VMware® Virtual SAN™ 6.2

Virtual SAN Management API Cookbook for Python

V0.1 – JULY 2016

Table of Contents

Introduction

Typically, Virtual SAN management is performed through the vSphere® Web Client. Tasks can include those such as initial configuration, ongoing maintenance, and reporting of capacity, performance, or health of Virtual SAN. The Web Client provides comprehensive element management of each component of a Virtual SAN cluster.

While most element management is easily accomplished with the Web Client user interface, performing many repeatable tasks are a manual process. Some aspects of Virtual SAN management are automated, such as disk claiming, periodic health checks, as well as error and other reporting. These automated tasks are specific to each individual Virtual SAN cluster, and often have to be repeated many times when managing multiple independent Virtual SAN clusters.

Consistency and repeatability, is a challenge when performing tasks manually. It is quite common to leverage tools such as an Application Programming Interface (API) along with code to execute tasks in a consistent and repeatable fashion.

Expectations

This document is intended to assist you with understanding types of things that can be managed programmatically through the Virtual SAN Management API and SDK.

It is neither comprehensive in showing all possible actions nor prescriptive in showing the only way to accomplish these tasks. This document will focus on the use of Python, but be aware other languages may be chosen, which will not be covered in this document.

Throughout the document we will alternate showing what types of tasks can be done through the standard web client user interface, and then how to achieve the same result through Python. None of the included code samples are supported by VMware, and are merely representative of possible ways to accomplish tasks.

Virtual SAN Management API

Virtual SAN 6.2 introduced a new Virtual SAN Management API that which extends upon the current vSphere API. This API is exposed by both vCenter Server managing Virtual SAN, as well as VMware ESXi hosts. Setup and all configuration of aspects of Virtual SAN, as well as runtime state, is available by leveraging the Virtual SAN Management API.

There are a variety of vSphere Managed Objects exposed by the Virtual SAN Management API that provide functionality specific to vCenter Server, ESXi, or both. These Managed Objects are:

Managed Object	Function	Available
VsanVcDiskManagementSystem	VSAN Cluster configuration and query APIs for disks	vCenter
VsanVcStretchedClusterSystem	VSAN Stretched Cluster related configuration and query APIs	vCenter
VsanVcClusterConfigSystem	VSAN Cluster configuration setting and query APIs	vCenter
VsanVcClusterHealthSystem	VSAN Cluster health related configuration and query APIs	vCenter
VsanSpaceReportSystem	VSAN Cluster space usage related query APIs	vCenter
VsanPerformanceManager	VSAN Cluster performance related configuration and query APIs	vCenter & ESXi
VsanObjectSystem	VSAN Cluster setting APIs for object status query and storage policy	vCenter & ESXi
HostVsanSystem	Host level VSAN related configuration and query APIs	ESXi
HostVsanHealthSystem	Host level VSAN Health related configuration and query APIs	ESXi
VsanUpgradeSystem	Used to perform and monitor VSAN on-disk format upgrades. *VsanUpgradeSystemEx deprecates VsanUpgradeSystem*	vCenter
VsanUpgradeSystemEx		vCenter

Table 1 – Managed Objects presented by the Virtual SAN Management API

Virtual SAN SDKs

Software Development Kits (SDK) have also been provided with the release of Virtual SAN 6.2 for several popular programming languages. By providing SDKs in multiple languages, VMware has made it easier to use the Virtual SAN API. Some of the languages that have SDKs for the Virtual SAN API include Python, Ruby, Java, C#, and Perl. They each include respective language bindings to the vSphere Management SDK, as well as documentation that details the usage of each API.

These are all extensions of the vSphere API for their respective programming language. While these are languages that are supported today, additional languages may be supported in the future.

Getting Started with the Virtual SAN Management SDK

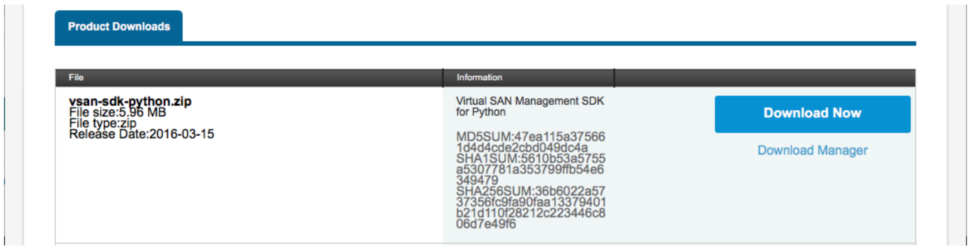
To get started, the Virtual SAN Management SDK, one of the previously mentioned packages needs to be downloaded from the VMware Developer site. The starting page for SDKs can be found here:

http://developercenter.vmware.com/sdks

This URL is a common launching point for SDKs for different VMware products including Virtual SAN. The Virtual SAN Management SDKs can be found in the **Storage** section. Each of the different Virtual SAN Management SDK links leads to dedicated content for that SDK including documentation and reference material and a download link.

While the Virtual SAN Management SDK content is publicly available, a **My VMware** account is required. Anyone can register for a **My VMware** account for free. Existing customers may already have accounts if they have performed tasks such as license management, downloaded software, or worked with VMware support.

From the download page, download the appropriate Virtual SAN Management SDK for the language you wish to use.



File	Information
vsan-sdk-python.zip File size:5.96 MB File type:zip Release Date:2016-03-15	Virtual SAN Management SDK for Python MD5SUM:47ea115a37566 1d4d4cde2cbd049dc4a SHA1SUM:5610b53a5755 a5307781a353799ffb54e6 349479 SHA256SUM:36b6022a57 37356fc9fa90faa13379401 b21d110f28212c223446c8 06d7e49f6

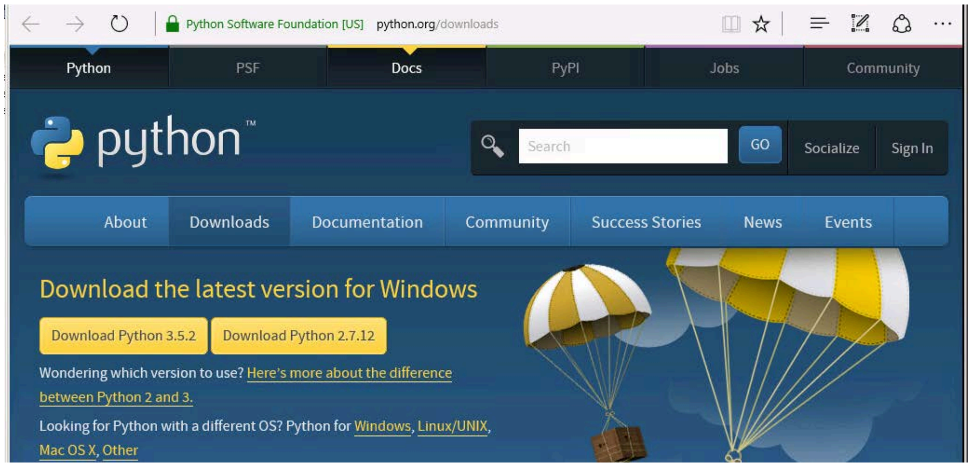
The Virtual SAN Management SDKs are made available as a compressed .zip file. To use the SDK, the contents of the .zip file will need to be extracted. While some Operating Systems provide native support for extracting contents from .zip files, others do not.

Using the Virtual SAN Management SDK for Python

To be able to use the Virtual SAN Management SDK for Python, it is important to have Python installed on the system that will be used to execute scripts that use the SDK.

Python can be easily installed on Windows, Linux, and Mac operating systems. This provides flexibility for developers to use the platform of their choice, while providing portability of scripts across these very different operating systems.

For the purpose of this document, Python for Windows 10 will be used. Python is available for free from Python.org.



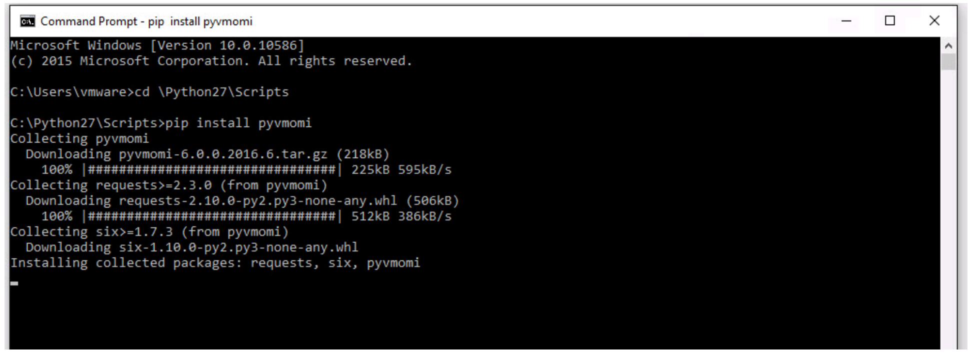
Download the latest version for Windows

Wondering which version to use? Here's more about the difference between Python 2 and 3.

Looking for Python with a different OS? Python for Windows, Linux/UNIX, Mac OS X, Other

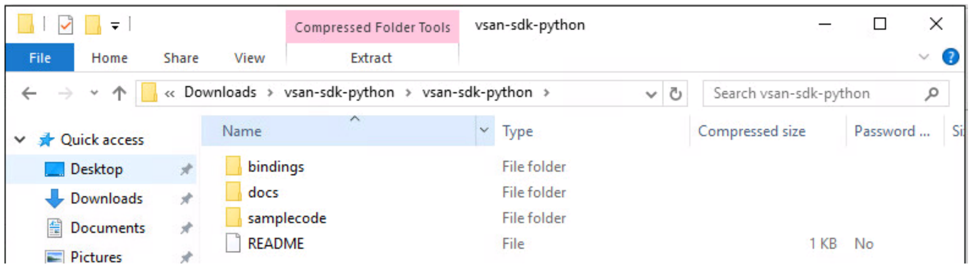
Python 2.7.12 was picked for the purpose of this paper. Python defaults to installing to C:\Python27. After Python has been downloaded and installed, it is important to add pyVmomi. PyVmomi is the Python SDK for the VMware vSphere API, allowing developers to access ESXi and vCenter using Python. The Virtual SAN Management API is built as an extension of the vSphere API therefore, pyVmomi is also used to allow Python access to the Virtual SAN Management API.

An easy way to add pyVmomi to the installation is through the use of pip. Pip is the Python Package Manager. To install pyVmomi, simply type **"pip install pyvmomi"**



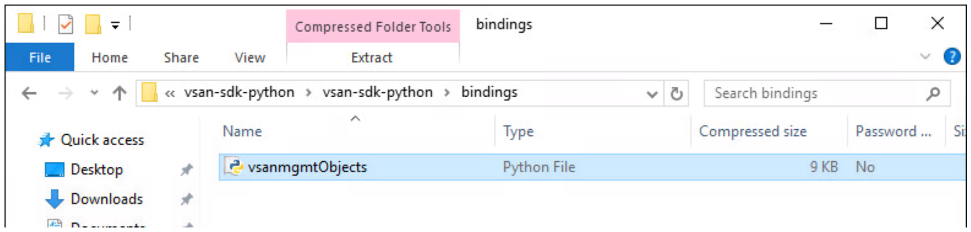
```
Microsoft Windows [Version 10.0.10586]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\vmware>cd \Python27\Scripts

C:\Python27\Scripts>pip install pyvmomi
Collecting pyvmomi
  Downloading pyvmomi-6.0.0.2016.6.tar.gz (218kB)
    100% |################################| 225kB 595kB/s
Collecting requests>=2.3.0 (from pyvmomi)
  Downloading requests-2.10.0-py2.py3-none-any.whl (506kB)
    100% |################################| 512kB 386kB/s
Collecting six>=1.7.3 (from pyvmomi)
  Downloading six-1.10.0-py2.py3-none-any.whl
Installing collected packages: requests, six, pyvmomi
_
```

Once pyVmomi has been installed, the bindings included in the Virtual SAN Management SDK for Python can be extracted from the vsan-sdk-python.zip file



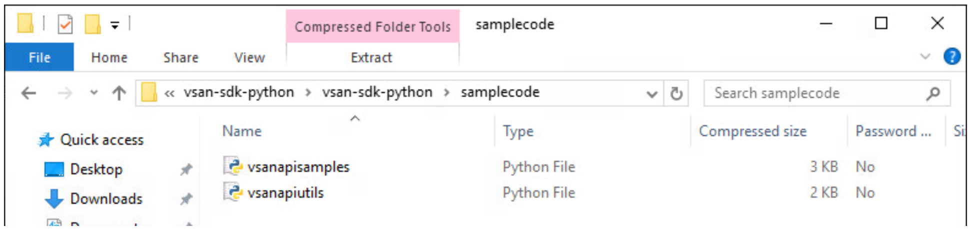
Name	Type	Compressed size	Password ...
bindings	File folder		
docs	File folder		
samplecode	File folder		
README	File	1 KB	No

Copy **vsanmgmtObjects.py** from the bindings folder to Python's Scripts folder (C:\Python27\Scripts).



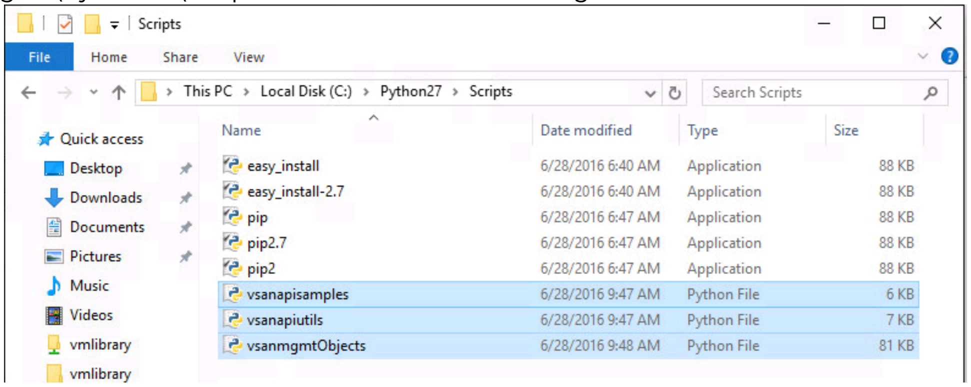
Name	Type	Compressed size	Password ...
vsanmgmtObjects	Python File	9 KB	No

Sample code scripts could also be copied to Python's Scripts folder (C:\Python27\Scripts).



Name	Type	Compressed size	Password ...
vsanapisamples	Python File	3 KB	No
vsanapiutils	Python File	2 KB	No

The resulting C:\Python27\Scripts folder will look something like this:



Name	Date modified	Type	Size
easy_install	6/28/2016 6:40 AM	Application	88 KB
easy_install-2.7	6/28/2016 6:40 AM	Application	88 KB
pip	6/28/2016 6:47 AM	Application	88 KB
pip2.7	6/28/2016 6:47 AM	Application	88 KB
pip2	6/28/2016 6:47 AM	Application	88 KB
vsanapisamples	6/28/2016 9:47 AM	Python File	6 KB
vsanapiutils	6/28/2016 9:47 AM	Python File	7 KB
vsanmgmtObjects	6/28/2016 9:48 AM	Python File	81 KB

Once the bindings and sample code have been copied to the Scripts folder, additional scripts can be easily run from this location.

Virtual SAN 'Recipes'

A few sample 'Recipes' are included in this document to detail the process of how one would go about putting together Virtual SAN Management API for Python scripts together.

These will only be code snippets included in this document. Each recipe will include a link to a competed sample script in the respective summary section.

Important Note: The code samples included in this document are not supported by VMware. The code included is only provided as sample code for the purpose of demonstrating different tasks using the Virtual SAN Management API.

Deploying Virtual SAN

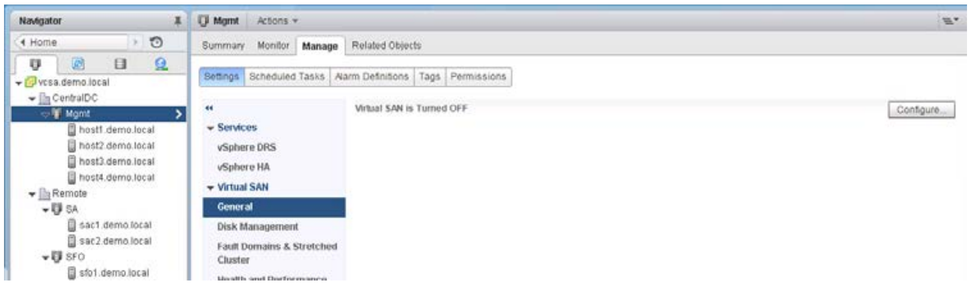
Deploying Virtual SAN can be accomplished easily using the Web Client using the Enable Virtual SAN wizard. The wizard prompts the administrator for some items like whether disks will be claimed manually or automatically, if Deduplication and Compression are to be enabled, which disks will be used, and more.

Specific actions are required to be performed across hosts in the cluster, in a particular order. For example, disks cannot be added to the Virtual SAN datastore until Virtual SAN has been enabled, and hosts have a dedicated network to communicate between each other. Once hosts have been added to a vSphere cluster, and settings like vSphere HA and DRS are configured, a Virtual SAN cluster can be configured.

These tasks are easy individually, but when deploying many Virtual SAN clusters very rapidly, the consistency and repeatability of these tasks rely on the administrator. Manually executing these tasks is time consuming, and is potentially subject to variance if performed inconsistently.

1. Selecting a Cluster for Virtual SAN

Virtual SAN is configured per cluster. Enabling Virtual SAN using the vSphere Web Client is a very simple process. In the Hosts and Clusters view, simply click the cluster that will have Virtual SAN enabled, and go to Settings under the Manage tab.



Because vCenter Server does not have a global option to enable Virtual SAN on specific clusters, or all clusters, the process is manual. Each cluster must be individually selected, and the Virtual SAN Configuration Wizard must be run.

When using the vSphere Web Client authentication is taken care of automatically, and provides a secure session for performing tasks. When using code, the mechanism of connecting to a vCenter Server, passing authentication, and establishing a session must be created.

```
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 upon exit
   atexit.register(Disconnect, si)

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

In the above code, a function, called **getClusterInstance** is called with the Cluster Name, which is passed as a script argument, as well as the mechanism to securely connect to vCenter Server with credentials, which were also passed as arguments.

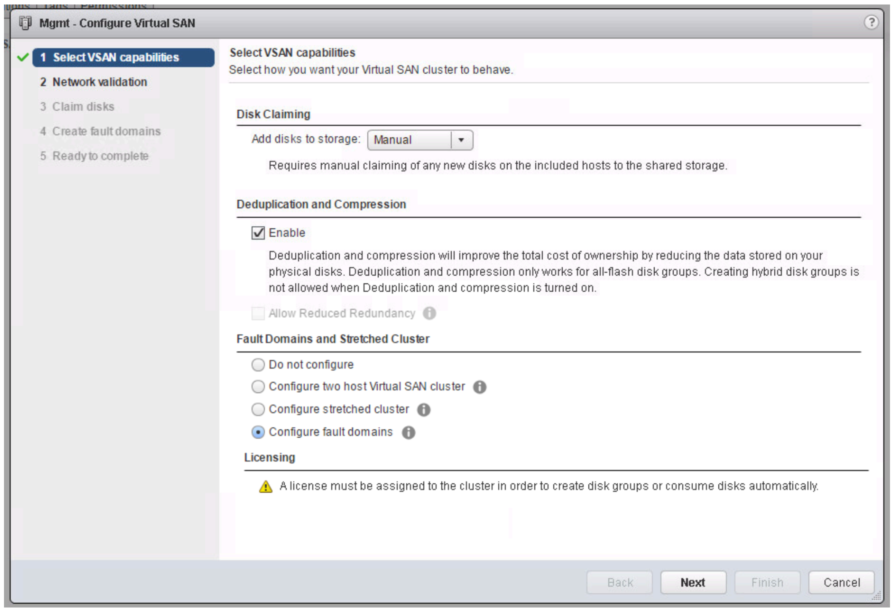
Functions are snippets of code that are often valuable because they operate as smaller programs within a larger program performing a specific action. Functions such as **getClusterInstance** below could likely be reused across many scripts to connect to a specific cluster.

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

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

2. Using the Virtual SAN Configuration Wizard

The Virtual SAN Configuration Wizard is the starting point for enabling Virtual SAN on a vSphere cluster. A lot of work has been done by VMware Engineering to make the task of deploying Virtual SAN very easy. The wizard exposes the steps of choosing Virtual SAN capabilities, ensuring that VMkernel adapters are configured, the ability to choose storage devices, and then complete the setup.

Depending on which capabilities are selected, additional tasks can be performed using the wizard, like claiming disks automatically, enabling Deduplication and Compression and configuring Fault Domains, Stretched Clusters, or 2 Node configurations.



Select VSAN capabilities
Select how you want your Virtual SAN cluster to behave.

Disk Claiming
Add disks to storage: Manual
Requires manual claiming of any new disks on the included hosts to the shared storage.

Deduplication and Compression
[✓] Enable
Deduplication and compression will improve the total cost of ownership by reducing the data stored on your physical disks. Deduplication and compression only works for all-flash disk groups. Creating hybrid disk groups is not allowed when Deduplication and compression is turned on.
[ ] Allow Reduced Redundancy

Fault Domains and Stretched Cluster
( ) Do not configure
( ) Configure two host Virtual SAN cluster
( ) Configure stretched cluster
(●) Configure fault domains

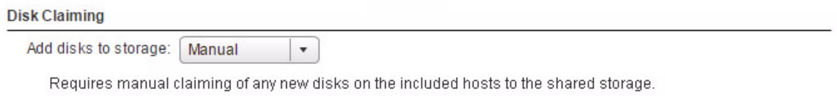
Licensing
A license must be assigned to the cluster in order to create disk groups or consume disks automatically.

The Virtual SAN Configuration Wizard is a launching point for configuring Virtual SAN. For the purpose of this document, we will break down the individual sections as they pertain to different functions.

3. Automatically Claiming Disks

Virtual SAN can be configured to claim disks either automatically or manually. The default method is to manually claim disks. This setting is valid both for the initial configuration, as well as normal operation of Virtual SAN. It is important to understand the behavior of this setting when deploying Virtual SAN.

When disk claiming is set to automatic, disks will be claimed on the best effort of the Virtual SAN Wizard to choose the appropriate types for the **Cache tier** and the **Capacity tier**. This is easy for Hybrid architectures, but could be problematic depending on capacity sizes in an All-Flash configuration if disks that are desired to be used for the **Cache tier** are similar in capacity to those that are desired to be used in the **Capacity tier.** In some cases it may be preferred to claim disks manually.



Disk Claiming
Add disks to storage: Manual
Requires manual claiming of any new disks on the included hosts to the shared storage.

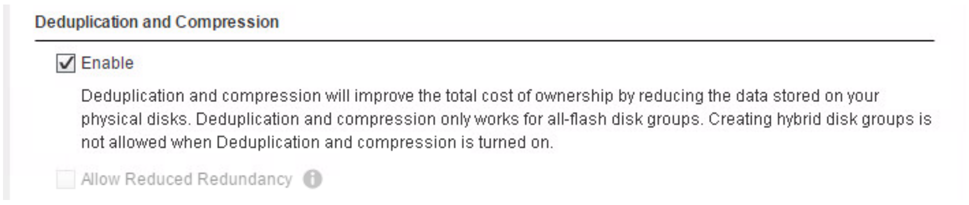
Here is sample code to Manually claim disks.

```
   print 'Enable VSAN by claiming disks manually'
#Build vsanReconfigSpec step by step, it only takes effect after method VsanClusterReconfig is called
   vsanReconfigSpec = vim.VimVsanReconfigSpec( modify=True,
      vsanClusterConfig=vim.VsanClusterConfigInfo( enabled=True,
         defaultConfig=vim.VsanClusterConfigInfoHostDefaultInfo(autoClaimStorage=False)))

# Change the Virtual SAN configuration to claim disks manually
   task = vsanClusterSystem.VsanClusterReconfig(cluster, vsanReconfigSpec)
   vsanapiutils.WaitForTasks([task], si)
```

4. Enabling Deduplication and Compression on All-Flash

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

Enabling Deduplication and Compression is enabled either by a checkbox upon initially creating a Virtual SAN cluster, or can be enabled or disabled after the cluster has been created.


Deduplication and Compression

☑ Enable

Deduplication and compression will improve the total cost of ownership by reducing the data stored on your physical disks. Deduplication and compression only works for all-flash disk groups. Creating hybrid disk groups is not allowed when Deduplication and compression is turned on.

☐ Allow Reduced Redundancy

If a Virtual SAN cluster is created without enabling Deduplication and Compression, the process of enabling it performs a rolling upgrade, which can be time consuming, and in some cases require reduced availability. Choosing to enable Deduplication and Compression at creation time can be beneficial to mitigate the rolling upgrade process.

This can easily be done including code to enable Deduplication and Compression.

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

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

**Note Deduplication and Compression are not enabled independently using the vSphere Web Client. There is no significant performance benefit enabling one and not the other. Enabling both is the only supported configuration. The vSphere Web Client enables both simultaneously.*

5. Networking: Multicast Configuration

Virtual SAN requires Multicast to deliver metadata traffic among cluster nodes. Using the Virtual SAN Wizard will always set the default Multicast addressing. This process is not exposed by the Virtual SAN Wizard.

In some situations, such as one where two independent Virtual SAN clusters share the same Layer 2 network, it is recommended to change the Multicast address for one of the two clusters. This can be accomplished manually using the steps outlined in KB Article 2075451.

We can easily set the Multicast addressing using Python.

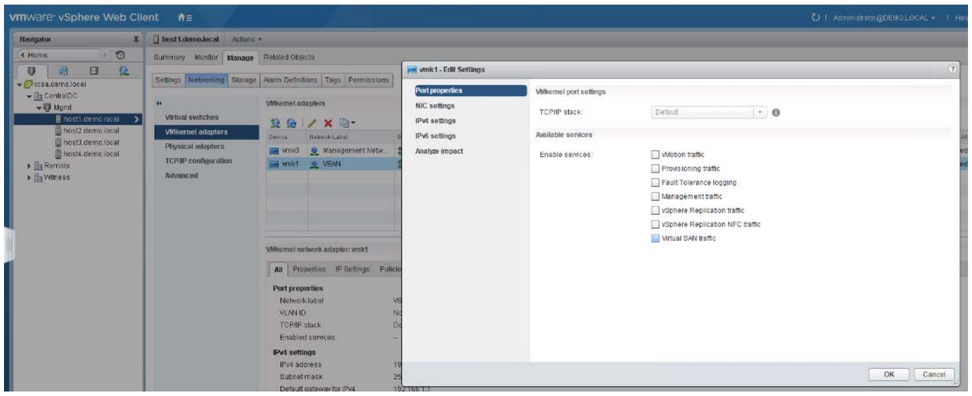
```
# Set the Multicast Addressing
  configInfo = vim.VsanHostConfigInfo(
     networkInfo=vim.VsanHostConfigInfoNetworkInfo(
        port=[vim.VsanHostConfigInfoNetworkInfoPortConfig(
           device=args.vmknic, ipConfig=vim.VsanHostIpConfig(
              upstreamIpAddress='224.1.2.3',
              downstreamIpAddress='224.2.3.4'
           ))]))

    task = hostProps[host]['configManager.vsanSystem'].UpdateVsan_Task(configInfo)
# Execute the task
  vsanapiutils.WaitForTasks(task, si)
```

This could be modified to accept arguments, rather than hard coded addresses, passed at runtime.

6. Networking: VMkernel Selection

Hosts must have a VMkernel interface tagged for Virtual SAN traffic to be able to access a Virtual SAN datastore. This is relatively easy to perform on a single or few hosts, but can be challenging at scale.

To tag a VMkernel interface, a host must be selected, and then the Networking menu from the Management tab selected. Once there the VMkernel interface that will be used for Virtual SAN traffic must be edited and tagged to include Virtual SAN traffic.



vmk1 - Edit Settings

Port properties
NIC settings
IPv4 settings
IPv6 settings
Analyze impact

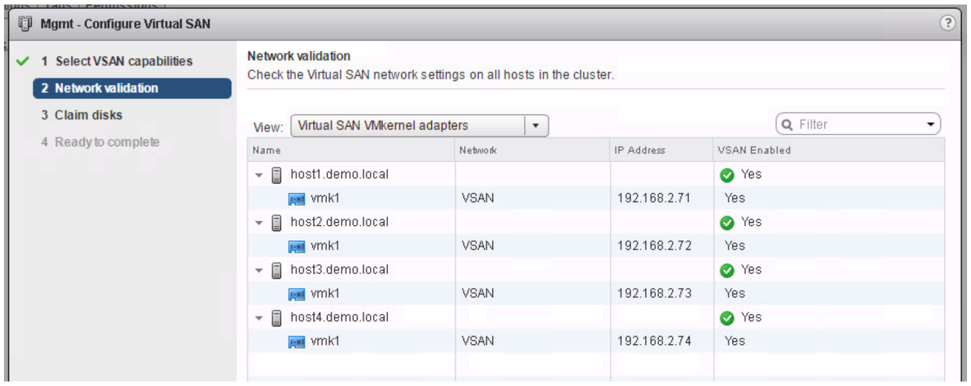
VMkernel port settings

TCP/IP stack: Default

Available services

Enable services:
- ☐ vMotion traffic
- ☐ Provisioning traffic
- ☐ Fault Tolerance logging
- ☐ Management traffic
- ☐ vSphere Replication traffic
- ☐ vSphere Replication NFC traffic
- ☑ Virtual SAN traffic

While this is an easy task, it must be accomplished for each host in the cluster before Virtual SAN is enabled. Once each host has a VMkernel interface tagged for Virtual SAN traffic, during setup the Wizard will indicate that all hosts are properly configured



Mgmt - Configure Virtual SAN

✓ 1 Select VSAN capabilities
2 Network validation
3 Claim disks
4 Ready to complete

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

View: Virtual SAN VMkernel adapters

Name	Network	IP Address	VSAN Enabled
host1.demo.local			✓ Yes
vmk1	VSAN	192.168.2.71	Yes
host2.demo.local			✓ Yes
vmk1	VSAN	192.168.2.72	Yes
host3.demo.local			✓ Yes
vmk1	VSAN	192.168.2.73	Yes
host4.demo.local			✓ Yes
vmk1	VSAN	192.168.2.74	Yes

Today, if hosts do not already have a VMkernel interface tagged for Virtual SAN traffic, the Wizard must be closed, VMkernel interfaces tagged, and the Wizard must be run again.

Selecting VMkernel NICs and enabling Virtual SAN traffic can ensure proper configuration at script execution.

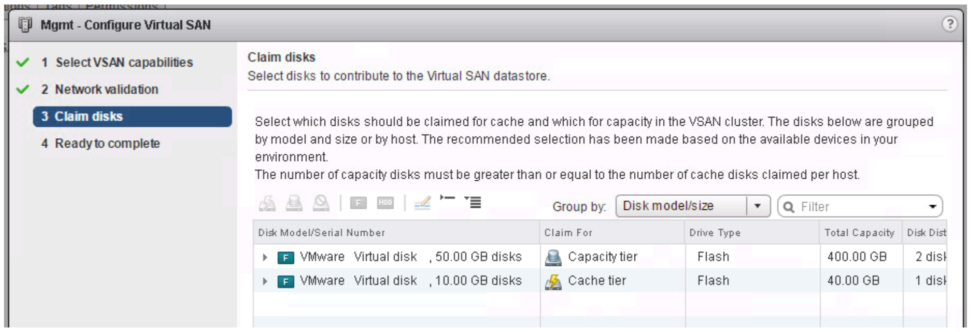
```
# Update the configuration spec for VMkernet Networking
# Enumerate the selected VMkernel interface for each host, and add it to the list of tasks
   for host in hosts:
     print 'Enable VSAN traffic in host {} with {}'.format(hostProps[host]['name'], args.vmknic)
     task = hostProps[host]['configManager.vsanSystem'].UpdateVsan_Task(configInfo)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

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

7. Disk Management and Claiming Disks

Virtual SAN leverages storage devices of several types, depending on the architecture, for the purpose of backing the Virtual SAN datastore. There are different types of devices that Virtual SAN uses, like storage controllers, Solid-State Drives (SSDs), and traditional spinning Hard Disk Drives (HDDs).

Disks can be added to a Virtual SAN cluster upon creation of the cluster, or can be added manually later through Disk Management.

When using the vSphere Web Client, drives that have existing partitions are not listed as eligible to add to the pool of disks used by Virtual SAN. The task of listing eligible disks in the Web Client is done on the backend, and not exposed to the administrator. Once selected, they will be claimed and added to the Virtual SAN cluster.



Claim disks

Select disks to contribute to the Virtual SAN datastore.

Select which disks should be claimed for cache and which for capacity in the VSAN cluster. The disks below are grouped by model and size or by host. The 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	Disk Dist
VMware Virtual disk , 50.00 GB disks	Capacity tier	Flash	400.00 GB	2 disk
VMware Virtual disk , 10.00 GB disks	Cache tier	Flash	40.00 GB	1 disk

We can query each host in the cluster for eligible disks, and include the option to clear partitions on disks that are not eligible. Once all eligible disks have been identified, they can be claimed.

```
  for host in hosts:
# Enumerate find the ineligible disks
    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())
```

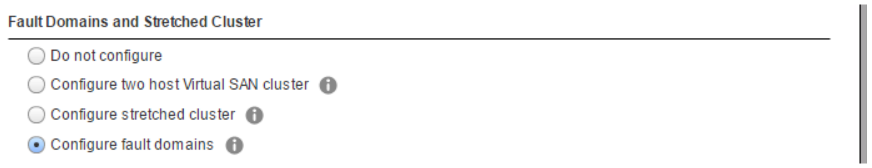
Also notice that in our wizard above that all disks are flash based devices. How do we determine which devices to use for cache, and which devices to use for capacity? Because cache devices are normally smaller than capacity devices, the wizard assigns **Cache tier** for the smaller devices and **Capacity tier** for the larger devices. In a Hybrid configuration, flash devices would default to **Cache tier** and traditional HDDs would be listed as **Capacity tier** devices.

```
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)
```

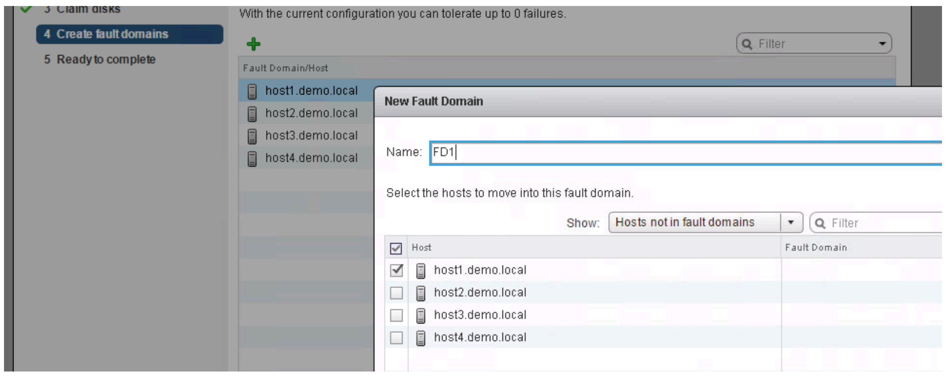
8. Fault Domain Creation

Virtual SAN 6 introduced the ability to logically group hosts into Fault Domains. This feature gives administrators the ability to logically separate Virtual SAN hosts much in the same way that hosts are physically separated.

Grouping one or more hosts into a Fault Domain using the vSphere Web Client can be done upon Virtual SAN cluster creation, or after a cluster has been created.


Fault Domains and Stretched Cluster

- Do not configure
- Configure two host Virtual SAN cluster
- Configure stretched cluster
- [✓ Configure fault domains]



With the current configuration you can tolerate up to 0 failures.

Fault Domain/Host
host1.demo.local
host2.demo.local
host3.demo.local
host4.demo.local

New Fault Domain

Name: FD1

Select the hosts to move into this fault domain.

Show: Hosts not in fault domains

Host	Fault Domain
✓ host1.demo.local	
host2.demo.local	
host3.demo.local	
host4.demo.local	

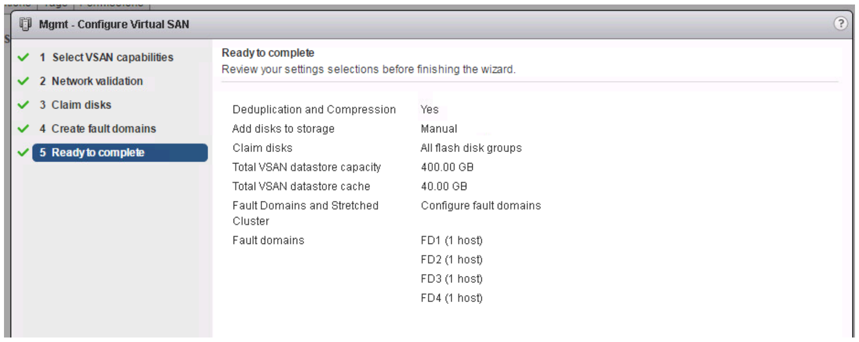
This is a process that must be accomplished by the Virtualization Admin manually.

Fault Domains and assigned hosts can easily be passed as script arguments.

```
  # 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(',')])
        # Add Fault Domains to the Domain Specification
        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)
```

9. Finishing up

When completing a Virtual SAN cluster setup using the Virtual SAN Configuration Wizard, a nice summary screen provides one last review of the proposed configuration before enabling Virtual SAN with the selections a Virtualization Administrator has made.



Ready to complete
Review your settings selections before finishing the wizard.

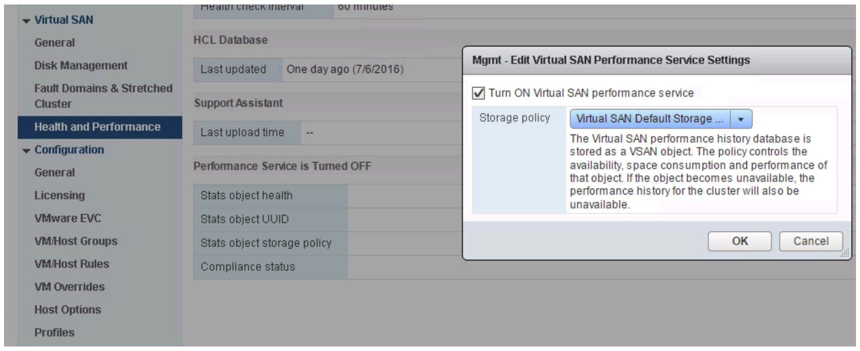
Deduplication and Compression	Yes
Add disks to storage	Manual
Claim disks	All flash disk groups
Total VSAN datastore capacity	400.00 GB
Total VSAN datastore cache	40.00 GB
Fault Domains and Stretched Cluster	Configure fault domains
Fault domains	FD1 (1 host)
	FD2 (1 host)
	FD3 (1 host)
	FD4 (1 host)

This review screen is the last opportunity to make sure that the cluster is configured properly. This is a necessary step when manually configuring Virtual SAN through the vSphere Web Client.

10. Enabling the Performance Service

Virtual SAN 6.2 added a new Performance Service that maintains a database of performance metrics on the Virtual SAN datastore. It must be manually enabled for each Virtual SAN cluster because it is not enabled during initial setup.

To setup the Performance Service, select Health and Performance from the Manage tab of a Virtual SAN cluster.



Health check interval 60 minutes

Last updated One day ago (7/6/2016)

Last upload time --

Mgmt - Edit Virtual SAN Performance Service Settings

☑ Turn ON Virtual SAN performance service

Storage policy: Virtual SAN Default Storage ...

The Virtual SAN 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.

It is even easier using code to enable the Performance Service on Virtual SAN.

```
   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)
```

11. License Assignment

Virtual SAN license assignment isn't handled through the Virtual SAN Configuration Wizard. Virtual SAN clusters are licensed through the vSphere Web Client. Multiple clusters can be licensed simultaneously through the Web Client, but it is still a manual process.

The Virtual SAN Management API allows includes the ability to assign licenses upon deployment of a Virtual SAN cluster.

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

Using code to assign licenses to different clusters provides flexibility when assigning licenses to Virtual SAN. In cases where licensed features are upgraded across a large environment, using code to assign new licensing could be significantly easier than manual license allocation.

Recipe Summary

Deploying Virtual SAN manually is a fairly simple process. While there are some additional settings that change the deployment options, it is still very simple. Despite the process being simplistic in nature, there are always opportunities for error when manually performing processes at scale.

The above code snippets show how easy it is to automate the deployment of Virtual SAN using code consistently and repeatedly at both large and small scale.

The individual code snippets are available as a single Python script from the VMware Developer Center at the following URL.

https://developercenter.vmware.com/samples?id=1133

Configuring Virtual SAN Stretched Clusters or 2 Node

In Virtual SAN 6.1, VMware introduced Stretched Clusters and 2 Node configurations. When deploying Stretched Cluster or 2 Node configurations, one or more hosts are located in one fault domain, an equal number of hosts in a second fault domain, and host that provides the Witness component for the cluster.

Stretched Clusters and 2 Node configurations have a few distinct characteristics from traditional Virtual SAN clusters.

- Effectively 2 comprised of between 1 and 15 nodes per site
- Require a Witness Host to make up the 3rd site for metadata content
  This can be a physical host or the freely available Virtual SAN Witness Appliance

Important Note: The previous recipe is used starting with Step 3. Again, this is sample code, not supported by VMware.

1. Deploying a Virtual SAN Witness Appliance

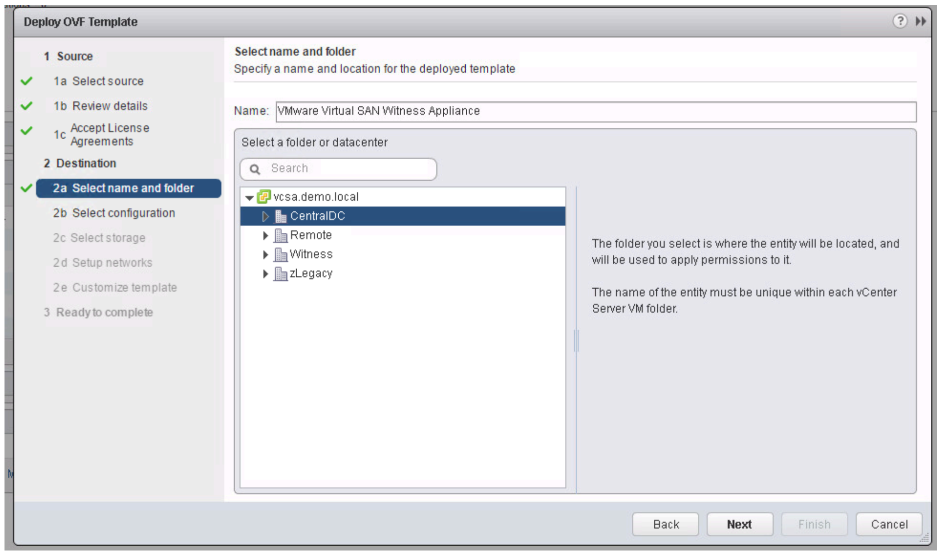
The first step required in supporting Virtual SAN Stretched Clusters or 2 Node configurations, is to either configure a physical host, or a Virtual SAN Witness Appliance to perform the Witness responsibilities.

The Virtual SAN Witness Appliance is great alternative to using a physical ESXi host due to not requiring a dedicated license or dedicated physical disks for Virtual SAN metadata. Virtual Appliances are often provided as an OVA file, or an OVF file, along with additional components such as a manifest and virtual disk files.

There are several methods to import Virtual Appliances into a vSphere environment. These methods include options to import Virtual Appliances as a menu item in the vSphere Web Client (with the help of the vSphere Client Integration Plugin), or the legacy vSphere Client. There is also an OVF Tool, provided for various operating systems, that allows for importing Virtual Appliances from a command line.

The Virtual SAN Witness Appliance can be downloaded from VMware.com:
https://my.vmware.com/web/vmware/info/slug/datacenter_cloud_infrastructure/vmware_virtual_san/6_2#drivers_tools

To import a Virtual Appliance from the vSphere Web Client, an administrator only needs to right click on the cluster the appliance will be deployed to, designate the Virtual Appliance source, select additional items (such as host, datastore, network, etc.), as well as any prompts for settings specific to the Virtual Appliance.



Select name and folder
Specify a name and location for the deployed template

Name: VMware Virtual SAN Witness Appliance

Select a folder or datacenter

The folder you select is where the entity will be located, and will be used to apply permissions to it.

The name of the entity must be unique within each vCenter Server VM folder.

Remember that previously mentioned tools can natively, or through the use of a plugin, upload Virtual Appliances. Deploying the Virtual SAN Witness Appliance can also be accomplished as part of a Python script, but requires code to handle the upload process.

The Virtual SAN Witness Appliance is available as an OVA. And OVA is a self-contained file that is comprised of several files. Each of these files must be uploaded to vCenter Server during the OVF Deployment process.

To upload multiple files, we first must be able to upload a single file.

```
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 the function is defined to upload a single file, another can be created to upload multiple files.

```
def uploadFiles(fileItems, lease, ovfURL, vmName=None, log=None):
   '''
   Upload witness vm's vmdk files to vCenter by using 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)
```

Uploading files is only a small portion of Deploying a Virtual Appliance. Additional tasks such as cluster, host, network, and datastore placement are required, as well as passing any parameters that the Virtual Appliance requires.

```
def DeployWitnessOVF(ovfURL, si, host, vmName, dsRef, vmFolder, vmPassword=None, network=None, log=None):
   '''
   Deploy witness VM to vCenter,  the import process consists of two steps:
   1>Create the VMs and/or vApps that make up the entity.
   2>Upload virtual disk contents.
   @param ovfURL: ovf source url
   @param si: Managed Object ServiceInstance
   @param host: HostSystem which the VM located
   @param vmName: VM name
   @param dsRef: Datastore which the VM located
   @param vmFolder: Folder which the VM belong to
   @param vmPassword: Password for the VM
   @param network: Managed Object Network of the VM
   @param log:
   @return:
   '''
   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) > 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
  uploadFiles(res.fileItem, lease, ovfURL, vmName, log)
  lease.Complete()

  return lease.info.entity
```

The **DeployWitnessOVF** function takes care of setting networking, configuring the supplied password as one of vApp options, and proper placement of the appliance on a specific host or resource pool. Fortunately, the Virtual SAN Witness Appliance only requires a password as an additional argument that needs to be passed.

The **DeployWitnessOVF** function will parse the contents of the OVF, but does not have the capability to parse an OVA, which the Witness Appliance is downloaded as. The Witness OVA file will have to be extracted from an OVA to a folder containing the OVF and other required files.

The OVA file is essentially a .tar archive, that can be extracted easily using a wide variety of tools. For the purpose of this paper, TarTool will be used due to its simplicity and command line availability. **No specific recommendation of this tool should be inferred by the reader, as many tools provide this functionality.*

After deploying the Virtual SAN Witness Appliance, the VM must be powered on and some tasks need to take place. Putting these together, the process of deploying a Witness appliance could look something like this:

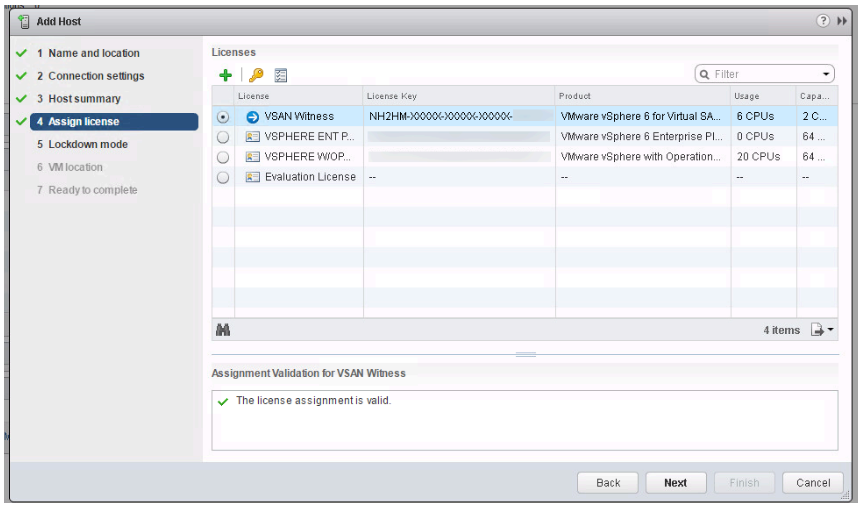
```
    print 'Start to add virtual witness host'
    '''
    Steps add virtual witness, rather than a dedicated physical ESXi host to be a witness host,
    VMware has developed the VSAN witness appliance to take care of the witness requirements
    1) Deploy witness VM specifying host, storage, & network 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
```

2. Adding the Witness Appliance to vCenter Server

The Virtual SAN Witness Host, either in physical or Virtual Appliance form, must be added to the vCenter environment where the Virtual SAN Stretched Cluster or 2 Node configuration is running. It is important to remember that the Witness Host **cannot** be a member of the Virtual SAN cluster.

Hosts can be manually added to a vSphere Datacenter, or clusters within that datacenter, using the vSphere Web Client, vSphere Client, or other scripting tools. It is important to note that when using the legacy vSphere Client, adding a Virtual SAN Witness Appliance as a host will not properly assign the Virtual SAN Witness license. As a result, if manually adding a Witness, using the vSphere Web Client is a preferred method versus the legacy vSphere Client.



	License	License Key	Product	Usage	Capa...
●	VSAN Witness	NH2HM-XXXXX-XXXXX-XXXXX-[redacted]	VMware vSphere 6 for Virtual SA...	6 CPUs	2 C...
○	VSPHERE ENT P...	[redacted]	VMware vSphere 6 Enterprise Pl...	0 CPUs	64 ...
○	VSPHERE W/OP...	[redacted]	VMware vSphere with Operation...	20 CPUs	64 ...
○	Evaluation License	--	--	--	--

Assignment Validation for VSAN Witness

✓ The license assignment is valid.

A function is required to facilitate the process of adding the Witness Appliance as a host in vCenter.

```
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, user doesn't need input the host's thumbprint of the SSL certificate, the logic below

    #will do 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)
```

Adding the host becomes relatively easy using the AddHost function created above.

```
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)
```

3. Configuring an existing cluster as a Stretched Cluster or 2 Node

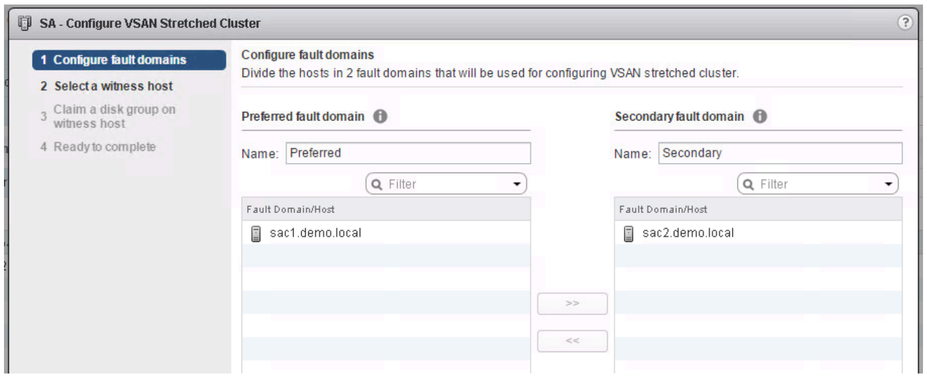
The process of configuring a Virtual SAN cluster as either a Stretched Cluster or 2 Node configuration can be done upon initial creation, or after a Virtual SAN cluster has been initially setup.

To convert an existing cluster to a Stretched or 2 Node cluster, the process is as follows.

- Choose which host(s) will be placed in the Preferred Fault Domain
- Choose which host(s) will be placed in the Secondary Fault Domain
- Select a Witness host
    - o Choose cache and capacity disks for the Witness host
- Complete the setup

Reusing much of the code from the Deploy recipe, some more tasks are still required, including setting up Fault Domains, designating a Witness Host, and selecting the Witness Host disks that will be used.

Configuring Fault Domains


SA - Configure VSAN Stretched Cluster
1 Configure fault domains
2 Select a witness host
3 Claim a disk group on witness host
4 Ready to complete
>>
<<

Configure fault domains

Divide the hosts in 2 fault domains that will be used for configuring VSAN stretched cluster.

Preferred fault domain

Name: Preferred

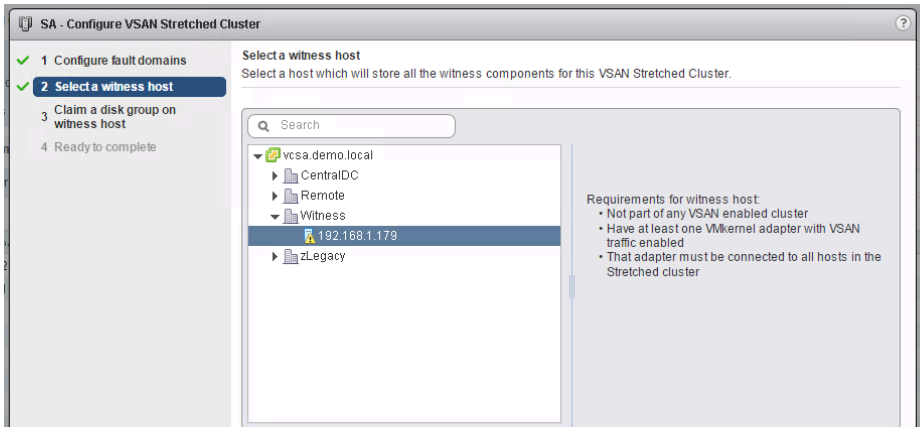
Fault Domain/Host: sac1.demo.local

Secondary fault domain

Name: Secondary

Fault Domain/Host: sac2.demo.local

Selecting a Witness Host

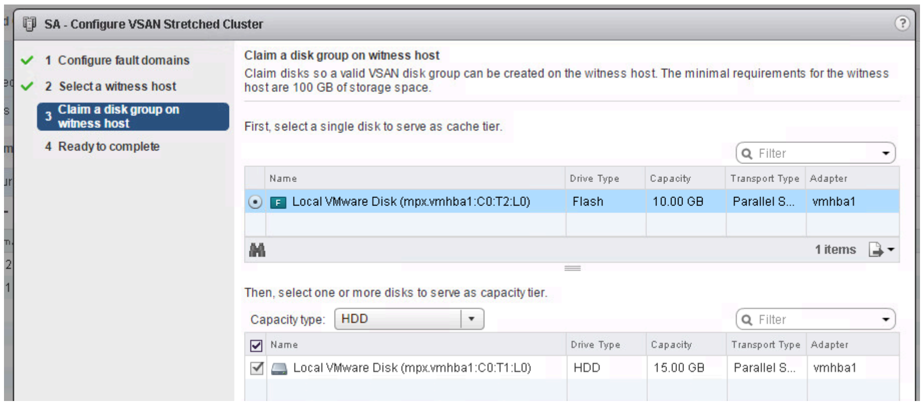


Select a witness host

Select a host which will store all the witness components for this VSAN Stretched Cluster.

Requirements for witness host:
- Not part of any VSAN enabled cluster
- Have at least one VMkernel adapter with VSAN traffic enabled
- That adapter must be connected to all hosts in the Stretched cluster

Selecting Witness Host disks



SA - Configure VSAN Stretched Cluster

1 Configure fault domains
2 Select a witness host
3 Claim a disk group on witness host
4 Ready to complete

Claim a disk group on witness host

Claim disks so a valid VSAN disk group can be created on the witness host. The minimal requirements for the witness host are 100 GB of storage space.

First, select a single disk to serve as cache tier.

	Name	Drive Type	Capacity	Transport Type	Adapter
●	Local VMware Disk (mpx.vmhba1:C0:T2:L0)	Flash	10.00 GB	Parallel S...	vmhba1

1 items

Then, select one or more disks to serve as capacity tier.

Capacity type: HDD

☑	Name	Drive Type	Capacity	Transport Type	Adapter
☑	Local VMware Disk (mpx.vmhba1:C0:T1:L0)	HDD	15.00 GB	Parallel S...	vmhba1

To setup Fault Domains, we can enumerate the hosts in the cluster, answer yes or no for which Fault Domain we choose to place the hosts into, saving that information in 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   )
```

Like in the previous recipe, only eligible disks can be used for Virtual SAN devices. Putting these in an array will allow for them to be claimed at the time of cluster configuration.

```
  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)
        else:
          biggerSsds.append(ssd)
      diskMapping = vim.VsanHostDiskMapping(
          ssd = smallSsds[0],
          nonSsd = biggerSsds
        )
```

Once hosts have been put into two Fault Domain arrays, and eligible disks have been determined for the Witness host, reconfiguring the Cluster can occur.

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

Recipe Summary

Just as deploying Virtual SAN was in the first recipe, Stretched Cluster and 2 Node configurations can also be easily and repeatedly automated with code.

The above code snippets show how easy it is to automate Stretched Cluster and 2 Node configurations of Virtual SAN using code consistently and repeatedly at both large and small scale.

The individual code snippets are available as a single Python script from the VMware Developer Center at the following URL.

https://developercenter.vmware.com/samples?id=1134

Performing a Virtual SAN On-Disk Upgrade

Upgrading vSphere hosts is a process that has been around for quite some time. There are several methods of upgrading vSphere hosts like Upgrade Manager, esxcli, and more. These existing methods will upgrade ESXi hosts that are running Virtual SAN, but will not upgrade the Virtual SAN On-Disk format.

The VSAN-FS On-Disk format initially introduced in Virtual SAN 5.5 is version 1.0. Virtual SAN 6 introduced version 2.0. Version 2.0 of the Virtual SAN On-Disk format brought about changes, allowing for configurations such as All-Flash architectures, Stretched Clusters, support of the vsanSparse format for snapshots, and more. With the introduction of Virtual SAN 6.2, version 3.0 was introduced. Version 3.0 supports even more functionality with Deduplication and Compression as well as Erasure Coding for All-Flash architectures, and features including software checksums and IOPS limits for all architectures.

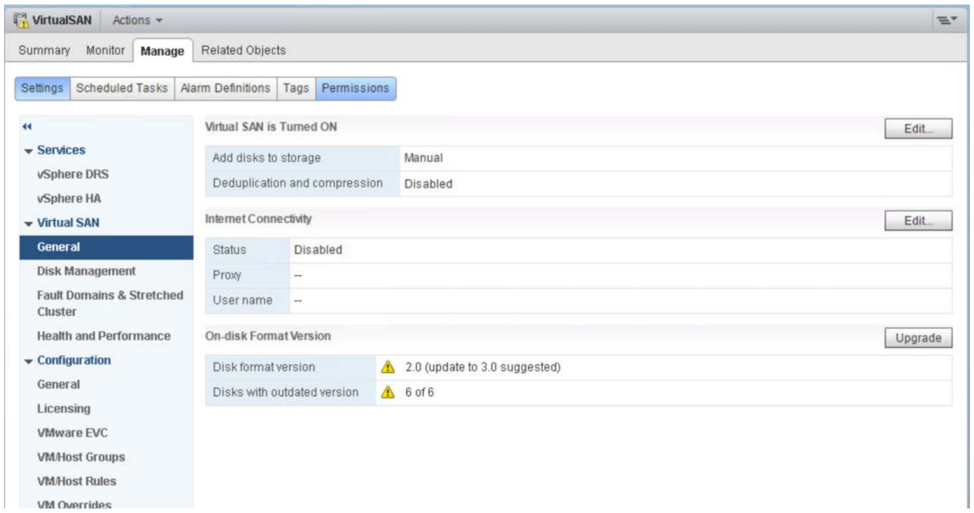
Depending on the configuration and state of a Virtual SAN cluster, performing an On-Disk Format upgrade can have different requirements. This is because the On-Disk format upgrade process essentially evacuates data from each Disk Group in a host, removes and recreates the Disk Group in the new format, and migrates data back, all in a rolling upgrade process.

This recipe will cover some of the things to consider when programmatically updating the On-Disk format of Virtual SAN.

1. Determining the current On-Disk format

Before attempting to perform a Virtual SAN On-Disk upgrade, it is important to know what the current format is, as well as the latest supported format for the ESXi build that Virtual SAN cluster is running.

This is automatically visible within the vSphere Web Client from the **Manage** tab under **Settings** for a vSphere cluster running Virtual SAN.



Virtual SAN is Turned ON

Add disks to storage	Manual
Deduplication and compression	Disabled

Internet Connectivity

Status	Disabled
Proxy	--
User name	--

On-disk Format Version

Disk format version	⚠ 2.0 (update to 3.0 suggested)
Disks with outdated version	⚠ 6 of 6

Notice in the graphic that the Disk format version is 2.0, yet upgradeable to 3.0. Additionally, all the disks, 6 of 6, are running the outdated version.

The vSphere Web Client easily shows this information, but requires a Virtualization Admin to manually check the status of the On-Disk format for the cluster.

We'll connect to the cluster, and determine the highest On-Disk Format supported by 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)
```

A function, provides the ability to compare the existing On-Disk 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
```

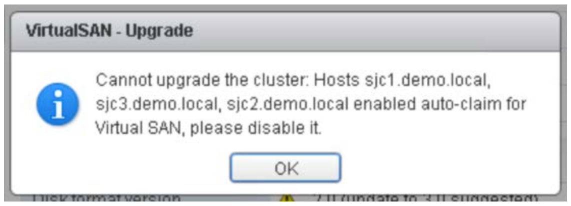
We can gather each of the disk group member devices into diskMappings, then pass them into the hasOlderVersionDisks function to determine if an upgrade is necessary or not.

```
  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)
```

In cases where a cluster is already at the highest level support, there is obviously no need to perform an upgrade. In cases where an upgrade is necessary, there are some additional tasks that need to be performed, depending on the cluster's configuration.

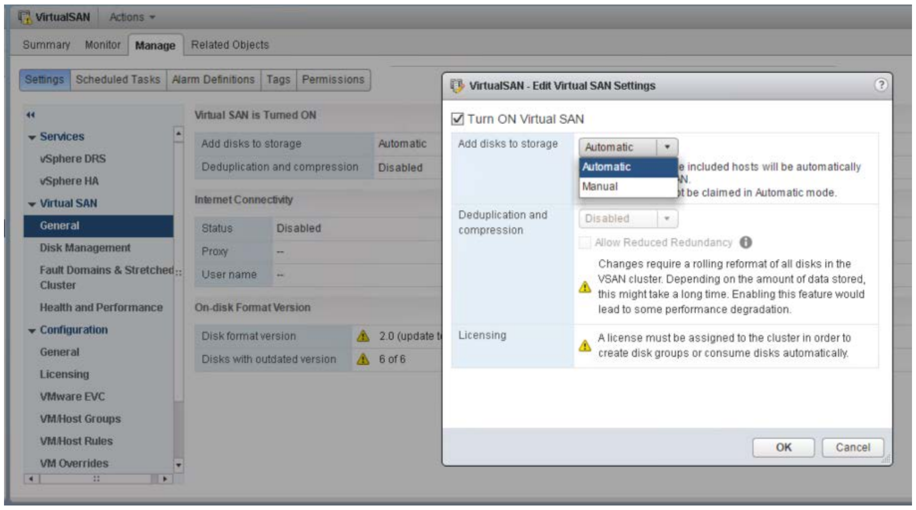
2. Role of Disk Claiming in Virtual SAN On-Disk Format Upgrades

If Virtual SAN is configured to automatically claim disks, an On-Disk format upgrade cannot occur. In the vSphere Web Client, if this is the case, an error will occur.


VirtualSAN - Upgrade

Cannot upgrade the cluster: Hosts sjc1.demo.local, sjc3.demo.local, sjc2.demo.local enabled auto-claim for Virtual SAN, please disable it.

A Virtualization Admin will have to go and change the Disk Claiming method from automatic to manual.



Virtual SAN is Turned ON

Add disks to storage: Automatic
Deduplication and compression: Disabled

Internet Connectivity

Status: Disabled
Proxy: --
User name: --

On-disk Format Version

Disk format version: 2.0 (update t
Disks with outdated version: 6 of 6

VirtualSAN - Edit Virtual SAN Settings

☑ Turn ON Virtual SAN

Add disks to storage: Automatic (Automatic / Manual) … e included hosts will be automatically … N. … t be claimed in Automatic mode.

Deduplication and compression: Disabled
Allow Reduced Redundancy
Changes require a rolling reformat of all disks in the VSAN cluster. Depending on the amount of data stored, this might take a long time. Enabling this feature would lead to some performance degradation.

Licensing: A license must be assigned to the cluster in order to create disk groups or consume disks automatically.

Having to make this change across more than one cluster can be very time consuming process. This can be better accomplished through an upgrade script checking for automatic disk claiming, and changing the cluster's disk claiming method to manual.

```
    autoClaimChanged = False
    if vsanConfig.defaultConfig.autoClaimStorage:
      print 'autoClaimStorage should be set to false before upgrade VSAN disks'
      autoClaimChanged = True
      vsanReconfigSpec = vim.VimVsanReconfigSpec(
        modify = True,
        vsanClusterConfig = vim.VsanClusterConfigInfo(
          defaultConfig = vim.VsanClusterConfigInfoHostDefaultInfo(
            autoClaimStorage = False)))

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

3. On-Disk Format Upgrade Preflight Check

If Virtual SAN is configured to automatically claim disks, an On-Disk format upgrade cannot occur. In the vSphere Web Client, if this is the case, an error will occur.

When an upgrade is selected from the vSphere Web Client, a "preflight" check occurs. While this is handled by the vSphere Web Client, it must also be done in script.

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

***Note Deduplication and Compression are not enabled independently using the vSphere Web Client. There is no significant performance benefit enabling one and not the other. Enabling both is the only supported configuration. The vSphere Web Client enables both simultaneously.*

If there are any issues with the preflight check, they must be resolved beforehand. We'll need to indicate this in our script so the Virtualization Admin can address accordingly.

```
      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
```

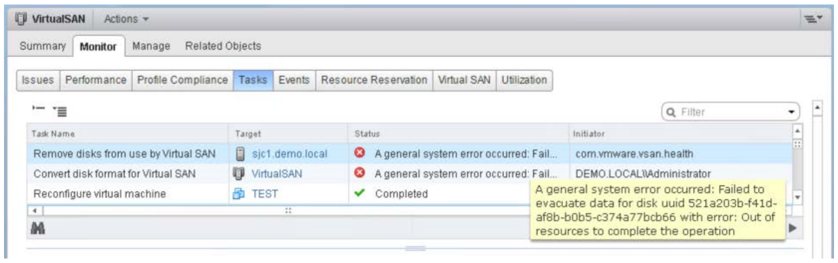
4. Reduced Redundancy During On-Disk Format Upgrades

Virtual SAN On-Disk format upgrades require the existing VM storage policies to be satisfied during the upgrade process.

In a 3 node cluster, a Failure To Tolerate =1 policy requires 3 nodes. Bringing a node offline to perform an upgrade would create a situation of reduced redundancy.

This doesn't mean that 3 node clusters can never be upgraded. It only means that they can only be upgraded with reduced redundancy.

By default, the upgrade process does not allow reduced redundancy. Attempting to perform an On-Disk format upgrade without sufficient spare resources will fail.



Task Name	Target	Status	Initiator
Remove disks from use by Virtual SAN	sjc1.demo.local	A general system error occurred: Fail...	com.vmware.vsan.health
Convert disk format for Virtual SAN	VirtualSAN	A general system error occurred: Fail...	DEMO.LOCAL\\Administrator
Reconfigure virtual machine	TEST	✓ Completed	

A general system error occurred: Failed to evacuate data for disk uuid 521a203b-f41d-af8b-b0b5-c374a77bcb66 with error: Out of resources to complete the operation

In the case where there are not enough Virtual SAN resources to satisfy a VM storage policy, such as a 3 node cluster with FTT=1 using mirroring, a reduced redundancy flag must be set. There is no way to accomplish this through the vSphere Web Client, and Virtualization Admins are required to perform the upgrade from the Ruby vSphere Console (RVC). This process is detailed in KB 2113221 for the Version 1 to Version 2 upgrade. The Version 3 upgrade is similar.

If performing an On-Disk format upgrade using a Python script, allowing the upgrade to occur with the reduced redundancy flag can be set as part of initiating the upgrade.

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

5. Enabling Deduplication and Compression during an On-Disk Format Upgrade

If Virtual SAN is configured to automatically claim disks, an On-Disk format upgrade cannot occur. In the vSphere Web Client, if this is the case, an error will occur.

When Virtual SAN 6.2 hosts continue to operate with the Version 2.0 On-Disk format, changing the deduplication and compression setting to enabled will automatically initiate the upgrade process to the Version 3.0 On-Disk format.

With that being stated, when upgrading from the Version 2.0 format to Version 3.0, deduplication and compression may or may not be enabled simultaneously. However, if an administrator simply chooses to upgrade to the new Version 3.0 format, the process to enable deduplication and compression would then have to be run afterwards, as a separate rolling upgrade.

This process is not explicitly detailed, and scripting the process could easily accommodate both upgrading and enabling deduplication and compression simultaneously.

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

Recipe Summary

In some cases, it may be significantly easier to execute a simple script to accomplish more tasks than to manually execute equivalent tasks from the vSphere Web Client, such as upgrading the On-Disk format with reduced redundancy, while enabling deduplication and compression simultaneously.

The above code snippets show how easy it is to upgrade the On-Disk format of a Virtual SAN cluster using a simple script along with specific parameters like "--enabledc" and "--reduced-redundancy".

The individual code snippets are available as a single Python script from the VMware Developer Center at the following URL.

https://developercenter.vmware.com/samples?id=1135

Document Summary

As stated before, the code examples listed in this document are for the purpose of illustrating some capabilities using Python with the Virtual SAN Management API. These code examples are not supported by VMware.

Be sure to visit the VMware Developercenter at https://Developer Center.vmware.com/ for more examples of code using the Virtual SAN Management API and other VMware APIs.

References

Additional Documentation

For more information about VMware Virtual SAN, please visit the product pages at http://www.vmware.com/products/virtual-san

Below are some links to online documentation:

Product Documentation

Virtual SAN 6.2 Design and Sizing Guide

Virtual SAN 6.2 Network Design Guide
Virtual SAN 6.2 Stretched Cluster & 2 Node Guide

VMware Virtual SAN Community

Support Knowledge base

VMware vSphere and VMware vCenter Server Resources:

Product Overview

Product Documentation

VMware Contact Information

For additional information or to purchase VMware Virtual SAN, VMware's global network of solutions providers is ready to assist. If you would like to contact VMware directly, you can reach a sales representative at 1-877-4VMWARE (650-475-5000 outside North America) or email sales@vmware.com. When emailing, please include the state, country, and company name from which you are inquiring.

About the Author

This cookbook was put together using content from various resources from Virtual SAN Engineering.

Jase McCarty is a Staff Technical Marketing Architect at VMware with a focus on storage solutions. He has been in the Information Technology field for over 25 years, with roles on both the customer and vendor side. Jase has Co-Authored two books on VMware virtualization, routinely speaks technology focused user group meetings, and has presented at VMworld and EMC World.

Follow Jase on Twitter: @jasemccarty