `index.html` with a Web browser.

Ruby Bindings

You can access the vSAN Management API by using `vsanmgmt.api.rb` file under the `bindings` directory. Place the file on a path where Ruby can find it.

Running the Sample Applications

The vSAN Management SDK for Ruby SDK provides sample applications that are located under the `samplecode` directory.

You can use the sample code to get vSAN managed objects on vCenter Server or ESXi hosts. The code automatically identifies the target server type.

The `vsanapisamples.rb` depends on the `vsanapiutis.rb`, which provides a utility library for retrieving vSAN managed objects.

To run the Ruby sample applications, use the following commands:

```
ruby vsanapisamples.rb –o <host or vCenter Server address> –u <username> –p <password>
    <cluster name>
ruby vsaniscsisamples.rb –o <host or vCenter Server address> –u <username> –p <password>
      <cluster name>
```

Use `–h` or `––help` to view information about the parameters.

Setting Up a vSAN Cluster

3

By using the vSAN Management API, you can automate the configuration of a cluster for vSAN or you can configure multiple clusters at a time. The procedure for setting up a vSAN cluster using the vSAN Management API is similar to the procedure that you follow while using the vSphere Client.

Note All examples in this chapter are in Python language.

This chapter includes the following topics:

- Connecting to vCenter Server and Selecting a Cluster for vSAN
- Configuring VMkernel Networking for vSAN
- Enabling vSAN on a Cluster
- Claiming and Managing Disks
- Enabling Deduplication and Compression on All-Flash Clusters
- Configuring Fault Domains
- Assigning the vSAN License

Connecting to vCenter Server and Selecting a Cluster for vSAN

Before configuring a vSAN cluster by using the vSAN Management API, you must establish a secure connection with vCenter Server and filter the clusters on which you want to enable vSAN.

In the following example, a secure connection is established with the vCenter Server using the user name and password authentication. Later, the `getClusterInstance` function is called by passing the cluster name as an argument.

```
if sys.version_info[:3] > (2, 7, 8):
    context = ssl.create_default_context()
    context.check_hostname = False
    context.verify_mode = ssl.CERT_NONE

# Connect to vCenter Server
si = SmartConnect(
    host=args.host,
    user=args.user,
```

```
    pwd=password,
    port=int(args.port),
    sslContext=context)

# Disconnect from vSAN upon exit
atexit.register(Disconnect, si)

# Connect to the cluster passed as an argument
cluster = getClusterInstance(args.clusterName, si)
```

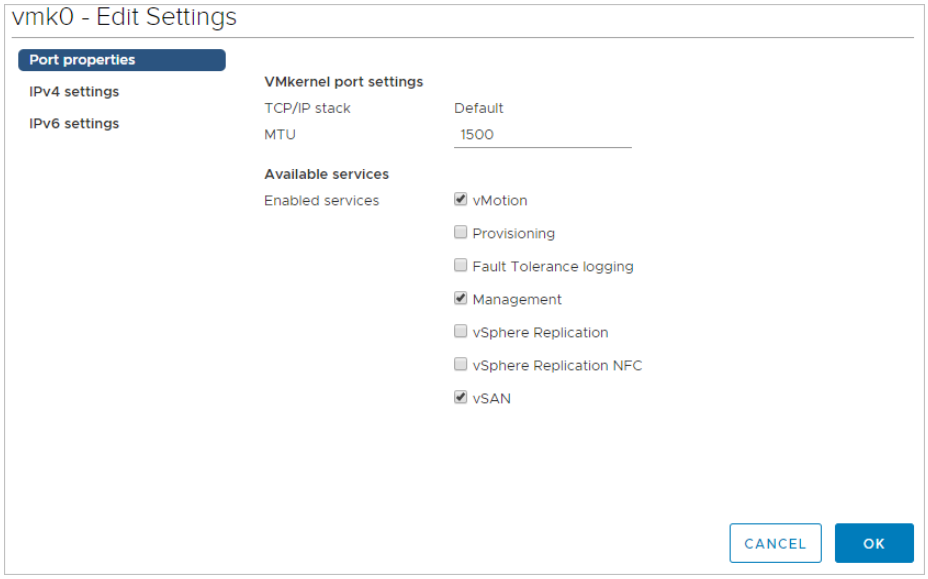
After establishing a secure connection with the vCenter Server and identifying the cluster, a connection is made to that cluster. The `getClusterInstance` function can be reused across the client application to connect to the clusters on which you want to configure vSAN.

```
def getClusterInstance(clusterName, serviceInstance):
    content = serviceInstance.RetrieveContent()
    searchIndex = content.searchIndex
    datacenters = content.rootFolder.childEntity


# Look for the cluster in each datacenter attached to vCenter Server
for datacenter in datacenters:
    cluster = searchIndex.FindChild(datacenter.hostFolder, clusterName)
    if cluster is not None:
        return Cluster
    else:
        return None
```

Configuring VMkernel Networking for vSAN

You must configure every host that is part of the vSAN cluster with a VMkernel adapter that is tagged for vSAN.

In the vSphere Client, you configure VMkernel networking for vSAN on each host by using a standard switch. You can also do this by using a vSphere Distributed Switch for easier and consistent configuration. In both cases, prior to configuring the cluster for vSAN, you must configure the hosts with VMkernel network adapters for vSAN.



When you configure vSAN on a cluster, the Configure vSAN wizard validates the networking configuration on the hosts. If some of the hosts does not have the VMkernel network adapter enabled for vSAN, then you must suspend the configuration of the cluster, and set up the host networking for the vSAN traffic.

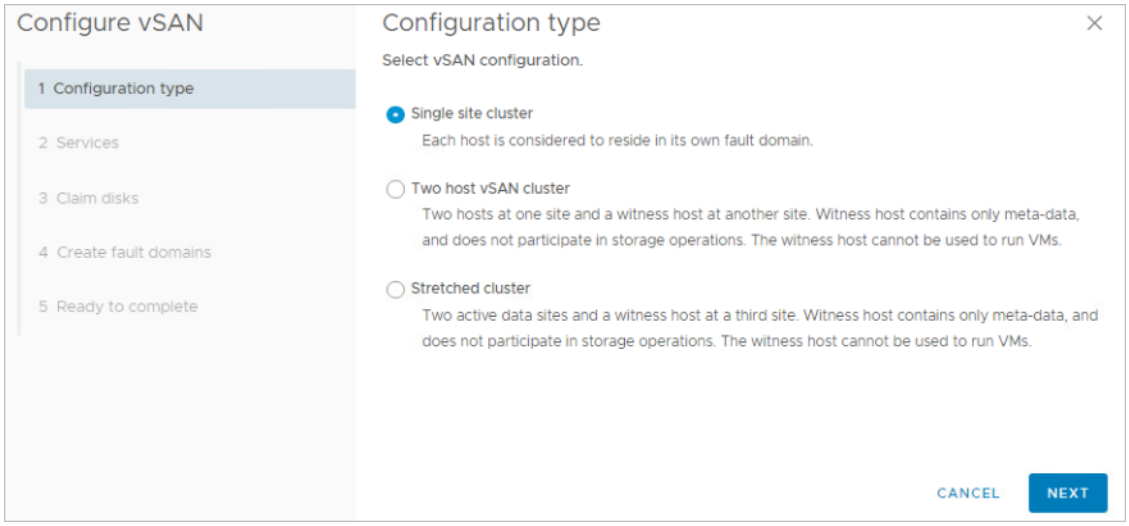
In your client applications, you can set up preselected VMkernel network adapters for the vSAN traffic.

```
# Update configuration spec for VMkernel networking
configInfo = vim.vsan.host.ConfigInfo(
    networkInfo=vim.vsan.host.ConfigInfo.NetworkInfo(port=[
        vim.vsan.host.ConfigInfo.NetworkInfo.PortConfig(device=args.vmknic)
    ]))

# Enumerate the selected VMkernel adapter for each host and add it to the list of tasks
for hosts in hosts:
    print 'Enable vSAN traffic on host {} with {}'.format(
        hostProps[host]['name'], args.vmknic)
task = hostProps[host]['configManager.vsanSystem'].UpdateVsan_Task(configInfo)
tasks.append(task)

# Execute the tasks
vsanapiutils.WaitForTasks(tasks, si)
```

Enabling vSAN on a Cluster

After filtering the clusters that you want to configure for vSAN, the next step is to enable vSAN on these clusters.

In the vSphere Client, you use the **Configure vSAN** wizard to configure individual clusters for vSAN. You must use the **Configure vSAN** wizard to configure each cluster.



To enable vSAN in your vSAN Management API client applications, build an object of type `VimVsanReconfigSpec` by passing a `VsanClusterConfigInfo` parameter with the property `enable` set to `true`.

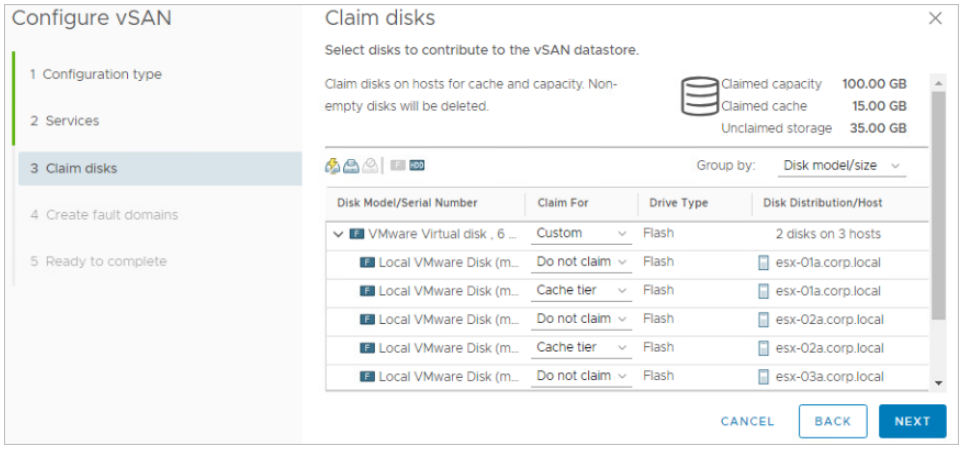
```
# Build vsanReconfigSpec step by step.  It takes effect only after calling the VsanClusterReconfig
method
clusterConfig = vim.VsanClusterConfigInfo(enabled=True)
vsanReconfigSpec = vim.VimVsanReconfigSpec(
    modify=True, vsanClusterConfig=clusterConfig)
```

Claiming and Managing Disks

You can add disks to the vSAN cluster during the initial configuration, or you can add them later on.

When claiming disks by using the **Configure vSAN** wizard in the vSphere Client, you can only see the disks that are eligible, meaning they do not have existing vSAN partitions. vCenter Server filters out the non-eligible disks and they are not exposed for adding to the vSAN cluster.

For both hybrid and all-flash vSAN clusters, you assign the devices for cache and capacity tiers.



While claiming disks using the client applications, do the following:

Query for ineligible disks, and if required, clear the existing vSAN partitions on them.

```
# Enumerate the ineligible disks
for host in hosts:
    disks = [
        result.disk
        for result in hostProps[host]['configManager.vsanSystem']
        .QueryDisksForVsan() if result.state == 'ineligible'
    ]
    print 'Find ineligible disks {} in host {}'.format(
        [disk.displayName for disk in disks], hostProps[host]['name'])

# For each disk, interactively ask admin whether to individually wipe ineligible disks or not
for disk in disks:
    if yes('Do you want to wipe disk {}?\nPlease Always check the partition table and the data'
           ' stored on those disks before doing any wipe! (yes/no)?'.format(
               disk.displayName)):
        hostProps[host]['configManager.storageSystem'].UpdateDiskPartitions(
            disk.deviceName, vim.HostDiskPartitionSpec())
```

If the vSAN cluster is all-flash configuration, then segregate the devices as small and large. If the vSAN cluster is hybrid configuration then segregate the devices as flash and HDD.

```
diskmap = {host: {'cache': [], 'capacity': []} for host in hosts}
cacheDisks = []
capacityDisks = []

# For all flash architectures
if isallFlash:
    for host in hosts:
        ssds = [result.disk for result in hostProps[host]
        ['configManager.vsanSystem'].QueryDisksForVsan() if
                result.state == 'eligible' and result.disk.ssd]
smallerSize = min([disk.capacity.block * disk.capacity.blockSize for disk in ssds])
for ssd in ssds:
    size = ssd.capacity.block * ssd.capacity.blockSize
if size == smallerSize:
    diskmap[host]['cache'].append(ssd)
cacheDisks.append((ssd.displayName, sizeof_fmt(size), hostProps[host]['name']))
else:
diskmap[host]['capacity'].append(ssd)
capacityDisks.append((ssd.displayName, sizeof_fmt(size), hostProps[host]['name']))
else:

# For hybrid architectures
for host in hosts:
    disks = [result.disk for result in hostProps[host]
    ['configManager.vsanSystem'].QueryDisksForVsan() if
             result.state == 'eligible']
ssds = [disk for disk in disks if disk.ssd]
hdds = [disk for disk in disks if not disk.ssd]
for disk in ssds:
    diskmap[host]['cache'].append(disk)
```

```
size = disk.capacity.block * disk.capacity.blockSize
cacheDisks.append((disk.displayName, sizeof_fmt(size), hostProps[host]['name']))
for disk in hdds:
    diskmap[host]['capacity'].append(disk)
size = disk.capacity.block * disk.capacity.blockSize
capacityDisks.append((disk.displayName, sizeof_fmt(size), hostProps[host]['name']))
for host, disks in diskmap.iteritems():
    if disks['cache'] and disks['capacity']:
        dm = vim.VimVsanHostDiskMappingCreationSpec(
            cacheDisks=disks['cache'], capacityDisks=disks['capacity'],
            creationType='allFlash' if isallFlash else 'hybrid',
            host=host)

# Execute the task
task = vsanVcDiskManagementSystem.InitializeDiskMappings(dm)
tasks.append(task)
```

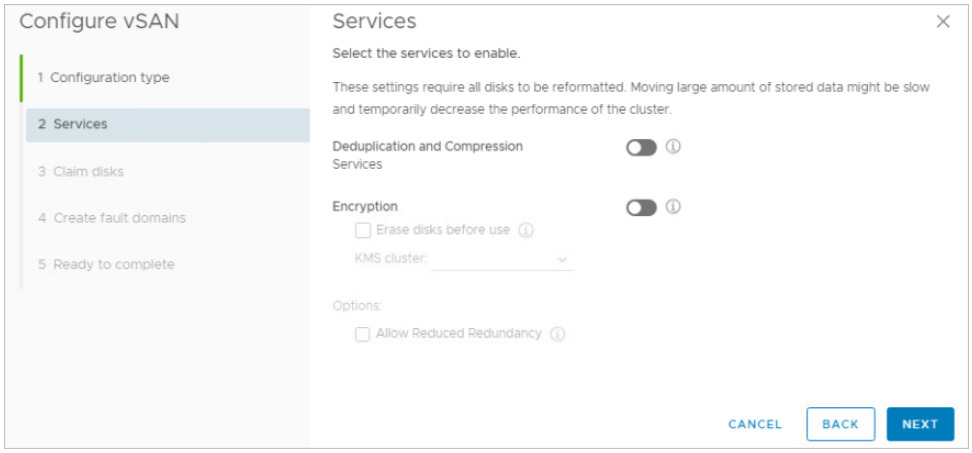
Enabling Deduplication and Compression on All-Flash Clusters

It is often advantageous to enable Deduplication and Compression on All-Flash vSAN deployments. Reduced capacity utilization can often make All-Flash vSAN more cost effective than Hybrid vSAN deployments.

If you are using the Configure vSAN wizard, the Enable Deduplication and Compression option can be selected while creating the vSAN cluster. However, after creating vSAN cluster, the process of enabling or disabling the Deduplication and Compression option using the Configure vSAN wizard can be time consuming. In some scenarios it might lead to reduced availability.

However, using the vSAN Management API, the process of enabling the Deduplication and Compression is simple.

In the vSphere Client, you enable deduplication and compression in the **Configure vSAN** wizard, before you claim any disks for the cluster.



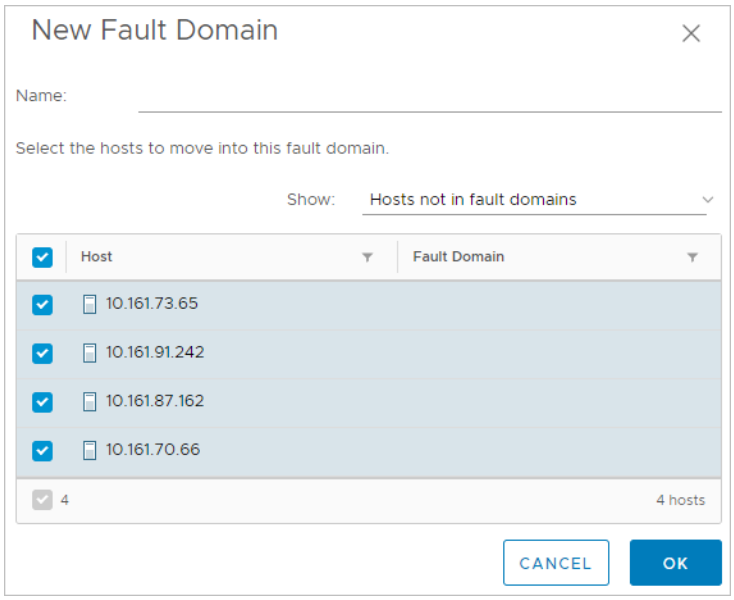
To enable deduplication and compression with the vSAN Management API, you set the `dataEfficiencyConfig` property of the `vsanReconfigSpec` object with an object of type `VsanDataEfficiencyConfig`.

```
if isallFlash:
    print 'Enable deduplication and compression for VSAN'
vsanReconfigSpec.dataEfficiencyConfig = vim.VsanDataEfficiencyConfig(
    compressionEnabled=args.enabledc, dedupEnabled=args.enabledc)

# Enable/disable deduplication and compression
task = vsanClusterSystem.VsanClusterReconfig(cluster, vsanReconfigSpec)
vsanapiutils.WaitForTasks([task], si)
```

Configuring Fault Domains

If your vSAN cluster spans across multiple racks or blade server chassis, you can logically group the hosts in fault domains to protect them against rack or chassis failure. You can separate the vSAN hosts in the same manner that they are physically separated.

In the vSphere Client, you can group hosts in fault domains during the initial configuration of the vSAN cluster or later.



Following is an example of how to configure fault domains by using the vSAN Management API:

```
# Perform these tasks if fault domains are passed as an argument
if args.faultdomains:
    print 'Add fault domains in vsan'
faultDomains = []
# args.faultdomains is a string like f1:host1,host2 f2:host3,host4
for faultdomain in args.faultdomains.split():
    fname, hostnames = faultdomain.split(':')
domainSpec = vim.cluster.VsanFaultDomainSpec(
    name=fname,
```

```
    hosts=[
        host for host in hosts
        if hostProps[host]['name'] in hostnames.split(',')
    ])
faultDomains.append(domainSpec)

# Apply domain specification to vSAN Config
vsanReconfigSpec.faultDomainsSpec = vim.VimClusterVsanFaultDomainsConfigSpec(
    faultDomains=faultDomains)

# Configure fault domains
task = vsanClusterSystem.VsanClusterReconfig(cluster, vsanReconfigSpec)
vsanapiutils.WaitForTasks([task], si)
```

Assigning the vSAN License

You must assign the vSAN license to the vSAN cluster before the 60-day evaluation period expires.

By using the vSAN Management API, you can automate the license assignment on the vSAN clusters in your environment. This way, you can handle license upgrades and renewal more efficiently.

```
if args.vsanlicense:
    print 'Assign VSAN license'
lm = si.content.licenseManager
lam = lm.licenseAssignmentManager
lam.UpdateAssignedLicense(entity=cluster._moId, licenseKey=args.vsanlicense)
```

Configuring Stretched and Two-Host Clusters

4

You can automate the configuration of stretched and two-host vSAN clusters using the vSAN Management API.

Note All examples in this chapter are in Python language.

This chapter includes the following topics:

- Deploying the vSAN Witness Appliance
- Adding the vSAN Witness Appliance to vCenter Server
- Configuring a vSAN Stretched Cluster or Two-Host Cluster

Deploying the vSAN Witness Appliance

Deploying the vSAN Witness Appliance is an alternative to using a physical host to serve as the witness host in your stretched cluster configuration. Unlike a physical host, the appliance does not require a dedicated license or physical disks to store vSAN data.

You can download the vSAN Witness Appliance from the VMware website as a standard OVA file. Then you can install it by using the vSphere Client just like any other OVA file.

You can also upload the vSAN Witness Appliance OVA file through a script. Start with uploading each of the consisting OVA files separately in vCenter Server. First, create a function that uploads a single file in vCenter Server.

```
def uploadFile(srcURL, dstURL, create, lease, minProgress, progressIncrement, vmName=None, log=None):


    '''
    This function will upload vmdk file to vc by using http protocol
    @param srcURL: source url
    @param dstURL: destnate url
    @param create: http request method
    @param lease: HttpNfcLease object
    @param minProgress: file upload progress initial value
    @param progressIncrement: file upload progress update value
    @param vmName: imported virtual machine name
    @param log: log object @return:
    '''
srcData = urllib2.urlopen(srcURL)
```

```
length = int(srcData.headers['content-length'])
ssl._create_default_https_context = ssl._create_unverified_context
protocol, hostPort, reqStr = splitURL(dstURL)
dstHttpConn = createHttpConn(protocol, hostPort)
reqType = create and 'PUT' or 'POST'
dstHttpConn.putrequest(reqType, reqStr)
dstHttpConn.putheader('Content-Length', length)
dstHttpConn.endheaders()
bufSize = 1048768  # 1 MB
total = 0
progress = minProgress
if log:
# If args.log is available, then log to it
log = log.info
else
log = sys.stdout.write
log("%s: %s: Start: srcURL=%s dstURL=%s\n" % (time.asctime(time.localtime()), vmName, srcURL,
                                              dstURL))
log("%s: %s: progress=%d total=%d length=%d\n" % (time.asctime(time.localtime()), vmName, progress,
                                                  total, length))
while True:
    data = srcData.read(bufSize)
if lease.state != vim.HttpNfcLease.State.ready:
    break
dstHttpConn.send(data)
total = total + len(data)
progress = (int)(total * (progressIncrement) / length)
progress += minProgress
lease.Progress(progress)
if len(data) == 0:
    break
log("%s: %s: Finished: srcURL=%s dstURL=%s\n" % (time.asctime(time.localtime()), vmName, srcURL,
                                                 dstURL))
log("%s: %s: progress=%d total=%d length=%d\n" % \ (time.asctime(time.localtime()), vmName,
                                                    progress, total, length))
log("%s: %s: Lease State: %s\n" % \
    (time.asctime(time.localtime()), vmName, lease.state))
if lease.state == vim.HttpNfcLease.State.error:
    raise lease.error
dstHttpConn.getresponse()
return progress
```

Once you have a function for deploying a single file, create another one for uploading multiple files.

```
def uploadFiles(fileItems, lease, ovfURL, vmName=None, log=None):
    '''
    Upload witness vm's vmdk files to vCenter Server by using the HTTP protocol
    @param fileItems: the source vmdks read from ovf file
    @param lease: Represents a lease on a VM or a vApp, which can be used to import or export disks for
    the entity
    @param ovfURL: witness vApp ovf url
    @param vmName: The name of witness vm @param log: @return:
    '''
```

```
uploadUrlMap = {}
for kv in lease.info.deviceUrl:
    uploadUrlMap[kv.importKey] = (kv.key, kv.url)
progress = 5
increment = (int)(90 / len(fileItems))
for file in fileItems:
    ovfDevId = file.deviceId
srcDiskURL = urlparse.urljoin(ovfURL, file.path)
(viDevId, url) = uploadUrlMap[ovfDevId]
if lease.state == vim.HttpNfcLease.State.error:
    raise lease.error
elif lease.state != vim.HttpNfcLease.State.ready:
    raise Exception("%s: file upload aborted, lease state=%s" % (vmName,
                                                                 lease.state))
progress = uploadFile(srcDiskURL, url, file.create, lease, progress, increment,
                      vmName, log)
```

The next step is to define and implement a function that configures the networking settings, the supplied password as a vApp option, and the placement of the appliance on a specific host or resource pool.

The `DeployWitnessOVF` function, defined in the following example, configures the networking settings, the supplied password as a vApp option, and the placement of the appliance on a specific host or resource pool. This function parses only the contents of the OVF and not the entire vSAN Witness Appliance OVA. Password is the only additional argument required by the vSAN Witness Appliance. You must extract the contents of the witness OVA file to a folder containing the OVF and other required files. The OVA file is a `.tar` archive, that you can extract by using a wide variety of tools.

```
"""
Deploying  witness VM to vCenter.
The import process consists of the following steps:
1>Creating the VMs and/or vApps that make up the entity.
2>Uploading the virtual disk contents.
@param ovfURL: ovf source url
@param si: Managed Object ServiceInstance
@param host: HostSystem on which the VM located
@param vmName: VM name
@param dsRef: Datastore on which the VM located
@param vmFolder: Folder to which the VM belong to
@param vmPassword: Password for the VM
@param network: Managed Object Network of the VM
@return: Witness VM entity
"""


def DeployWitnessOVF(ovfURL, si, host, vmName, dsRef, vmFolder, vmPassword=None, network=None):
    rp = host.parent.resourcePool
    params = vim.OvfManager.CreateImportSpecParams()
    params.entityName = vmName
    params.hostSystem = host
    params.diskProvisioning = 'thin'

    f = urllib.urlopen(ovfURL)
```

```
ovfData = f.read()

import xml.etree.ElementTree as ET

params.networkMapping = []
if vmPassword:
    params.propertyMapping = [vim.KeyValue(key='vsan.witness.root.passwd', value=vmPassword)]
ovf_tree = ET.fromstring(ovfData)

for nwt in ovf_tree.findall('NetworkSection/Network'):
    nm = vim.OvfManager.NetworkMapping()
    nm.name = nwt.attrib['name']
    if network != None:
        nm.network = network
    else:
        nm.network = host.parent.network[0]
    params.networkMapping.append(nm)

res = si.content.ovfManager.CreateImportSpec(ovfDescriptor=ovfData,
                                             resourcePool=rp, datastore=dsRef, cisp=params)
if isinstance(res, vim.MethodFault):
    raise res
if res.error and len(res.error) & gt; 0:
    raise res.error[0]
if not res.importSpec:
    raise Exception("CreateImportSpec raised no errors, but importSpec is not set")

lease = rp.ImportVApp(spec=res.importSpec, folder=vmFolder, host=host)
while lease.state == vim.HttpNfcLease.State.initializing:
    time.sleep(1)

if lease.state == vim.HttpNfcLease.State.error:
    raise lease.error

# Upload files
uploadUrlMap = {}
for kv in lease.info.deviceUrl:
    uploadUrlMap[kv.importKey] = (kv.key, kv.url)

progress = 5
increment = (int)(90 / len(res.fileItem))
for file in res.fileItem:
    ovfDevId = file.deviceId
    srcDiskURL = urlparse.urljoin(ovfURL, file.path)
    (viDevId, url) = uploadUrlMap[ovfDevId]
    if lease.state == vim.HttpNfcLease.State.error:
        raise lease.error
    elif lease.state != vim.HttpNfcLease.State.ready:
        raise Exception("%s: file upload aborted, lease state=%s" % \
                        (vmName, lease.state))
    srcData = urllib2.urlopen(srcDiskURL)
    length = int(srcData.headers['content-length'])
    result = urlparse.urlparse(url)
    protocol, hostPort, reqStr = result.scheme, result.netloc, result.path
    if protocol == 'https':
```

```
            dstHttpConn = httplib.HTTPSConnection(hostPort)
        else:
            dstHttpConn = httplib.HTTPConnection(hostPort)
        reqType = file.create and 'PUT' or 'POST'
        dstHttpConn.putrequest(reqType, reqStr)
        dstHttpConn.putheader('Content-Length', length)
        dstHttpConn.endheaders()

        bufSize = 1048768  # 1 MB
        total = 0
        currProgress = progress
        while True:
            data = srcData.read(bufSize)
            if lease.state != vim.HttpNfcLease.State.ready:
                break
            dstHttpConn.send(data)
            total = total + len(data)
            currProgress += (int)(total * (increment) / length)
            progress += minProgress
            lease.Progress(progress)
            if len(data) == 0:
                break
        if lease.state == vim.HttpNfcLease.State.error:
            raise lease.error

        dstHttpConn.getresponse()
        progress = currProgress
    lease.Complete()

    return lease.info.entity
```

Adding the vSAN Witness Appliance to vCenter Server

After you deploy the vSAN Witness Appliance, you must add it to vCenter Server to serve as the witness host in your stretched cluster or two-host configuration. The witness host must not be part of the vSAN cluster.

You can use the vSphere Client to add the vSAN Witness Appliance as a host to vCenter Server.

To add the host programmatically, first create a function that adds the vSAN Witness Appliance as a host in vCenter Server.

```
def AddHost(host, user='root', pwd=None, dcRef=None, si=None, sslThumbprint=None, port=443):
    ''' Add a host to a data center Returns a host system '''


cnxSpec = vim.HostConnectSpec(
    force=True, hostName=host, port=port, userName=user, password=pwd, vmFolder=dcRef.vmFolder)
if sslThumbprint:
    cnxSpec.sslThumbprint = sslThumbprint
hostParent = dcRef.hostFolder
try:
    task = hostParent.AddStandaloneHost(addConnected=True, spec=cnxSpec)
```

```
vsanapiutils.WaitForTasks([task], si)
return getHostSystem(host, dcRef, si)
except vim.SSLVerifyFault as e:
# By catching this exception, you do not need to input the host's thumb print of the SSL certificate
# The following script does this automatically.
cnxSpec.sslThumbprint = e.thumbprint
task = hostParent.AddStandaloneHost(addConnected=True, spec=cnxSpec)
vsanapiutils.WaitForTasks([task], si)
return getHostSystem(host, dcRef, si)
except vim.DuplicateName as e:
raise Exception("AddHost: ESX host %s has already been added to VC." % host)
```

Then add the host by calling the function.

```
print 'Add witness host {} to datacenter {}'.format(witnessVm.name,
                                                    args.witnessdc)
dcRef = searchIndex.FindChild(
    entity=si.content.rootFolder, name=args.witnessdc)
witnessHost = AddHost(
    witnessVm.guest.ipAddress, pwd=args.vmpassword, dcRef=dcRef, si=si)
```

Configuring a vSAN Stretched Cluster or Two-Host Cluster

You can configure a vSAN stretched cluster or two-host cluster.

Following is the process for configuring an existing vSAN cluster as a stretched cluster:

1 Select the hosts that participate in the preferred fault domain.

2 Select the hosts that participate in the secondary fault domain.

3 Select the witness host and configure cache and capacity disks for it.

4 Finish the configuration.

To configure a stretched cluster or two-host setup by using the vSAN Management API, enumerate the hosts in the cluster, select which hosts to add to each fault domain, and then save this data to an array.

```
preferedFd = args.preferdomain
secondaryFd = args.seconddomain
firstFdHosts = []
secondFdHosts = []
for host in hosts:
    if yes('Add host {} to preferred fault domain ? (yes/no)'.format(hostProps[host]['name'])):
    firstFdHosts.append(host)
for host in set(hosts) - set(firstFdHosts):
    if yes('Add host {} to second fault domain ? (yes/no)'.format(hostProps[host]['name'])):
    secondFdHosts.append(host)
faultDomainConfig = vim.VimClusterVSANStretchedClusterFaultDomainConfig(
```

```
    firstFdHosts=firstFdHosts,
    firstFdName=preferedFd,
    secondFdHosts=secondFdHosts,
    secondFdName=secondaryFd)
```

The next step is to define the eligible disks for the witness host.

```
disks = [result.disk for result in witnessHost.configManager.vsanSystem.QueryDisksForVsan() if
         result.state == 'eligible']
diskMapping = None
if disks:
    ssds = [disk for disk in disks if disk.ssd]
nonSsds = [disk for disk in disks if not disk.ssd]
# host with hybrid disks
if len(ssds) > 0 and len(nonSsds) > 0:
    diskMapping = vim.VsanHostDiskMapping(
        ssd=ssds[0],
        nonSsd=nonSsds
    )
# host with all-flash disks,choose the ssd with smaller capacity for cache layer.
if len(ssds) > 0 and len(nonSsds) == 0:
    smallerSize = min([disk.capacity.block * disk.capacity.blockSize for disk in ssds])
smallSsds = []
biggerSsds = []
for ssd in ssds:
    size = ssd.capacity.block * ssd.capacity.blockSize
if size == smallerSize:
    smallSsds.append(ssd)
biggerSsds.append(ssd)
diskMapping = vim.VsanHostDiskMapping(
    ssd=smallSsds[0]
nonSsd = biggerSsds
)
```

After adding the host to the fault domain arrays and defining the eligible disks for the witness host, configure the stretched cluster.

```
print 'start to create stretched cluster'
task = vsanScSystem.VSANVcConvertToStretchedCluster(
    cluster=cluster,
    faultDomainConfig=faultDomainConfig,
    witnessHost=witnessHost, preferredFd=preferedFd,
    diskMapping=diskMapping)
vsanapiutils.WaitForTasks([task], si)
```

Upgrading the vSAN On-Disk Format

5

After upgrading the vSphere environment to a newer version, upgrade the vSAN on-disk format. The latest on-disk format provides the complete feature set of vSAN.

Depending on the size of disk groups, the disk format upgrade can be time-consuming because the disk groups are upgraded one at a time. For each disk group upgrade, all data from each device is evacuated and the disk group is removed from the vSAN cluster. The disk group is then added back to vSAN with the new on-disk format. For more details, see *Administering VMware vSAN* at http://docs.vmware.com.

Note All examples in this chapter are in Python language.

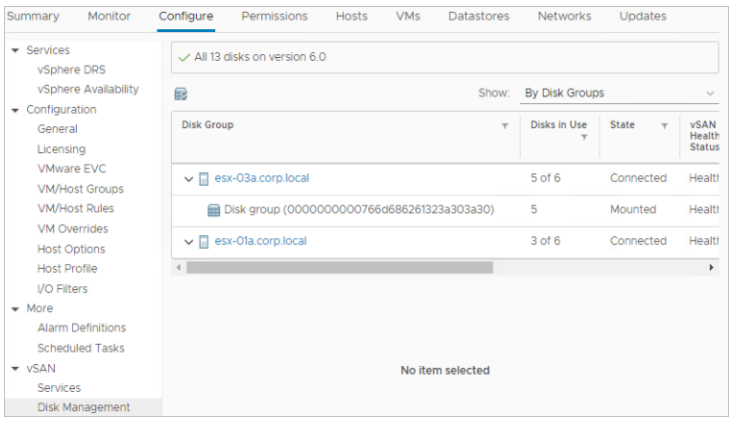
This chapter includes the following topics:

- Determining the Current vSAN On-Disk Format
- Performing the On-Disk Upgrade Preflight Check
- Upgrading with Reduced Redundancy

Determining the Current vSAN On-Disk Format

Before upgrading the vSAN on-disk format, determine the current version of the on-disk format of your vSAN cluster. You must also determine the latest supported format for the ESXi build that the vSAN cluster is running.

In the vSphere Client, you can determine the current on-disk format under **Configure > vSAN > Disk Management** on the vSAN cluster.



To determine the vSAN on-disk format programmatically, first connect to the cluster:

```
cluster = getClusterInstance(args.clusterName, si)
vcMos = vsanapiutils.GetVsanVcMos(si._stub, context=context)
vsanUpgradeSystem = vcMos['vsan-upgrade-systemex']
supportedVersion = vsanUpgradeSystem.RetrieveSupportedVsanFormatVersion(cluster)
print 'The highest vSAN disk format version that given cluster supports is
{}
'.format(supportedVersion)
```

Next, create a function that compares the current on-disk format version to the latest supported version:

```
def hasOlderVersionDisks(hostDiskMappings, supportedVersion):


    for hostDiskMappings in hostDiskMappings:
    for diskMapping in hostDiskMappings:
    if diskMapping.ssd.vsanDiskInfo.formatVersion < supportedVersion:
    return True
for disk in diskMapping.nonSsd:
    if disk.vsanDiskInfo.formatVersion < supportedVersion:
    return True
return False
```

Finally, gather each of the disk group member devices into `diskMappings`, and then pass them into the `hasOlderVersionDisks` function to determine if an upgrade is necessary or not:

```
vsanSystems = CollectMultiple(si.content, cluster.host,
                              ['configManager.vsanSystem']).values()
vsanClusterSystem = vcMos['vsan-cluster-config-system']
diskMappings = CollectMultiple(
    si.content,
    [vsanSystem['configManager.vsanSystem'] for vsanSystem in vsanSystems],
    ['config.storageInfo.diskMapping']).values()
diskMappings = [
    diskMapping['config.storageInfo.diskMapping']
    for diskMapping in diskMappings
]
needsUpgrade = hasOlderVersionDisks(diskMappings, supportedVersion)
```

Performing the On-Disk Upgrade Preflight Check

When you upgrade the vSAN on-disk format using the vSphere Client, a preflight check is performed. Similarly, when you upgrade the on-disk format programmatically, you must perform the preflight check using the following script:

```
print 'Perform VSAN upgrade preflight check'
upgradeSpec = vim.VsanDiskFormatConversionSpec(
    dataEfficiencyConfig=vim.VsanDataEfficiencyConfig(
        compressionEnabled=args.enabledc, deduplicationEnabled=args.enabledc))
```

If many problems exist with the pre-flight check, you must resolve them before you upgrade. You can list the reported problems so that they can be addressed.

```
issues = vsanUpgradeSystem.PerformVsanUpgradePreflightCheckEx(cluster, spec=upgradeSpec).issues
if issues:
    print 'Please fix the issues before upgrade VSAN'
for issue in issues:
    print issue.msg
return
```

Upgrading with Reduced Redundancy

vSAN on-disk format upgrades require the existing VM storage policies to be satisfied during the upgrade process. For example, in a three node cluster, a Failure To Tolerate =1 policy requires three nodes. Bringing a node offline to perform the upgrade can create reduced redundancy.

By default, the upgrade process does not permit reduced redundancy. Attempts to perform an on-disk format upgrade without sufficient spare resources fail. In cases where the vSAN cluster has insufficient resources to satisfy a VM storage policy, such as a three node cluster with FTT=1 using mirroring, you must set a reduced redundancy flag. You can use the Ruby vSphere Remote Console (RVC) to set the reduced redundancy flag.

You can also set the reduced redundancy flag programmatically. You can set the flag when you initiate the upgrade using the following script:

```
print 'call PerformVsanUpgradeEx to upgrade disk versions'
task = vsanUpgradeSystem.PerformVsanUpgradeEx(
    cluster=cluster,
    performObjectUpgrade=args.objupgrade,
    allowReducedRedundancy=args.reduceredundancy)
```

Managing iSCSI Service

6

vSAN iSCSI target service enables hosts and physical workloads that reside outside the vSAN cluster to access the vSAN datastore.

This service enables an iSCSI initiator on a remote host to transport block-level data to an iSCSI target on a storage device in the vSAN cluster. vSAN 6.7 and later releases support Windows Server Failover Clustering (WSFC), so WSFC nodes can access vSAN iSCSI targets.

After configuring the vSAN iSCSI target service, you can discover the vSAN iSCSI targets from a remote host. To discover vSAN iSCSI targets, use the IP address of any host in the vSAN cluster, and the TCP port of the iSCSI target.

Note All examples in this chapter are in Python language.

This chapter includes the following topics:

- Enabling vSAN iSCSI Service
- Creating iSCSI Targets and LUNs
- Disabling iSCSI Service

Enabling vSAN iSCSI Service

To enable iSCSI target service using the vSphere Client, navigate to vSAN cluster and then click the **Configure** tab. Under **vSAN**, select the **Enable vSAN iSCSI target service** check box.

Here is an example of how to enable iSCSI target service using the vSAN Management API. This example is based on the code in the `vsaniscsisamples.py` sample file located under the `samplecode` directory.

```
# Enable iSCSI service using the vSAN Cluster Reconfiguration API on vCenter, and
# The config port is set to 3260 by default. However, this can be customized.
def EnableIscsi(vsanStoragePolicy, si, context, apiVersion, vcMos, cluster):
    defaultVsanConfigSpec = vim.cluster.VsanIscsiTargetServiceDefaultConfigSpec(
        networkInterface="vmk0",
        port=2300)
    vitEnableSpec = vim.cluster.VsanIscsiTargetServiceSpec(
        homeObjectStoragePolicy=vsanStoragePolicy,
        defaultConfig=defaultVsanConfigSpec,
        enabled=True)
```

```
    vccs = vcMos['vsan-cluster-config-system']
    clusterReconfigSpec = vim.vsan.ReconfigSpec(iscsiSpec=vitEnableSpec)
    vitEnableVsanTask = vccs.ReconfigureEx(cluster, clusterReconfigSpec)
    vitEnableVcTask = vsanapiutils.ConvertVsanTaskToVcTask(
        vitEnableVsanTask, si._stub)
    vsanapiutils.WaitForTasks([vitEnableVcTask], si)
    print('Enable vSAN iSCSI service task finished with status: %s' %
          vitEnableVcTask.info.state)
```

Creating iSCSI Targets and LUNs

To create an iSCSI target and its associated LUN using the vSphere Client, navigate to vSAN cluster and then click the **Configure** tab. Under **vSAN**, click **iSCSI Targets** and then click the **Add a new iSCSI target** icon.

Here is an example of how to create iSCSI targets and LUNs using the vSAN Management API. This example is based on the code in the `vsaniscsisamples.py` sample file located under the `samplecode` directory.

```
# Creating vSAN iSCSI targets and an associated LUN of 1GB size.
def CreateIscsiTargetAndLun(cluster, si):
    targetAlias = "sampleTarget"
    targetSpec = vim.cluster.VsanIscsiTargetSpec(
        alias=targetAlias,
        iqn='iqn.2015-08.com.vmware:vit.target1')
    vsanTask = vits.AddIscsiTarget(cluster, targetSpec)
    vcTask = vsanapiutils.ConvertVsanTaskToVcTask(vsanTask, si._stub)
    vsanapiutils.WaitForTasks([vcTask], si)
    print('Create vSAN iSCSI target task finished with status: %s' %
          vcTask.info.state)

    lunSize = 1 * 1024 * 1024 * 1024  # 1GB
    lunSpec = vim.cluster.VsanIscsiLUNSpec(
        lunId=0,
        lunSize=lunSize,
        storagePolicy=vsanStoragePolicy)
    vsanTask = vits.AddIscsiLUN(cluster, targetAlias, lunSpec)
    vcTask = vsanapiutils.ConvertVsanTaskToVcTask(vsanTask, si._stub)
    vsanapiutils.WaitForTasks([vcTask], si)
    print('Create vSAN iSCSI LUN task finished with status: %s' %
          cTask.info.state)
```

Disabling iSCSI Service

To disable iSCSI target service using the vSphere Client, navigate to vSAN cluster and then click the **Configure** tab. Under **vSAN**, deselect the **Enable vSAN iSCSI target service** check box.

Here is an example of how to disable iSCSI target service using the vSAN Management API. This example is based on the code in the `vsaniscsisamples.py` sample file located under the `samplecode` directory.

```
# Disable iSCSI service through vSAN iSCSI API on vCenter.
def DisableIscsi(vitDisableSpec, si, context, apiVersion, vcMos):
    vitDisableSpec = vim.cluster.VsanIscsiTargetServiceSpec(enabled=False)
    clusterReconfigSpec = vim.vsan.ReconfigSpec(iscsiSpec=vitDisableSpec)
    vccs = vcMos['vsan-cluster-config-system']
    vitDisableVsanTask = vccs.ReconfigureEx(cluster, clusterReconfigSpec)
    vitDisableVcTask = vsanapiutils.ConvertVsanTaskToVcTask(
        vitDisableVsanTask, si._stub)
    vsanapiutils.WaitForTasks([vitDisableVcTask], si)
    print('Disable vSAN iSCSI service task finished with status: %s' %
          vitDisableVcTask.info.state)
```

Monitoring vSAN

7

You can obtain statistical data about various aspects of vSAN performance, as generated and maintained by the vSAN performance service of the cluster. You can also view vSAN cluster health information.

Note All examples in this chapter are in Python language.

This chapter includes the following topics:

- Viewing vSAN Health Check Status
- Monitoring vSAN Performance

Viewing vSAN Health Check Status

To view the vSAN health using the vSphere Client, navigate to the vSAN cluster, click the **Monitor** tab, and then click **vSAN**. Select **Health** to review the vSAN health check categories.

Here is an example of how to the vSAN health using the vSAN Management API.

```
# Caching vSAN cluster health summary at vCenter.
def GetClusterHealthSummary(cluster, vcMos):
    fetchFromCache = True
    vhs = vcMos['vsan-cluster-health-system']
    healthSummary = vhs.QueryClusterHealthSummary(
        cluster=cluster, includeObjUuids=True, fetchFromCache=fetchFromCache)
    clusterStatus = healthSummary.clusterStatus
    return clusterStatus
```

Monitoring vSAN Performance

You can use vSAN the performance service to monitor the performance of your vSAN cluster, and investigate potential problems.

The performance service collects and analyzes performance statistics and displays the data. You can use the performance charts to manage your workload and determine the root cause of problems.

Enabling the Performance Service

The performance service is disabled by default upon the creation of the vSAN cluster. You can enable the performance service after you configure the vSAN cluster to monitor the performance of the cluster, the participating hosts, disks, and VMs.

In the vSphere Client, you can enable the performance service from Health and Performance settings on the cluster.

To enable the vSAN performance service using the vSphere Client, navigate to the vSAN cluster, click the **Monitor** tab, and then click **vSAN**. Click **Performance** and then click **Enable**.

Following is an example of how to enable the performance service by using the vSAN Management API:

```
print 'Enable perf service on this cluster'
# Apply the Performance Service to the VSAN config
vsanPerfSystem = vcMos['vsan-performance-manager']

# Apply the config update
task = vsanPerfSystem.CreateStatsObjectTask(cluster)
vsanapiutils.WaitForTasks([task], si)
```

Viewing vSAN Cluster Performance

To view the vSAN cluster performance, using the vSphere Client, navigate to the vSAN cluster, click the **Monitor** tab, and then click **Performance**. Select **vSAN - Virtual Machine Consumption** with a time range for your query.

Here is an example of how to view the vSAN cluster performance using the vSAN Management API.

```
# Get vSAN cluster performance
def getClusterPerformance(cluster, vsanPerfSystem):
    spec = vim.cluster.VsanPerfQuerySpec()
    spec.entityRefId = "cluster-domclient:5287a00e-e90d-dbdc-1909-bf952fdaad3a"
    endTime = datetime.datetime.utcnow()
    startTime = endTime - datetime.timedelta(hours=1)
    spec.startTime = startTime
    spec.endTime = endTime
    result = vsanPerfSystem.VsanPerfQueryPerf(querySpecs=[spec], cluster=clusterMoID)
    return result
```

Viewing vSAN Host Performance

To view the vSAN host performance, using the vSphere Client, navigate to the vSAN cluster and select a host. Click the **Monitor** tab and then click **Performance**. Select **vSAN - Virtual Machine Consumption** with a time range for your query.

The sample code for getting vSAN host performance is similar to the sample code for getting vSAN cluster performance above, except for the following:

- Instead of `spec.entityRefId`, specify `host-domclient`.
- In place of `cluster-domclient`, specify `host-UUID`.

Viewing vSAN VM Performance

To view the vSAN VM performance, using the vSphere Client, navigate to the vSAN cluster and select a VM. Click the **Monitor** tab, and then click **Performance**. Select **vSAN - Virtual Machine Consumption** with a time range for your query. Now, select **vSAN - Virtual Disk** with a time range for your query.

The sample code for getting vSAN VM performance is similar to the sample code for getting vSAN cluster performance above, except for the following:

- Instead of `spec.entityRefId`, specify `virtual-machine`.
- In place of `cluster-domclient`, specify `VM-UUID`.