vSAN SDKs Programming Guide

VMware vSAN 6.6

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

Contents

1 Introduction to the vSAN Managmenet 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 Clusters for vSAN 11
Enabling vSAN on a Cluster 12
Enabling Deduplication and Compression on All-Flash Clusters 13
Configuring VMkernel Networking for vSAN 14
Claiming and Managing Disks 16
Configuring Fault Domains 18
Enabling the Performance Service 19
Assigning the vSAN License 19

4 Configuring a Stretched Cluster and Two Node 20

Deploying the vSAN Witness Appliance 20
Adding the vSAN Witness Appliance to vCenter Server 23
Configuring a vSAN Cluster as a Stretched Cluster or Two Node 24

5 vSAN On-Disk Format Upgrade 26

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

Introduction to the vSAN Managmenet SDKs

1

The vSAN Managment 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 client applications performing the following tasks:

- Configure a vSAN cluster. You can configure all aspects of a vSAN cluster, such as set VMkernel networking, claim disks, configure fault domains, enable deduplication and compression of all flash clusters, and assign the vSAN license.
- Confiure a vSAN stretched cluster. You can deploy the vSAN Witness Appliance and configure an existing vSAN cluster as a 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 delivered in five programming languages that you can use to access the vSAN Management API and develop client applications for managing vSAN clusters.

Using the vSAN Management SDKs

2

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

This section 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 6.6 depends on the vSphere Web Services SDK 6.0. You use the vSphere Web Services SDK for logging in to vCenter Server and for retrieving vCenter Server managed objects.

API Reference

The vSAN Management SDK for Java packs the API reference for the vSAN Management API, which you can find under the `docs` directory.

WSDL Files

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.

Running the Sample Applications

The vSAN Management SDK for Java includes sample applications, `build` and `run` scripts, as well as 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.

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

Before you run 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
```

To build the sample applications, run the `$PYTHON build.py` command.

Finally use the `run` script to run the sample applications under Windows or Linux:

```
./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 6.6 depends on the vSphere Web Services SDK 6.0. You use the vSphere Web Services SDK for logging in to vCenter Server and for retrieving vCenter Server managed objects.

API Reference

The vSAN Management SDK for .NET packs the API reference for the vSAN Management API, which you can find under the `docs` directory.

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
- `WseWsdl3.exe`. A WSDL to proxy class tool
- `Microsoft.Web.Services3.dll`
- 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 6.6 depends on pyVmomi 6.5 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.

API Reference

The vSAN Management SDK for Python packs the API reference for the vSAN Management API, which you can find under the `docs` directory.

Python Bindings

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

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

Running the Sample Applications

The vSAN Management SDK for Python provide 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 runt 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 Java clients against the vSAN Management API. The vSAN Management SDK for Perl 6.6 depends on viperl 6.0 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.

API Reference

The vSAN Management SDK for Perl packs the API reference for the vSAN Management API, which you can find under the `docs` directory.

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 those two 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 utility library for retrieving vSAN managed objects.

Run the following sample to test the vCenter Server side API:

```
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 Python clients against the vSAN Management API. The
vSAN Management SDK for Ruby 6.6 depends on RbVmomi 2.3, 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.

API Reference

The vSAN Management SDK for Ruby packs the API reference for the vSAN Management API, which you can find under the `docs` directory.

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 rung 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.

3 Setting Up a vSAN Cluster

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 steps for setting up a vSAN cluster by using the vSAN Management API are similar to the steps that you go through when using the vSphere Web Client.

This section includes the following topics:

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

Connecting to vCenter Server and Selecting Clusters for vSAN

Before you configure a vSAN cluster by using the vSAN Management API, first you must establish a secure connection with vCenter Server and filter the clusters where you want to enable vSAN

In the below example, first a secure connection is established with vCenter Server through username and password authentication. Then the `getClusterInstance` function is called with the cluster name passed 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 vCenter Sever upon exit
```

```
       atexit.register(Disconnect, si)

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

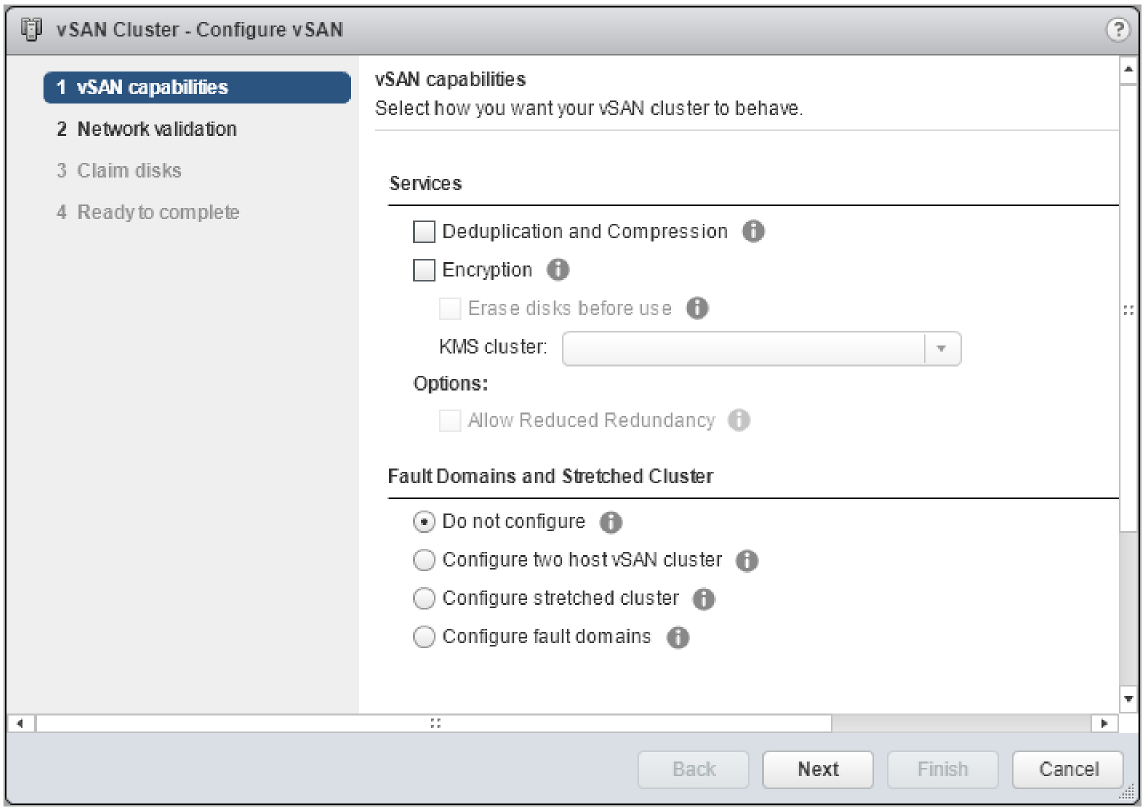
Once you have established a secure connection with vCenter Server and identified the cluster, you have to connect to that cluster. You can reuse the `getClusterInstance` function across your client application to connect to clusters where you want to configure vSAN.

```
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 return None
```

Enabling vSAN on a Cluster

Once you have filtered the clusters that you want to configure for vSAN, the next step is to enable vSAN on these clusters.

In the vSphere Web Client, you use the **Configure vSAN** wizard to configure individual clusters for vSAN. You must go through the wizard steps separately for every cluster. When you start the wizard on a cluster, the first step is to enable vSAN.


vSAN Cluster - Configure vSAN

1 vSAN capabilities
2 Network validation
3 Claim disks
4 Ready to complete

vSAN capabilities
Select how you want your vSAN cluster to behave.

Services
☐ Deduplication and Compression
☐ Encryption
☐ Erase disks before use
KMS cluster:
Options:
☐ Allow Reduced Redundancy

Fault Domains and Stretched Cluster
[✓ Do not configure]
Configure two host vSAN cluster
Configure stretched cluster
Configure fault domains

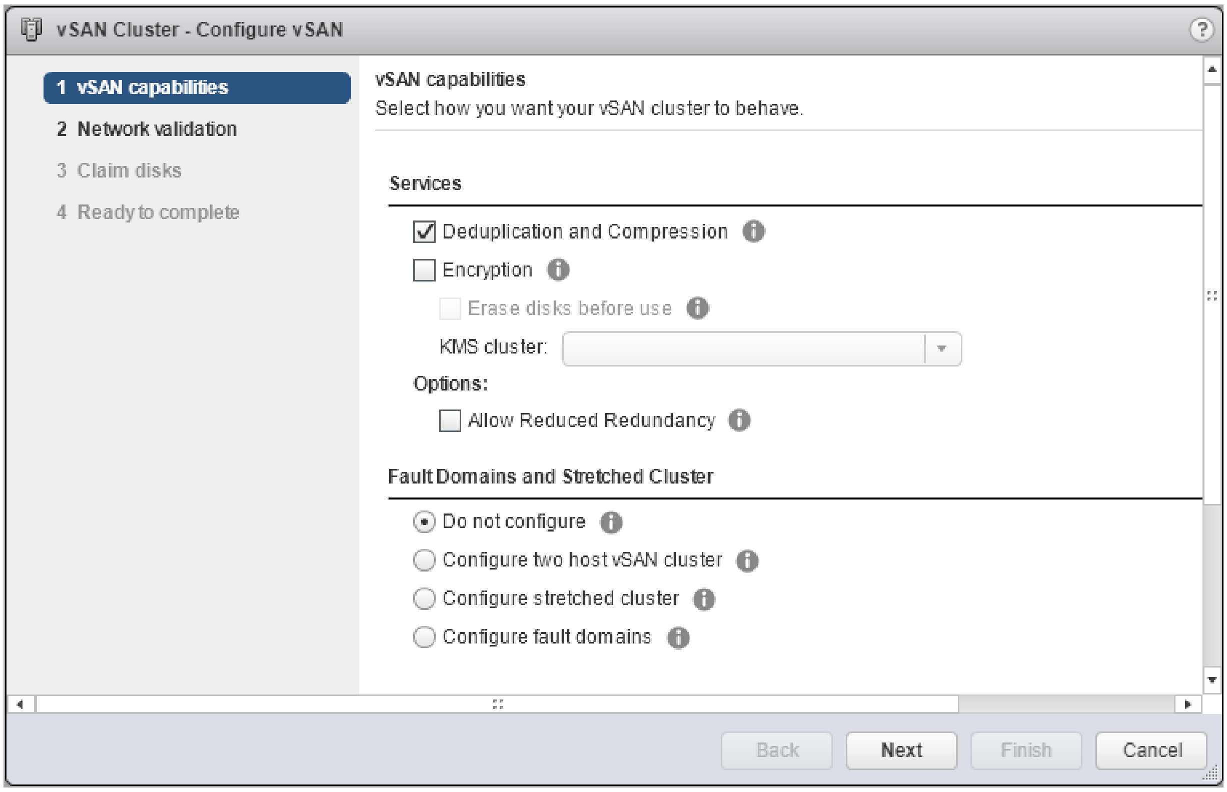
To enable vSAN in your vSAN Management API client applications, you build an object of type `VimVsanReconfigSpec` by passing a `VsanClusterConfigInfo` parameter with the property `enable` set to `true`.

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

Enabling Deduplication and Compression on All-Flash Clusters

For all-flash clusters, enable deduplication and compression when creating the vSAN cluster. If you create the cluster without deduplication and compression and decide to enable them later on, the process triggers rolling upgrade that is time consuming and might require a reduced availability.

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


vSAN Cluster - Configure vSAN

1 vSAN capabilities
2 Network validation
3 Claim disks
4 Ready to complete

vSAN capabilities
Select how you want your vSAN cluster to behave.

Services

- [✓] Deduplication and Compression
- [ ] Encryption
- [ ] Erase disks before use

KMS cluster:

Options:

- [ ] Allow Reduced Redundancy

Fault Domains and Stretched Cluster

- [●] Do not configure
- ○ Configure two host vSAN cluster
- ○ Configure stretched cluster
- ○ Configure fault domains

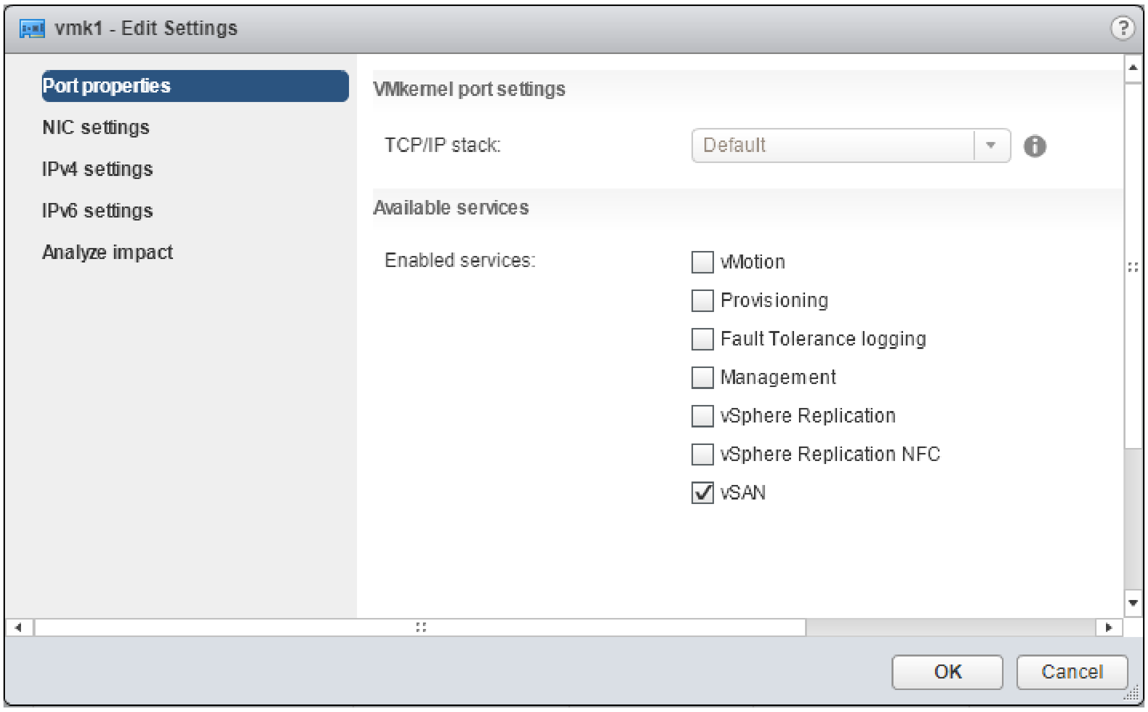
To enable deduplication and compression with the vSAN Management API, you set the `dataEfficiencyConfig` property of the `vsanReconfigSpec` object with an object of type `VsanDataEfficiencyConfig`.

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

# Enabled/Disable Deduplication and Compression
task = vsanClusterSystem.VsanClusterReconfig(cluster, vsanReconfigSpec)
vsanapiutils.WaitForTasks([task], si)
```

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


vmk1 - Edit Settings

- Port properties
- NIC settings
- IPv4 settings
- IPv6 settings
- Analyze impact

VMkernel port settings

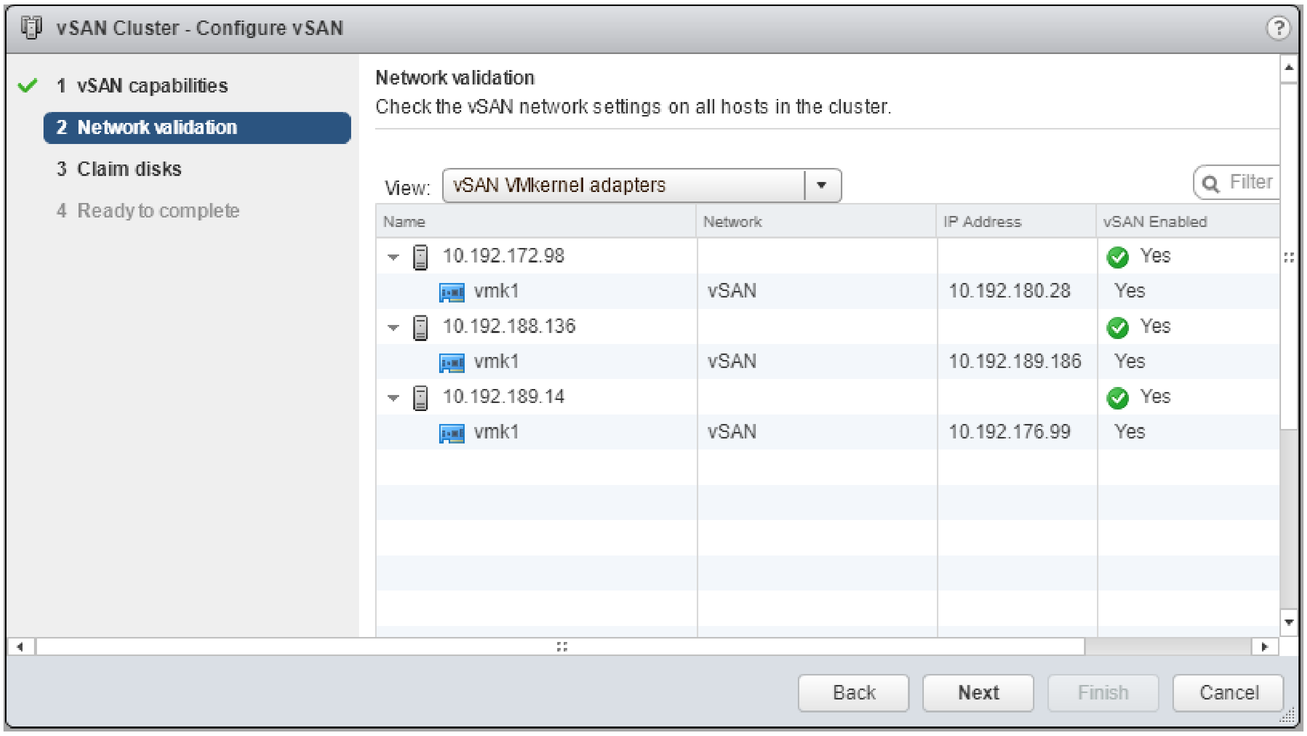
TCP/IP stack: Default

Available services

Enabled services:

- [ ] vMotion
- [ ] Provisioning
- [ ] Fault Tolerance logging
- [ ] Management
- [ ] vSphere Replication
- [ ] vSphere Replication NFC
- [✓] vSAN

When you configure vSAN on a cluster, the Configure vSAN wizard validates the networking configuration on the hosts. In case some of the hosts is missing a VMkernel network adapter enabled for vSAN, you must suspend the configuration of the cluster, and set up the host networking for the vSAN traffic.


vSAN Cluster - Configure vSAN

- ✓ 1 vSAN capabilities
- 2 Network validation
- 3 Claim disks
- 4 Ready to complete

Network validation
Check the vSAN network settings on all hosts in the cluster.

View: vSAN VMkernel adapters

Name	Network	IP Address	vSAN Enabled
10.192.172.98			✓ Yes
vmk1	vSAN	10.192.180.28	Yes
10.192.188.136			✓ Yes
vmk1	vSAN	10.192.189.186	Yes
10.192.189.14			✓ Yes
vmk1	vSAN	10.192.176.99	Yes

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

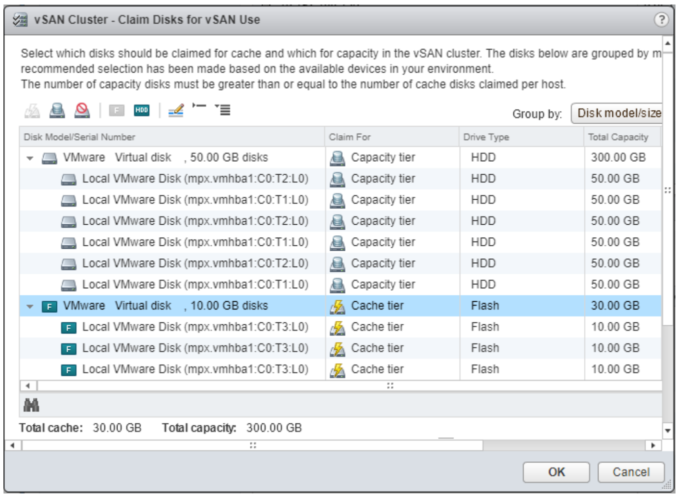
```
# Update the configuration spec for VMkernet xetworking
# Enumerate the selected VMkernel adapter for each host, and add it to the list of tasks
        for host in hosts:
             print 'Enable vSAN traffic on host {} with {}'.format(hostProps[host]['name'], 
args.vmknic)
             task =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

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 Web 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 all flash configurations, the wizard assigns the smaller devices as cache tiers and the larger devices as capacity tiers. In a hybrid configuration, flash devices are assigned as cache tiers and HDD devices as capacity tiers.


vSAN Cluster - Claim Disks for vSAN Use

Select which disks should be claimed for cache and which for capacity in the vSAN cluster. The disks below are grouped by m
recommended selection has been made based on the available devices in your environment.
The number of capacity disks must be greater than or equal to the number of cache disks claimed per host.

Group by: Disk model/size

Disk Model/Serial Number	Claim For	Drive Type	Total Capacity
VMware Virtual disk , 50.00 GB disks	Capacity tier	HDD	300.00 GB
Local VMware Disk (mpx.vmhba1:C0:T2:L0)	Capacity tier	HDD	50.00 GB
Local VMware Disk (mpx.vmhba1:C0:T1:L0)	Capacity tier	HDD	50.00 GB
Local VMware Disk (mpx.vmhba1:C0:T2:L0)	Capacity tier	HDD	50.00 GB
Local VMware Disk (mpx.vmhba1:C0:T1:L0)	Capacity tier	HDD	50.00 GB
Local VMware Disk (mpx.vmhba1:C0:T2:L0)	Capacity tier	HDD	50.00 GB
Local VMware Disk (mpx.vmhba1:C0:T1:L0)	Capacity tier	HDD	50.00 GB
VMware Virtual disk , 10.00 GB disks	Cache tier	Flash	30.00 GB
Local VMware Disk (mpx.vmhba1:C0:T3:L0)	Cache tier	Flash	10.00 GB
Local VMware Disk (mpx.vmhba1:C0:T3:L0)	Cache tier	Flash	10.00 GB
Local VMware Disk (mpx.vmhba1:C0:T3:L0)	Cache tier	Flash	10.00 GB

Total cache: 30.00 GB Total capacity: 300.00 GB

In your client applications, first you can query for eligible disks and optionally clear any existing vSAN partitions on the non-eligible ones if needed.

```
# 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 the admin as to 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())
```

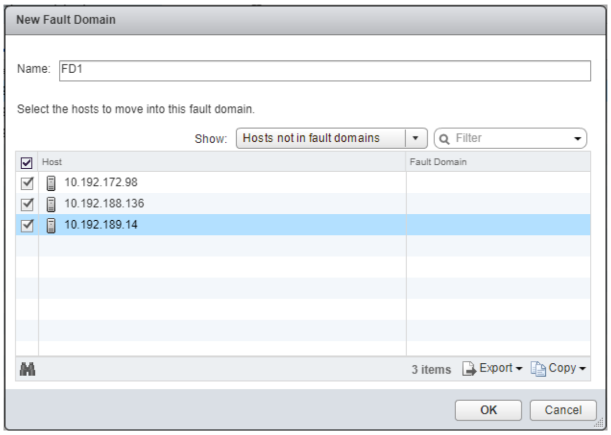
The second step is to differentiate between the smaller and the larger devices for an all flash configuration and between the flash and the HDD devices for a hybrid configuration. Then you can claim the disks respectively for cache and capacity tiers.

```
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)
```

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 similarly to the way they are physically separated.

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



Name: FD1

Select the hosts to move into this fault domain.

Show: Hosts not in fault domains

Host	Fault Domain
✓ 10.192.172.98	
✓ 10.192.188.136	
✓ 10.192.189.14	

3 items

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

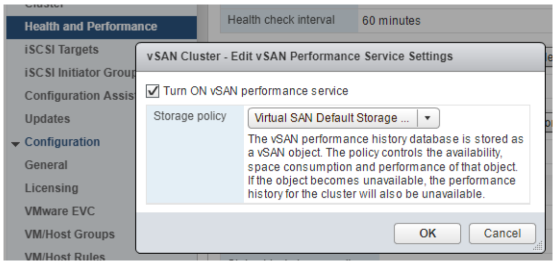
```
 # Perform these tasks if Fault Domains are passed as an argument
 if args.faultdomains:
   print 'Add fault domains in vsan'
   faultDomains = []
   #args.faultdomains is a string like f1:host1,host2 f2:host3,host4
   for faultdomain in args.faultdomains.split():
      fname, hostnames = faultdomain.split(':')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

 # Apply the Domain Specification to the vSAN Config
   vsanReconfigSpec.faultDomainsSpec = vim.VimClusterVsanFaultDomainsConfigSpec(
      faultDomains=faultDomains
   )
```

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

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 Web Client, you can enable the performance service from Health and Performance settings on the cluster:


vSAN Cluster - Edit vSAN Performance Service Settings

[✓] Turn ON vSAN performance service

Storage policy: Virtual SAN Default Storage ...

The vSAN performance history database is stored as a vSAN object. The policy controls the availability, space consumption and performance of that object. If the object becomes unavailable, the performance history for the cluster will also be unavailable.

Health check interval: 60 minutes

Here is 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)
```

Assigning the vSAN License

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

In the vSphere Web Client, you assign a license to the vSAN cluster manually, through the Configure vSAN wizard or through the **Licensing** option under Administration.

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 a Stretched Cluster and Two Node 4

You can automate the configuration of a vSAN stretched cluster and two node by using the vSAN Management API.

This section includes the following topics:

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

Deploying the vSAN Witness Appliance

Deploying the vSAN Witness Appliance is an alternative to using a physical host to serve as the witness node 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 Web site as a standard OVA file. Then you can install it by using the vSphere Web 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 implement the `DeployWitnessOVF` 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 vSAN Witness Appliance only requires a password as an additional argument that you need to set.

The `DeployWitnessOVF` function parses the contents of the OVF, but cannot parse the entire vSAN Witness Appliance OVA. 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.

```
print 'Start to add virtual witness host'
  '''
  Steps to add the Witness Appliance, rather than a dedicating a physical ESXi host as the witness
node.
  1) Deploy the witness VM specifying host, storage, and network for the Witness VM
  2) Get the witness VM and add it to the data center as a witness host
  '''

  dc = searchIndex.FindChild(entity = si.content.rootFolder, name = args.datacenter)
  #specify the host for the witness VM
  hostSystem = getHostSystem(args.vmhost, dc, si)
  #specify the storage for the witness VM
  ds = searchIndex.FindChild(entity = dc.datastoreFolder, name = args.datastore)
  #specify the network for the witness VM
  if args.network:
    network = [net for net in dc.networkFolder.childEntity
        if net.name == args.network][0]
  else:
    network = dc.networkFolder.childEntity[0]
  witnessVm = DeployWitnessOVF(args.ovfurl, si, hostSystem, args.name, ds, dc.vmFolder,
vmPassword=args.vmpassword, network=network)

  task = witnessVm.PowerOn()
  vsanapiutils.WaitForTasks([task], si)

# Wait for vm to power on and become available
  beginTime = time.time()
  while True:
    try:
    # Connect to the Witness Host
      SmartConnect(host=witnessVm.guest.ipAddress,
      user='root',
      pwd=args.vmpassword,
      port=443,
      sslContext=context)
  except:
    time.sleep(10)
    timeWaiting = time.time() ---- beginTime
    if timeWaiting > (15 * 60):
      raise Exception("Timed out waiting (>15min) for VM to up!")
  else:
    break
```

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 node in your stretched cluster or two node configuration. The witness node must not be part of the vSAN cluster.

You can use the vSphere Web Client to add the vSAN Witness Appliance as a host to vCenter Server. The vSphere Web Client is a preferred interface over the legacy vSphere Client, because you can assign the vSphere Web Client license as part of the process.

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 don not need to input the host's thumbprint of the SSL certificate,
the logic below
  #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 call the function to add the host.

```
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 Cluster as a Stretched Cluster or Two Node

You can configure a stretched cluster or two node when you create the vSAN cluster or after that.

Here's the process for configuring an already existing vSAN cluster as stretched cluster:

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

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

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

4 Complete the configuration.

To configure a stretched cluster or two node setup by using the vSAN Management API, start by enumerating the hosts in the cluster, selecting which hosts will go to the two domains, and 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
      )
```

Once you have put the hosts into fault domain arrays and defined the eligible disks for the witness host, you can 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)
```

vSAN On-Disk Format Upgrade

5

After you upgrade your 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 the *Administering VMware vSAN* documentation at http://docs.vmware.com.

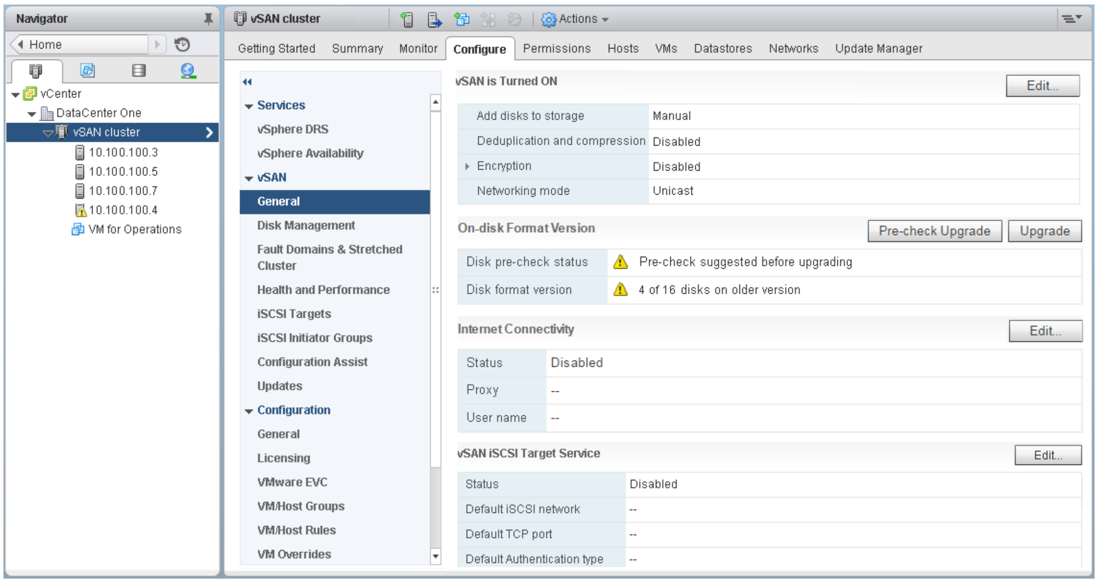
This section includes the following topics:

- 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 you upgrade 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 Web Client, you can determine the current on-disk format under **Configure > General** on the vSAN cluster.



vSAN is Turned ON

Add disks to storage	Manual
Deduplication and compression	Disabled
Encryption	Disabled
Networking mode	Unicast

On-disk Format Version

Disk pre-check status	⚠ Pre-check suggested before upgrading
Disk format version	⚠ 4 of 16 disks on older version

Internet Connectivity

Status	Disabled
Proxy	--
User name	--

vSAN iSCSI Target Service

Status	Disabled
Default iSCSI network	--
Default TCP port	--
Default Authentication type	--

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 Virtual SAN 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`, 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 through the vSphere Web Client, a preflight check-in is performed. When you upgrade the on-disk format programmatically, you must also perform the pre-flight check.

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

If many issues exist with the pre-flight check, you must resolve them before you upgrade. You can list the reported issues 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
      eturn
```

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 would 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 cannot do this through the vSphere Web Client, you can use the Ruby vSphere Remote Console (RVC).

You can also set the reduced redundancy flag programmatically. You can set the flag as part of initiating the upgrade.

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